State-of-the-Art Survey

LNCS 13201

Hannah Bast
Claudius Korzen
Ulrich Meyer
Manuel Penschuck (Eds.)

Algorithms for Big Data

DFG Priority Program 1736

OPEN ACCESS

Springer

Hannah Bast · Claudius Korzen ·
Ulrich Meyer · Manuel Penschuck (Eds.)

Algorithms for Big Data

DFG Priority Program 1736

Springer

Editors

Hannah Bast
University of Freiburg
Freiburg im Breisgau, Germany

Claudius Korzen
University of Freiburg
Freiburg, Germany

Ulrich Meyer
Goethe University Frankfurt
Frankfurt, Germany

Manuel Penschuck
Goethe University Frankfurt
Frankfurt, Germany

Goethe-Universität Frankfurt am Main

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-031-21533-9 ISBN 978-3-031-21534-6 (eBook)
https://doi.org/10.1007/978-3-031-21534-6

Preface

Computer systems pervade all parts of human activity: transportation systems, energy supply, medicine, the whole financial sector, and modern science have become unthinkable without hardware and software support. As these systems continuously acquire, process, exchange, and store data, we live in a big-data world where information is accumulated at an exponential rate.

The urgent problem has shifted from collecting enough data to dealing with its impetuous growth and abundance. In particular, data volumes often grow faster than the transistor budget of computers as predicted by Moore's law (i.e., doubling every 18 months). On top of this, we cannot any longer rely on transistor budgets to automatically translate into application performance, since the speed improvement of single processing cores has basically stalled and the requirements of algorithms that use the full memory hierarchy get more and more complicated. As a result, algorithms have to be massively parallel using memory access patterns with high locality. Furthermore, an x-times machine performance improvement only translates into x-times larger manageable data volumes if we have algorithms that scale nearly linearly with the input size. All these are challenges that need new algorithmic ideas. Last but not least, to have maximum impact, one should not only strive for theoretical results, but intend to follow the whole algorithm engineering development cycle consisting of theoretical work followed by experimental evaluation.

The "curse" of big data in combination with increasingly complicated hardware has reached all kinds of application areas: genomics research, information retrieval (web search engines, ...), traffic planning, geographical information systems, or communication networks. Unfortunately, most of these communities do not interact in a structured way even though they are often dealing with similar aspects of big-data problems. Frequently, they face poor scale-up behaviour from algorithms that have been designed based on models of computation that are no longer realistic for big data.

About the SPP 1736

This volume surveys the progress in selected aspects of this important and growing field. It emerged from a research program established by the German Research Foundation (DFG) as priority program SPP 1736 on Algorithmics for Big Data (https://www.big-data-spp.de) in 2013 where researchers from theoretical computer science worked together with application experts in order to tackle some of the problems discussed above.

The research program was prepared collaboratively by Susanne Albers, Hannah Bast, Kurt Mehlhorn, Ulrich Meyer (coordinator), Eugene Myers, Peter Sanders, Christian Scheideler, and Martin Skutella. The first meetings took place in Frankfurt/Main in 2012. Subsequently a grant proposal was worked out, submitted to the DFG on October 15, and the program was granted in the spring meeting of the DFG

Senat in 2013. The duration of the program was six years, divided into two periods of three years each.

A nationwide call for the individual projects attracted over 40 proposals out of which an international reviewer panel selected 15 funded research projects plus a coordination project (totalling about 20 full PhD student positions) by the end of 2013. Additionally, a few more projects with their own funding were associated in order to benefit from collaboration and joint events (workshops, PhD meetings, summer schools etc.) organised by the SPP. The members of the priority programme produced about 300 publications with more than 8200 citations by May 2022.

About This Book

The chapters of this volume summarize results of projects realized within the program and survey-related work. More than half of them centrally deal with various aspects of algorithms for large and complex networks:

- In "*Algorithms for Large-Scale Network Analysis and the NetworKit Toolkit*" (Chapter 1) *Eugenio Angriman, Alexander van der Grinten, Michael Hamann, Henning Meyerhenke, and Manuel Penschuck* survey SPP contributions to a scalable software library for the analysis of huge networks. While their focus is on recent algorithmic contributions in the areas of centrality computations, community detection, and sparsification, they also cover aspects such as current software engineering principles of the project and ways to visualize network data within a NetworKit-based workflow.
- In "*Generating Synthetic Graph Data from Random Network Models*" (Chapter 2) *Ulrich Meyer and Manuel Penschuck* report on novel randomized graph instance generation algorithms which have been developed in SPP collaborations. The described implementations heavily exploit parallelism and/or cache-efficiency, and most of them have been integrated into NetworKit, too. Furthermore, several generators have been used to supplement experimental campaigns of SPP works described in subsequent chapters including the following.
- In the two chapters "*Increasing the Sampling Efficiency for the Link Assessment Problem*" (Chapter 3) and "*A Custom Hardware Architecture for the Link Assessment Problem*" (Chapter 4) *André Chinazzo, Christian De Schryver, Katharina Zweig, and Norbert Wehn* provide an in-depth treatment of a specific network motif search problem—both from an algorithmic and a hardware focused point of view. A link assessment (LA) algorithm can be used to clean up large network data sets with noisy data.

Using instances of a particular type of random graphs discussed in the network generation chapter as a null model, the LA algorithm evaluates the structural similarities between the nodes, and thus differentiates meaningful relationships between nodes from noisy ones. After a detailed discussion of the algorithmic foundations (Chapter 3), the authors present the design of a dedicated hardware accelerator (Chapter 4) for solving the LA problem, which—compared to an Intel cluster—uses 38$\times$ less memory and is 1030$\times$ more energy efficient.

- In "*Graph-Based Methods for Rational Drug Design*" (Chapter 5) *Andre Droschinsky, Lina Humbeck, Oliver Koch, Nils M. Kriege, Petra Mutzel, and Till Schäfer* disuss computational methods for the goal-directed development of new drugs. The connection to graphs is based on the frequently valid assumption that chemical molecules with similar structure also show similar effects in a drug. Hence, molecules are modelled as graphs with attributes and large-scale graph algorithms for similarity search and clustering come into play. The authors provide an overview of recent results with a focus on the search for maximum common subgraphs and their extension to domain specific requirements.
- In "*Recent Advances in Practical Data Reduction*" (Chapter 6) *Faisal N. Abu-Khzam, Sebastian Lamm, Matthias Mnich, Alexander Noe, Christian Schulz, and Darren Strash* discuss recent algorithm engineering work in the area of fixed-parameter tractable NP-hard problems. They survey recent trends in data reduction engineering results for selected problems in NP (like Independent Sets, Vertex Cover, Treewidth, Steiner Trees, etc.) and P (Minimum Cut and Matching). Furthermore, the authors describe techniques that may be useful for future implementations and list a number of open problems and research questions.
- In "*Skeleton-based Clustering by Quasi-Threshold Editing*" (Chapter 7) *Ulrik Brandes, Michael Hamann, Luise Häuser, and Dorothea Wagner* report on SPP work for community detection on real-world graphs. They extend an earlier approach by Nastos and Gao who proposed to view community detection as a graph modification problem where the input is to be transferred into a quasi-threshold graph with a minimum number of edge additions and deletions and use its resulting connected components to determine the clustering. As minimizing the number of edit steps is NP hard, existing solutions rely on heuristics. The authors of the chapter introduce a new linear time heuristic for the inclusion-minimal variant of this edit problem and present improvements for the resulting clustering both in terms of running time and quality.
- In "*The Space Complexity of Undirected Graph Exploration*" (Chapter 8) *Yann Disser and Max Klimm* consider a setting where an agent with small memory has to visit all vertices of a huge graph at least once. The n vertices are indistinguishable for the agent but at least the edges have a locally unique color, which can be exploited for the traversal. The authors revisit results for this setting showing that $\Theta(\log n)$ bits of memory are necessary and sufficient for an agent to explore any graph with n vertices. Subsequently they provide SPP results for collaborative exploration using several agents each having sublogarithmic memory.

The topics of the chapters in the second part of this volume range from challenges in scalable cryptography, data streams, and energy-efficient scheduling to generic optimization and text (pre)processing including applications:

- In "*Scalable Cryptography*" (Chapter 9) *Dennis Hofheinz and Eike Kiltz* shed light on the quest for cryptographic methods that keep on working for significantly increased data set sizes. The security guarantees of currently used RSA encryption technology, for example, degrade linearly in the number of users and ciphertexts. This limits their applicability to smaller data sets or requires significantly larger keylengths which in turn slows down and complicates the whole process (in particular if the keylengths are to grow dynamically).

The authors discuss a number of settings in which it is possible to provide alternative scalable cryptographic building blocks. In particular, they survey SPP work on the construction of scalable public-key encryption schemes (a central cryptographic building block that helps secure communication), but also briefly mention other settings such as "reconfigurable cryptography".

– In "*Distributed Data Streams*" (Chapter 10) *Jannik Castenow, Björn Feldkord, Jonas Hanselle, Till Knollmann, Manuel Malatyali, and Friedhelm Meyer auf der Heide* consider a big data scenario where a server is wirelessly connected to a huge number of sensor nodes that continuously measure data. At each time step the server needs to calculate a function defined over the current measurements of the sensors.

Due to the sensors' restricted compute and battery power, the communication between server and sensors has to be optimized, for example by minimizing the total number of messages using clever randomized protocols. The authors review SPP results for three concrete functions: Top-k-Value Monitoring, Top-k-Position Monitoring, and (Approximate) Count Distinct Monitoring.

– In "*Energy-Efficient Scheduling*" (Chapter 11) *Susanne Albers* reports on algorithmic techniques for energy reduction in processing environments where machine parameters can be changed at runtime. In the first part she addresses dynamic speed scaling: Given a typically superlinear increase of energy consumption with rising processor speed, the goal is to cleverly use the whole speed range so as to minimize energy consumption while still providing the desired service. The author in particular reports on SPP results for multi-processor platforms with heterogeneous CPUs. She also examines power-down mechanisms (i.e., idle devices can be transitioned into low-power standby and sleep states) in multi-processor environments, where the active and idle periods of the components have to be carefully coordinated in order to maintain a guaranteed performance level.
– In "*The GENO Software Stack*" (Chapter 12)) *Joachim Giesen, Lars Kuehne, and Sören Laue* present a domain specific language for large-scale mathematical optimization called GENO (for generic optimization).

The GENO software generates a solver from a specification of an optimization problem, i.e. objective function and constraints are specified in a formal language. The problem specification is then translated into a general normal form, which in turn is passed on to a general purpose solver with optimized support for various hardware platforms including GPUs by carefully integrated BLAS (Basic Linear Algebra Subroutines) calls. The authors show that by putting all the components together the generated solvers are competitive with problem-specific hand-written solvers and orders of magnitude faster than competing approaches that offer comparable ease-of-use.

– In "*Algorithms for Big Data Problems in de Novo Genome Assembly*" (Chapter 13) *Anand Srivastav, Axel Wedemeyer, Christian Schielke, and Jan Schiemann* address some algorithmic problems related to genome assembly.

Concretely speaking they first present an algorithm which significantly reduces the input data size without practically impacting the assembly quality. They then turn to the important subproblem of efficiently counting k-mers for which they provide an external-memory solution. Further reconstruction steps boil down to the longest path problem and the Eulerian tour problem. In order to tackle those they present a linear time (per edge) streaming algorithm for heuristically constructing long paths in undirected graphs, and a streaming algorithm for the Euler tour problem with optimal one-pass complexity.

– In "*Scalable Text Index Construction*" (Chapter 14) *Timo Bingmann, Patrick Dinklage, Johannes Fischer, Florian Kurpicz, Enno Ohlebusch, and Peter Sanders* discuss the current state of the art in large-scale computation of text-indices.

When treating distributed, external, and shared memory approaches for different text indices and their applications the authors point out common techniques that are used in different models of computation or in the computation of different text indices. While most of the discussed work solely focuses on the *construction* of the text indices, they also show approaches to actually *answer* queries on text indices in distributed memory. In addition they discuss real-world applications in bioinformatics and text compression and future challenges.

We would like to thank all authors who submitted their work, the referees for their helpful comments, as well as the DFG for accepting and sponsoring the priority program SPP 1736 on Algorithms for Big Data. We hope that this volume will prove useful for further research in big data algorithms.

May 2022

Hannah Bast
Claudius Korzen
Ulrich Meyer
Manuel Penschuck

Organization

Reviewers

Susanne Albers	Technische Universität München, Germany
Eugenio Angriman	Humboldt-Universität zu Berlin, Germany
Hannah Bast	Albert-Ludwigs-Universität Freiburg, Germany
Timo Bingmann	Karlsruhe Institute of Technology, Germany
Ulrik Brandes	ETH Zurich, Switzerland
Holger Dell	Goethe University Frankfurt, Germany
Michael Hamann	Karlsruhe Institute of Technology, Germany
Till Knollmann	Paderborn University, Germany
Oliver Koch	University of Münster, Germany
Nils Kriege	University of Vienna, Austria
Florian Kurpicz	Technische Universität Dortmund, Germany
Sebastian Lamm	Karlsruhe Institute of Technology, Germany
Sören Laue	Friedrich-Schiller-Universität Jena, Germany
Ulrich Meyer	Goethe University Frankfurt, Germany
Friedhelm Meyer auf der Heide	Paderborn University, Germany
Henning Meyerhenke	Humboldt-Universität zu Berlin, Germany
Matthias Mnich	Hamburg University of Technology, Germany
Manuel Penschuck	Goethe University Frankfurt, Germany
Knut Reinert	Freie Universität Berlin, Germany
Peter Sanders	Karlsruhe Institute of Technology, Germany
Christian Schindelhauer	Albert-Ludwigs-Universität Freiburg, Germany
Christoph Scholl	Albert-Ludwigs-Universität Freiburg, Germany
Christian Schulz	Heidelberg University, Germany
Anand Srivastav	Christian-Albrechts-Universität Kiel, Germany
Alexander van der Grinten	Humboldt-Universität zu Berlin, Germany
Dorothea Wagner	Karlsruhe Institute of Technology, Germany
Axel Wedemeyer	Christian-Albrechts-Universität Kiel, Germany
Katharina Zweig	TU Kaiserslautern, Germany

Contents

Algorithms for Large and Complex Networks

Algorithms for Large-Scale Network Analysis and the NetworKit Toolkit . . . 3
Eugenio Angriman, Alexander van der Grinten, Michael Hamann, Henning Meyerhenke, and Manuel Penschuck

Generating Synthetic Graph Data from Random Network Models 21
Ulrich Meyer and Manuel Penschuck

Increasing the Sampling Efficiency for the Link Assessment Problem 39
André Chinazzo, Christian De Schryver, Katharina Zweig, and Norbert Wehn

A Custom Hardware Architecture for the Link Assessment Problem 57
André Chinazzo, Christian De Schryver, Katharina Zweig, and Norbert Wehn

Graph-Based Methods for Rational Drug Design. 76
Andre Droschinsky, Lina Humbeck, Oliver Koch, Nils M. Kriege, Petra Mutzel, and Till Schäfer

Recent Advances in Practical Data Reduction 97
Faisal N. Abu-Khzam, Sebastian Lamm, Matthias Mnich, Alexander Noe, Christian Schulz, and Darren Strash

Skeleton-Based Clustering by Quasi-Threshold Editing 134
Ulrik Brandes, Michael Hamann, Luise Häuser, and Dorothea Wagner

The Space Complexity of Undirected Graph Exploration. 152
Yann Disser and Max Klimm

Algorithms for Big Data and Their Applications

Scalable Cryptography 169
Dennis Hofheinz and Eike Kiltz

Distributed Data Streams 179
Jannik Castenow, Björn Feldkord, Jonas Hanselle, Till Knollmann, Manuel Malatyali, and Friedhelm Meyer auf der Heide

Energy-Efficient Scheduling 196
Susanne Albers

The GENO Software Stack. 213
Joachim Giesen, Lars Kuehne, and Sören Laue

Algorithms for Big Data Problems in de Novo Genome Assembly 229
Anand Srivastav, Axel Wedemeyer, Christian Schielke, and Jan Schiemann

Scalable Text Index Construction 252
Timo Bingmann, Patrick Dinklage, Johannes Fischer, Florian Kurpicz, Enno Ohlebusch, and Peter Sanders

Author Index 285

Algorithms for Large and Complex Networks

Algorithms for Large-Scale Network Analysis and the NetworKit Toolkit

Eugenio Angriman[1](✉), Alexander van der Grinten[1], Michael Hamann[2], Henning Meyerhenke[1], and Manuel Penschuck[3]

[1] Humboldt-Universität zu Berlin, Berlin, Germany
{angrimae,avdgrinten,meyerhenke}@hu-berlin.de
[2] Karlsruhe Institute of Technology, Karlsruhe, Germany
michael.hamann@kit.edu
[3] Goethe University Frankfurt, Frankfurt am Main, Germany
mpenschuck@ae.cs.uni-frankfurt.de

Abstract. The abundance of massive network data in a plethora of applications makes scalable analysis algorithms and software tools necessary to generate knowledge from such data in reasonable time. Addressing scalability as well as other requirements such as good usability and a rich feature set, the open-source software NETWORKIT has established itself as a popular tool for large-scale network analysis. This chapter provides a brief overview of the contributions to NETWORKIT made by the SPP 1736. Algorithmic contributions in the areas of centrality computations, community detection, and sparsification are in the focus, but we also mention several other aspects – such as current software engineering principles of the project and ways to visualize network data within a NETWORKIT-based workflow.

Keywords: Network analysis · Algorithms · Software package

1 Introduction

Network phenomena surround us, be they social contact networks, organizational structures, or infrastructure networks such as the energy grid, roads or the (physical) internet. Purely virtual networks such as the world wide web, online social networks, or co-authorship networks can become particularly large and play an ever increasing role in our daily lives [8,62]. Traditional data analysis has been and is very successful in discovering knowledge from non-network (e.g., geometric or relational) data [50]. Yet, networks and their analysis are about "dependence, both between and within variables" [26]. To uncover implicit dependencies hidden in the data, it thus requires appropriate algorithmic techniques (some of which are also covered in Leskovec et al.'s textbook on mining massive datasets [50]).

Massive networks, often with billions of vertices and edges, pose challenges to many established analysis concepts and algorithms due to their prohibitive computational costs. This leads to the ongoing development of efficient and scalable algorithms. The open-source software package NETWORKIT[1] [75 SPP] aims to combine a broad

[1] https://networkit.github.io/.

range of such algorithms for the analysis of large networks and to make them accessible via consistent, easy to use, and well-documented frontends. For instance, it offers a feature-rich Python API which integrates into the large Python ecosystem for data analysis. Under the hood, the heavy lifting is carried out by performance-oriented algorithms that are implemented in C++ and often use multicore parallelism. The package is also well suited to develop and evaluate novel algorithmic approaches. As such, NetworKit received numerous unique scalable algorithms and implementations in recent years, particularly designed to handle large inputs.

In this chapter, we present a high-level overview of NetworKit (Sect. 2) and portray algorithmic research results derived with and for NetworKit – mostly those obtained by projects of SPP 1736. We cover four main topics: centrality algorithms (Sect. 3), community detection (Sect. 4), graph sparsification (Sect. 5) as well as graph drawing and network visualization (Sect. 6). While these have been focus areas of NetworKit development during the lifetime of SPP 1736, the package has been used in various other application contexts such as quantum chemistry [56 SPP] and digital humanities [47].

2 *NetworKit*—An Overview

NetworKit is in development since 2013. The architecture of the current codebase was released in 2014. At the time of writing, NetworKit has a regular release cycle with two new major releases per year. Staudt et al. [75 SPP] describe the package's state at the end of 2015. In this section, we consequently focus on the many additions of new functionality as well as improvements to the code quality that have been realized in the meantime. This concerns new performance-oriented graph algorithms, engineering to speed up existing algorithms, more software engineering guidelines and best practices, as well as the modernization and extension of NetworKit's integration with other tools within a rich ecosystem (as detailed in Sect. 2.2).

2.1 Design Considerations

NetworKit consists of several Python modules wrapping an independently usable core library that is written in C++. Both parts are connected using Cython and are tightly integrated to offer consistent interfaces for most features. The package is organized into multiple modules, each focusing on one (class of) network analytic problem(s). Important modules deal with network centrality (`centrality`), community detection (`community` and `scd`) as well as graph generation and perturbation (`generators` and `randomization`). Some novel algorithms in the `centrality`, `community`, and `sparsification` modules that were developed within SPP 1736 are described in more detail in Sects. 3 to 5. Other important modules that are not covered here include modules for graph algorithms in the language of linear algebra (`algebraic`, following the philosophy of GraphBLAS [45 SPP]), decomposition of graphs into components (`components`), distance computations (`distance`), reading and writing graphs (`io`), link prediction (`linkprediction`), graph coarsening (`coarsening`), and more.

As a graph data structure, NetworKit uses an adjacency array using dynamic arrays (`std::vector`) to store vertices and their neighborhoods. It also supports edge weights and edge IDs. This data structure was chosen over static ones such as CSR matrices since it allows for efficient dynamic updates. The design is complemented by several non-trivial algorithms that can efficiently update their results if the underlying graph changes (i.e., after adding and/or deleting edges).

Many of NetworKit's algorithms use OpenMP for shared-memory parallelism. In fact, several algorithms in NetworKit exhibit best-in-class parallel performance [36 SPP]. Based on an empirical comparison [46 SPP] between NetworKit and several distributed frameworks for data and network analysis, NetworKit's speed advantage usually remains true in comparison to distributed systems with eight-fold resource consumption. Ref. [46 SPP] finds that a shared-memory machine is sufficient to solve many network analytic problems on real-world instances and concludes that shared-memory parallelism should be preferred to distributed graph algorithms as long as the input graph fits into main memory.

2.2 Ecosystem

In recent years, NetworKit matured into an actively maintained open-source project with more than 140 000 lines of code and a steadily growing number of users and contributors. By now, the software package exceeds a critical size that warrants efforts beyond the development of new algorithmic features.

To ease contributions and uphold the code quality, NetworKit offers detailed guidelines and implements a thorough review process. We also make heavy use of unit-tests, static code analysis and automated code-formatting as part of our continuous integration pipeline, which targets the three major operating systems. As many new tests improve the coding standards, we continuously modernize the codebase. Still, backwards compatibility is a major concern and manifests itself, for instance, in long-term compiler support and in as few changes breaking the API as possible (preceded by a deprecation period of at least one major version release).

Users benefit from a welcoming community, ever-improving documentation, interactive examples showcasing most features, a regular release schedule, and growing support for package managers (currently brew, Conda, pip, and Spack). NetworKit naturally interacts with external projects such as Gephi (see Sect. 6), SimExPal [4 SPP], and networkx as well as graph repositories and formats including konect, snap, and metis; recent changes make it now even possible to develop standalone NetworKit Python modules.

Graph data can not only be imported but also be synthesized. To this end, NetworKit offers versatile graph generators in the modules `generators` and `randomization`. Among others, they are designed to generate and supplement datasets for applications ranging from rapid prototyping to experimental campaigns. Here, we only mention the supported network *models* since Chap. 2 surveys novel generation *algorithms* obtained during SPP 1736. We include here citations to models or generators developed for/with NetworKit.

– Focus on community structure: Clustered-Random-Graph, LFR, PubWeb, R-MAT, Stochastic Block Model, Watts-Strogatz

– Prescribed degrees: Havel-Hakimi, Chung-Lu, Curveball and Global-Curveball [28 SPP], Edge-Switching
– Preferential attachment processes: Barabási-Albert, Dorogovtsev-Mendes
– Geometrically embedded: Hyperbolic Random Graph [52 SPP, 53 SPP, 54 SPP, 55 SPP, 19 SPP], Geometric Inhomogenous Random Graph [19 SPP], Mocnik [59, 60]
– Basic models: $G(n, p)$, Lattice.

Several generators have dynamic variants simulating the evolution of graphs over time.

3 Centrality Algorithms

One of the most popular concepts used for the analysis of a graph $G = (V, E)$ is *centrality*. Centrality measures assign a score to each vertex[2] (or group of vertices) based on its structural position or importance; these scores allow a corresponding vertex ranking [21]. As an example, the well-known PageRank [27] is a centrality measure originally devised for web page (and eventually search query) ranking. It is important to match the underlying research question with the appropriate centrality measure [77 SPP] and no single measure is universal. Thus, dozens of measures have been proposed in the literature [21].

As described in more detail below, the centrality research within NETWORKIT revolves not only around faster algorithms for computing individual scores and top-k rankings. Another emphasis is placed on two families of centrality-driven optimization problems (centrality improvement and group centrality) and how to scale approximation algorithms or heuristics for their solution to much larger input sizes. For a broader overview, also with a scalability focus, the reader is referred to Ref. [35 SPP].

It should also be noted that fast centrality algorithms can be useful in different (but related) contexts as well; e.g., scores of several centrality measures are used as shortcuts for more expensive influence maximization calculations [70 SPP]. Also, using score distributions for graph fingerprinting (putting graphs into classes where all members have similar distributions) is a conceivable use case with the need for numerous measures that can be computed quickly.

3.1 Individual Centrality Scores

We first discuss centrality measures for individual vertices, i.e., measures that assign a centrality score to each $v \in V$. During SPP 1736, our focus has been on two classes of centrality measures: centralities that make use of shortest path computations (i.e., (harmonic) closeness and betweenness) and algebraic centrality measures that consider more than just shortest paths (like Katz centrality and electrical closeness). Figure 1 depicts the distribution of these centralities for a single network, including the ED Walk centrality that we propose in Ref. [3 SPP].

[2] Edge centrality measures are ignored here in the interest of space.

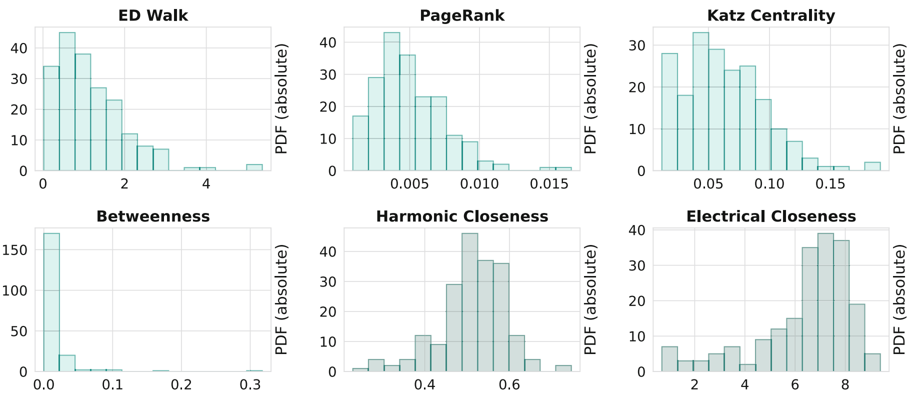


Fig. 1. Histograms of the distribution of vertex centrality measures of the JAZZ network, which models the collaboration of Jazz musicians [34].

Betweenness. Betweenness centrality is based on the fraction of shortest paths a vertex participates in. NETWORKIT implements the well-known Brandes algorithm [23] for exact betweenness and several algorithms for betweenness approximation. For static graphs, it has an implementation of the KADABRA algorithm [22]; additionally, NETWORKIT can approximate betweenness in dynamic graphs [15 SPP]. Both of these algorithms employ a sampling technique that was originally introduced by Riondato and Kornaropoulos [66]. More precisely, the algorithms sample pairs (s,t) of source and target vertices uniformly at random. For each (s,t), a single shortest path is sampled uniformly at random out of all shortest s-t paths. The algorithms count the number of occurrences of vertices on these paths; they differ in their stopping conditions. The multi-threaded implementation of the static KADABRA algorithm additionally exploits a fast data structure for asynchronous synchronization barriers [36 SPP]. To the best of our knowledge, NETWORKIT's implementation of KADABRA is the fastest betweenness approximation code that is available for multi-threaded machines.

In Ref. [39 SPP], this algorithm was extended to work with replicated graphs in distributed memory. The resulting algorithm obtains good parallel speedups and performs well even on multi-socket shared memory machines due to the fact that it can avoid NUMA bottlenecks. Since distributed memory algorithms are outside the scope of NETWORKIT, this implementation is available externally.

Closeness. Closeness centrality also uses the notion of shortest paths: it quantifies the importance of a vertex $v \in V$ depending on how close v is to all the other vertices of the graph [11]. It is defined as $c(v) := (n-1)/(\sum_{w \in V} d(v,w))$ and computing it for a single vertex requires to run a single-source shortest path (SSSP) algorithm. The textbook algorithm to identify the top-k vertices with highest closeness centrality computes $c(v)$ for each vertex of the graph by running n SSSPs, which is impractical for large-scale networks. NETWORKIT improves on this by providing an algorithm which finds the top-k vertices with highest closeness centrality along with their exact value of $c(\cdot)$ [12 SPP]. Even though the worst-case running time of the algorithm is also

$\Omega(|V||E|)$, experimental evaluation on real-world data shows that, for small values of k, the algorithm is in practice much more efficient than the textbook algorithm and other state-of-the-art strategies.

NetworKit additionally implements a batch-dynamic version of this algorithm [18 SPP,2 SPP], which also addresses harmonic centrality [21,67] – an alternative definition of closeness centrality introducing support for disconnected graphs. Experiments on both real-world and synthetic instances demonstrate that, for moderately large batches of edge updates, the dynamic algorithm is up to four orders of magnitude faster than a static recomputation from scratch.

Electrical Closeness. Electrical resistance is a distance function on graphs that is constructed by interpreting the graph as a network of electrical resistors and by measuring the effective resistance between vertices in this network. If the usual distance function (based on shortest-path distances) in the definition of closeness is replaced by effective resistance, one obtains the definition of *electrical* closeness. This centrality measure has been gaining attention due to the fact that it considers paths of any length. NetworKit has an efficient approximation algorithm to compute electrical closeness [6 SPP]. This algorithm exploits a well-known connection between electrical networks and uniform spanning trees to approximate electrical closeness faster than previous numerical algorithms (including the numerical algorithm from Ref. [17 SPP]) and can handle graphs with hundreds of millions of edges.

As part of our work on electrical closeness, NetworKit gained support for various numerical algorithms. These are typically either used as subprocedures of our algorithms or for performance and/or quality comparisons; however, they can also be called as standalone numerical solvers. Experiments with an (in terms of theoretical analysis) fast Laplacian solver revealed severe limitations in practice [43 SPP] – which is why it was discarded. Instead, we included a fast implementation [17 SPP] of the lean algebraic multigrid algorithm (LAMG) [51], which is particularly well-suited to solve series of Laplacian linear systems with identical system matrices.

Katz Centrality. NetworKit also implements an approximation algorithm for Katz centrality that can handle graphs with billions of edges within a few minutes [38 SPP]. The algorithm utilizes lower and upper bounds on the centrality score of each vertex and improves these bounds until the Katz centrality ranking is computed with sufficient precision. In comparison to earlier combinatorial algorithms for Katz centrality, our algorithm is the first to obtain a provable approximation bound and/or the correctness of the ranking. It is also at least 50% faster than numerical methods. NetworKit provides a parallel implementation of this algorithm that can also handle dynamic graphs. In Ref. [38 SPP], we additionally provide a GPU-based implementation which is not part of NetworKit.

3.2 Improving One's Own Centrality

One possible way to improve one's ranking position in a web search is to attract links from influential web pages. For some time, this led to so-called link farming [49] for search engine optimization. More generally, beyond web search, one wants to increase the centrality of a vertex by adding a specified number of new edges incident to it. Crescenzi et al. [30] addressed this problem for closeness centrality. As a follow-up to that work, Ref. [13 SPP] considered two betweenness centrality improvement problems: maximizing the betweenness *score* of a given vertex (MBI) and maximizing the *ranking position* of a given vertex (MRI). The paper proves that both problems are hard to approximate. Unless $\mathscr{P} = \mathscr{NP}$, MBI cannot be approximated within a factor greater than $1 - \frac{1}{2e}$ and for MRI there is no α-approximation algorithm for any constant $\alpha \leq 1$. The paper also proposes a simple greedy algorithm for MBI that performs well in practice and provides a $(1 - 1/e)$-approximation. This way, MBI can be approximated for (most) networks with up to 10^5 edges in a matter of seconds or a few minutes. The greedy algorithm's implementation builds, among others, upon a dynamic algorithm for betweenness centrality [16 SPP] that can update the betweenness scores of all vertices much faster after small graph changes (such as the insertion of one or few edges).

3.3 Group Centrality Optimization

Group centralities are network-analytic measures that quantify the importance of vertex groups [31]. In contrast to centrality measures that apply to individual vertices, the goal of these measures is to determine how well the entire group jointly "covers" the graph; i.e., the group centrality score is *not* determined by the scores of individual vertices.

NETWORKIT includes various group centrality algorithms to approximate sets of vertices that maximize the group centrality score. Most of the algorithms are based on submodular optimization. For example, NETWORKIT implements a greedy algorithm to approximate group degree and the group betweenness maximization algorithm by Mahmoody et al. [57]. New algorithms developed as part of SPP 1736 are the GED-Walk approximation algorithms from Ref. [3 SPP] and various group closeness algorithms; these algorithms are described below. A very recent addition to NETWORKIT is an approximation algorithm for group forest closeness centrality; for details we refer to Ref. [37 SPP].

Group Closeness. Group closeness measures the importance of a *group* of vertices $S \subset V$ as the reciprocal of the sum of the distances from S to the vertices in $V \setminus S$, where the distance from S to a vertex $v \in V$ is defined by the minimum $d(S,v) := \min_{u \in S} d(u,v)$. Finding the group $S^\star$ with highest group closeness is known to be an $\mathscr{NP}$-hard optimization problem [29, 1 SPP]. Thus, in practice, the problem is addressed on large-scale networks either with heuristics or approximation algorithms. NETWORKIT provides a greedy heuristic [14 SPP] that computes a set of vertices with high group centrality. On small enough instances where it is feasible to compute the optimum, it has been shown that the algorithm yields solutions with nearly optimal quality.

An alternative heuristic, which allows to trade quality for speed, is based on local search. NETWORKIT implements a family of local search heuristics for group closeness maximization that achieve different trade-offs between quality and running time [5 SPP]. In general, they are one to three orders of magnitude faster than the greedy algorithm. At the same time, our algorithms retain 80%—and, in numerous cases, even more than 99%—of the greedy algorithm's solution quality. NETWORKIT also includes the first approximation algorithm for group closeness maximization [1 SPP] (for undirected graphs) which yields solutions with higher quality than the greedy algorithm at the cost of additional running time.

A major limitation of group closeness is that it can only handle (strongly) connected graphs – the distance between unreachable vertices is either undefined or infinite, and an infinite denominator results in group closeness score of zero. Another group centrality measure that also handles disconnected graphs is group harmonic centrality, which is defined as $GH(S) := \sum_{u \in V \setminus S} d(S,u)^{-1}$. Maximizing GH has been shown to be an $\mathcal{NP}$-hard problem [1 SPP] as well and two approximation algorithms for group harmonic maximization have been introduced in Ref. [1 SPP]; both of them are available in NETWORKIT.

GED-Walk. GED-Walk (GED = group exponentially decaying) is an algebraic group centrality measure that was introduced in Ref. [3 SPP]. Similarly to Katz centrality (which only applies to individual vertices), GED-Walk counts the number of *walks* (and not paths) in the graph. Unlike Katz centrality, it counts walks that *cross* the group of vertices (instead of counting walks that *start* (or end) at certain vertices). Computing GED scores can essentially be done via sparse matrix-vector multiplication; hence, the measure can be computed faster than centrality measures that involve the computation of shortest paths. In Ref. [3 SPP], we propose a greedy algorithm that computes a group with approximately maximal GED-Walk centrality. The algorithmic approach is based on techniques derived from our Katz algorithm [38 SPP] and iteratively refines bounds on the group centrality score. In experiments, GED-Walk maximization turns out to be at least one order of magnitude faster than the corresponding greedy algorithms for group betweenness and group closeness. When applied within semi-supervised vertex classification, GED-Walk improves the accuracy compared to various existing measures.

4 Community Detection

Community detection aims to detect subgraphs that are internally densely and externally sparsely connected. From this fuzzy idea, many formalizations and algorithms have been developed [32]. A division of the graph into disjoint communities is the most frequently studied setting. The most popular quality measure for this setting is modularity [63]. As it is $\mathcal{NP}$-hard to find the (clustering with) optimal modularity score [24], heuristics are used in practice. A very popular one is the Louvain algorithm [20]. While it is already quite fast, it is purely sequential in its original formulation and thus does not exploit the many cores available in modern processors. Already the earliest work in NETWORKIT includes the development of a parallel variant of the Louvain algorithm named PLM [72]. This first work also includes a fast parallel label propagation algorithm named PLP and an ensemble algorithm that combines several runs of PLP with a final step where PLM is used. Later improvements to PLM, including the parallelization of additional steps, made PLM so fast that it outperformed the ensemble approach both in terms of speed and quality [74 SPP]. Further, a refinement round similar to Ref. [68]

has been introduced that further increases the quality at the expense of a slightly longer running time. PLM was later used in a case study on correspondences between clusterings [33 SPP]. With such correspondences one can reveal how one clustering differs from another one, e.g., when computed with different algorithms or after minor graph changes.

If only a community around a specific vertex or a set of vertices (so-called seed vertices) is desired, we do not need to detect communities that cover the whole graph. Many such algorithms greedily add new vertices until a local minimum of a certain quality function is reached. A first study on such local community detection algorithms [71 SPP] based on NetworKit has shown that they are quite slow and imprecise in comparison to PLM. A more recent study [41 SPP] shows that many local community algorithms detect a community in which the seed is not strongly connected. Only algorithms that employ further guidance, e.g., using edge scores based on triangles, are able to correctly identify a community the seed vertex is embedded in. The study further shows that the results of all local community detection algorithms can be improved by starting with the largest clique in the subgraph induced by the neighbors of the seed vertex. For this, the possibility to combine two local community detection algorithms has been added to NetworKit – a first one that detects the clique and then a second one that expands this clique into a community [41 SPP]. This allows changing both the seeding strategy and the latter expansion step.

For the experimental evaluation of community detection algorithms, suitable input instances are required [7]. Ideally, instances from applications of community detection with known ground truth communities should be used for this. However, they are frequently either quite small, unavailable due to privacy concerns or commercial interests, or the available ground truth data cannot be recovered from the graph's structure [32,65]. For this reason, synthetically generated benchmark graphs with generated ground truth communities are frequently used. The most popular one is the LFR benchmark graph generator [48], of which NetworKit also provides an implementation for the case of unweighted, undirected graphs with disjoint communities [73 SPP] (see also Chap. 2). Due to a partial parallelization and more efficient data structures, experiments show a speedup compared to the original implementation of 18 to 70 using 16 cores [73 SPP]. When the similarity between a detected and a (possible) ground truth community is low, it is often not clear if such a similarity could also be achieved by chance. Therefore, Hamann et al. [41 SPP] also introduced a simple baseline algorithm using a BFS that stops when the same number of vertices as contained in the ground truth community have been visited and returns them as community. Together with additional methods for the evaluation of the found communities, NetworKit thus provides a comprehensive framework for the development, evaluation, and application of local community detection algorithms.

Nastos and Gao [61] suggest quasi-threshold graphs, i.e., graphs that do not contain a path or cycle of four vertices as vertex-induced subgraph, as a model for communities in social networks. As a given graph is usually not a quasi-threshold graph, they suggest to insert and delete as few edges as possible to transform a graph into a quasi-threshold graph. The connected components are then considered as communities. The first scalable heuristic for this problem [25 SPP] has been implemented in NetworKit, for details we refer to Chap. 7.

5 Graph Sparsification

Centrality measures suggest that certain vertices or edges are more important than others. In graph *sparsification*, the idea is to exploit this fact to obtain a subset of the vertices and/or edges that preserve key properties of the graph, i.e., to select vertices and edges that are important for these properties. Properties of the graph can be preserved either directly or in a scaled version. For example, the degree distribution cannot be exactly preserved when we remove edges, but we can preserve the general shape of the degree distribution. Graph sparsification can provide insights into the structure of a graph, as it provides insights on how much redundancy there is and which edges are important for certain properties. An application of these insights is speeding up other network analysis tasks or making them possible in the first place by reducing the graph's size such that the running time and memory requirements are reduced [69]. Further, some of these sparsification techniques can also remove noise from the graph such that, e.g., more informative drawings can be generated [64 SPP]. In NetworKit, we provide a set of edge sparsification algorithms [40 SPP]. Given a graph $G = (V, E)$, they identify subsets $E' \subset E$ of the edges such that $G' = (V, E')$ preserves certain properties of G. We currently do not consider vertex sparsification, i.e., filtering vertices while maintaining properties of the graph – since in many network analysis tasks (like vertex centralities or community detection), we are interested in a result for every vertex. If some vertices were no longer part of the graph, we would need to extrapolate their results, requiring an additional post-processing step for every network analysis task.

With its diverse set of network analysis algorithms, NetworKit provides the ideal testbed for sparsification algorithms. A study [40 SPP] compares a set of six existing and one novel sparsification algorithm as well as five novel variants of the existing algorithms using NetworKit. The study shows that these sparsification algorithms can be classified into three groups: those that primarily preserve edges within densely connected areas, those that primarily preserve connectivity between different areas, and those that are almost or completely random. The algorithms in the first group strengthen the formation of communities and either keep or increase the average local clustering coefficient as already suggested by previous work [69, 64 SPP]. The novel local degree technique, on the other hand, keeps distances in the graph and thus the diameter small, see Fig. 2 for an example. As the results show, it is also good at preserving vertex centralities. Completely random filtering also works surprisingly well at preserving a wide range of network properties. The study shows that all methods perform better for most measures if, instead of directly filtering edges globally, a vertex of degree d keeps its top d^e neighbors for some exponent $e < 1$. This local filtering step has been proposed before [69] for a single sparsification algorithm and the study suggests to apply it to all considered algorithms. In particular, this preserves connectivity of the graph quite well and in general leads to a more even distribution of the preserved edges.

All of these sparsification algorithms can be decomposed into two steps: A first step that assigns each edge a score and a second step that only keeps a certain fraction of the highest-rated edges. Even the local filtering step can be implemented as a

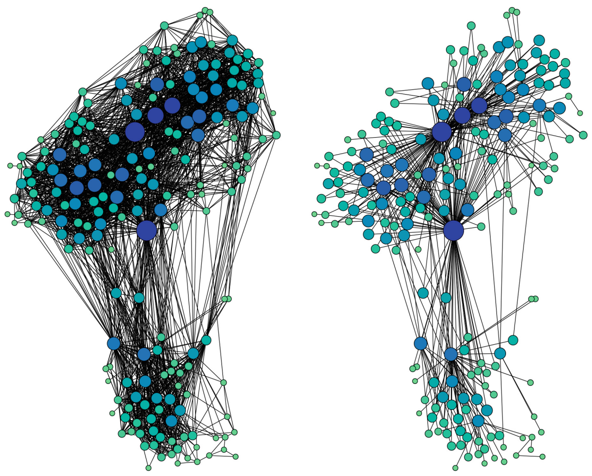


Fig. 2. Drawing using GEPHI [9] of the JAZZ network [34] (left) and a sparsified version containing 15% of the edges (right) using the novel local degree algorithm. Vertex size and color is proportional to degree. (Color figure online)

transformation of edge scores. This makes it possible to easily combine existing and new algorithms. Further, the resulting scores can be considered as edge centrality measures that permit a ranking of the edges. With the help of visualization software like GEPHI [9] (Sect. 6), the scores can also be visualized or used for interactive filtering of edges.

6 Graph Drawing and Network Data Visualization

In exploratory network analysis, one needs to evaluate several properties of the network, which requires writing code to run algorithms and plot their results. To speed up this process, NETWORKIT provides a dedicated `profiling` module that allows non-expert users to run several network analysis algorithms as a single program and visualize their results in a graphical report that can be rendered in a Jupyter Notebook or exported as an HTML or a LaTeX document. As thoroughly explained in Ref. [75 SPP], first the report lists global properties of the networks such as the size and the density. Then it provides an overview of the distribution of several centrality networks as histograms (as shown in Fig. 1, Sect. 3), followed by a more detailed statistical analysis. Finally, the report includes a matrix with the Spearman correlation coefficients between the rankings of the vertices according to the considered centrality measures; an example for the JAZZ network is shown in Fig. 3.

When dealing with large graphs, statistical overviews as the ones mentioned are indispensable, since the well-known vertex-edge diagrams do not even scale to graphs of medium size (without further adjustments). For small graphs, however, visualizations such as those diagrams can be very valuable. In general, the goal of graph visualization [10] is to represent graphs in a form that is meaningful to the human eye. Popular

+1.000	ED Walk					
+0.955	+1.000	PageRank				
+1.000	+0.956	+1.000	Katz Centrality			
+0.645	+0.736	+0.645	+1.000	Betweenness		
+0.967	+0.927	+0.967	+0.677	+1.000	Harmonic Closeness	
+0.982	+0.952	+0.982	+0.659	+0.954	+1.000	Electrical Closeness

Fig. 3. Spearman's correlation coefficients between vertex rankings obtained with different centrality measures for the JAZZ network. Darker [lighter] block shades indicate higher [smaller] correlation values.

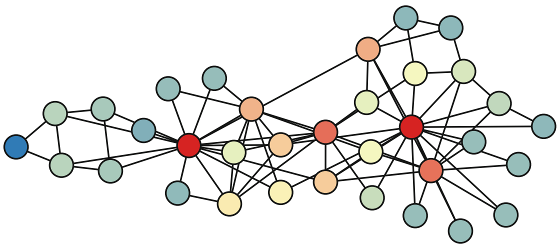


Fig. 4. Visualization example with GEPHI of the KARATE graph. Red vertices have the highest harmonic centrality, blue vertices the lowest. (Color figure online)

application areas for graph visualization are biology (e.g., genetic maps), chemistry (e.g., protein functions) [42], social network analysis [47], and many more. GEPHI [9] is a popular Java-based GUI application to explore and visualize graphs. NETWORKIT's `gephi` module [40 SPP] allows to use GEPHI to visualize graphs along with additional vertex- or edge attributes with minimal effort. Figure 4 shows the visualization in GEPHI of the popular KARATE graph obtained by the ForceAtlas2 graph drawing algorithm [44] and by coloring the vertices according to their harmonic centrality score.

Graph drawing actually precedes visualization in most cases. It is the process of computing meaningful coordinates for the graph vertices where such information is not supplied with the graph. NETWORKIT's approach for the most part is to use the graph drawing capability in GEPHI. It has, however, also an implementation of an algorithm for the maxent-stress objective function, following Ref. [58 SPP]. Here, the main intention is to solve an optimization problem that computes the three-dimensional structure of biomolecules, given distance information between some atom pairs. To this end, the original algorithm received several application-specific adaptations [76 SPP], e.g., to be able to handle noisy data appropriately. As a result, the new algorithm by far outperforms its competitors in terms of speed and flexibility, and often even produces a superior solution quality.

7 Conclusions

The main design goals of NETWORKIT (speed, rich feature set, usability, and integration into an ecosystem) prove to be very useful for users, but they can also be challenging for the developers. One lesson learned to keep an academic open-source project of this size manageable and alive, is to combine best practices in both software engineering and algorithm engineering [4 SPP]. For example, a proper modularization allows easier reuse and combination of components, leading to a better extensibility and maintainability. These keywords are well-known in software engineering, but they also have their effect in algorithm design and implementation – in particular a simplified exploration of the design space in experimental algorithmics. NETWORKIT has already proved to be very useful in this respect for developers.

We have seen that approximation and parallelism can bring us a long way regarding scalability. They are the obvious, but certainly not the only choices for acceleration: exploiting the structure of the data, e.g., small vs. large diameter [12 SPP], can yield significant speedups on real-world data—even in the context of exact computations and potentially on top of parallelism.

NETWORKIT is constantly improved and extended – according to the resources available to the project. There are numerous ideas for larger updates from various angles – of which we mention only two representative ones: inherent support for attributes within (some of) the algorithms and further/improved integration with other tools. The latter is particularly geared towards a closer connection with machine learning, both on an algorithmic and a software tool level. Given the current interest in machine learning for data analysis, complete workflows within one seamless toolchain including NETWORKIT and tools such as SCIKIT-LEARN can be expected to be very attractive for users from many domains.

References

1 SPP. Angriman, E., Becker, R., D'Angelo, G., Gilbert, H., van der Grinten, A., Meyerhenke, H.: Group-harmonic and group-closeness maximization - approximation and engineering. In: ALENEX. SIAM (2021)

2 SPP. Angriman, E., Bisenius, P., Bergamini, E., Meyerhenke, H.: Computing top-*k* closeness centrality in fully-dynamic graphs. Taylor & Francis (2021). Currently in review

3 SPP. Angriman, E., van der Grinten, A., Bojchevski, A., Zügner, D., Günnemann, S., Meyerhenke, H.: Group centrality maximization for large-scale graphs. In: ALENEX, pp. 56–69. SIAM (2020). https://doi.org/10.1137/1.9781611976007.5

4 SPP. Angriman, E., et al.: Guidelines for experimental algorithmics: a case study in network analysis. Algorithms **12**(7), 127 (2019). https://doi.org/10.3390/a12070127

5 SPP. Angriman, E., van der Grinten, A., Meyerhenke, H.: Local search for group closeness maximization on big graphs. In: BigData, pp. 711–720. IEEE (2019). https://doi.org/10.1109/BigData47090.2019.9006206

6 SPP. Angriman, E., Predari, M., van der Grinten, A., Meyerhenke, H.: Approximation of the diagonal of a Laplacian's pseudoinverse for complex network analysis. In: ESA, pp. 6:1–6:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.ESA.2020.6

7. Bader, D.A., Meyerhenke, H., Sanders, P., Schulz, C., Kappes, A., Wagner, D.: Benchmarking for graph clustering and partitioning. In: Alhajj, R., Rokne, J. (eds.) Encyclopedia of Social Network Analysis and Mining, pp. 73–82. Springer, New York (2014). https://doi.org/10.1007/978-1-4614-6170-8_23
8. Barabási, A.L., Pósfai, M.: Network Science. Cambridge University Press, Cambridge (2016)
9. Bastian, M., Heymann, S., Jacomy, M.: Gephi: an open source software for exploring and manipulating networks. In: ICWSM. The AAAI Press (2009)
10. Battista, G.D., Eades, P., Tamassia, R., Tollis, I.G.: Graph Drawing: Algorithms for the Visualization of Graphs. Prentice-Hall, Hoboken (1999)
11. Bavelas, A.: A mathematical model for group structures. Hum. Organ. **7**(3), 16–30 (1948)

12 SPP. Bergamini, E., Borassi, M., Crescenzi, P., Marino, A., Meyerhenke, H.: Computing top-k closeness centrality faster in unweighted graphs. ACM Trans. Knowl. Discov. Data **13**(5), 53:1–53:40 (2019). https://doi.org/10.1145/3344719

13 SPP. Bergamini, E., Crescenzi, P., D'Angelo, G., Meyerhenke, H., Severini, L., Velaj, Y.: Improving the betweenness centrality of a node by adding links. ACM J. Exp. Algorithmics **23**, 1–32 (2018). https://doi.org/10.1145/3166071

14 SPP. Bergamini, E., Gonser, T., Meyerhenke, H.: Scaling up group closeness maximization. In: ALENEX, pp. 209–222. SIAM (2018). https://doi.org/10.1137/1.9781611975055.18

15 SPP. Bergamini, E., Meyerhenke, H.: Approximating betweenness centrality in fully dynamic networks. Internet Math. **12**(5), 281–314 (2016). https://doi.org/10.1080/15427951.2016.1177802

16 SPP. Bergamini, E., Meyerhenke, H., Ortmann, M., Slobbe, A.: Faster betweenness centrality updates in evolving networks. In: SEA, pp. 23:1–23:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.SEA.2017.23

17 SPP. Bergamini, E., Wegner, M., Lukarski, D., Meyerhenke, H.: Estimating current-flow closeness centrality with a multigrid Laplacian solver. In: CSC, pp. 1–12. SIAM (2016). https://doi.org/10.1137/1.9781611974690.ch1

18 SPP. Bisenius, P., Bergamini, E., Angriman, E., Meyerhenke, H.: Computing top-k closeness centrality in fully-dynamic graphs. In: ALENEX, pp. 21–35. SIAM (2018). https://doi.org/10.1137/1.9781611975055.3

19 SPP. Bläsius, T., Friedrich, T., Katzmann, M., Meyer, U., Penschuck, M., Weyand, C.: Efficiently generating geometric inhomogeneous and hyperbolic random graphs. In: ESA, pp. 21:1–21:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ESA.2019.21

20. Blondel, V.D., Guillaume, J., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. J. Stat. Mech: Theory Exp. **2008**(10), P10008 (2008). https://doi.org/10.1088/1742-5468/2008/10/p10008
21. Boldi, P., Vigna, S.: Axioms for centrality. Internet Math. **10**(3–4), 222–262 (2014). https://doi.org/10.1080/15427951.2013.865686
22. Borassi, M., Natale, E.: KADABRA is an adaptive algorithm for betweenness via random approximation. ACM J. Exp. Algorithmics **24**(1), 1.2:1–1.2:35 (2019). https://doi.org/10.1145/3284359
23. Brandes, U.: A faster algorithm for betweenness centrality. J. Math. Sociol. **25**(2), 163–177 (2001). https://doi.org/10.1080/0022250X.2001.9990249
24. Brandes, U., et al.: On modularity clustering. IEEE Trans. Knowl. Data Eng. **20**(2), 172–188 (2008). https://doi.org/10.1109/TKDE.2007.190689

25 SPP. Brandes, U., Hamann, M., Strasser, B., Wagner, D.: Fast quasi-threshold editing. In: Bansal, N., Finocchi, I. (eds.) ESA 2015. LNCS, vol. 9294, pp. 251–262. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48350-3_22

26. Brandes, U., Robins, G., McCranie, A., Wasserman, S.: What is network science? Netw. Sci. **1**(1), 1–15 (2013). https://doi.org/10.1017/nws.2013.2
27. Brin, S., Page, L.: Reprint of: the anatomy of a large-scale hypertextual web search engine. Comput. Netw. **56**(18), 3825–3833 (2012). https://doi.org/10.1016/j.comnet.2012.10.007
28 SPP. Carstens, C.J., Hamann, M., Meyer, U., Penschuck, M., Tran, H., Wagner, D.: Parallel and I/O-efficient randomisation of massive networks using global curveball trades. In: ESA, pp. 11:1–11:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.ESA.2018.11
29. Chen, C., Wang, W., Wang, X.: Efficient maximum closeness centrality group identification. In: Cheema, M.A., Zhang, W., Chang, L. (eds.) ADC 2016. LNCS, vol. 9877, pp. 43–55. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46922-5_4
30. Crescenzi, P., D'Angelo, G., Severini, L., Velaj, Y.: Greedily improving our own closeness centrality in a network. ACM Trans. Knowl. Discov. Data **11**(1), 9:1–9:32 (2016). https://doi.org/10.1145/2953882
31. Everett, M.G., Borgatti, S.P.: The centrality of groups and classes. J. Math. Sociol. **23**(3), 181–201 (1999). https://doi.org/10.1080/0022250X.1999.9990219
32. Fortunato, S., Hric, D.: Community detection in networks: a user guide. Phys. Rep. **659**, 1–44 (2016). https://doi.org/10.1016/j.physrep.2016.09.002
33 SPP. Glantz, R., Meyerhenke, H.: Many-to-many correspondences between partitions: introducing a cut-based approach. In: SDM, pp. 1–9. SIAM (2018). https://doi.org/10.1137/1.9781611975321.1
34. Gleiser, P.M., Danon, L.: Community structure in jazz. Adv. Complex Syst. **6**(4), 565–574 (2003). https://doi.org/10.1142/S0219525903001067
35 SPP. van der Grinten, A., Angriman, E., Meyerhenke, H.: Scaling up network centrality computations - a brief overview. IT - Inf. Technol. **62**(3–4), 189–204 (2020). https://doi.org/10.1515/itit-2019-0032
36 SPP. Grinten, A., Angriman, E., Meyerhenke, H.: Parallel adaptive sampling with almost no synchronization. In: Yahyapour, R. (ed.) Euro-Par 2019. LNCS, vol. 11725, pp. 434–447. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29400-7_31
37 SPP. van der Grinten, A., Angriman, E., Predari, M., Meyerhenke, H.: New approximation algorithms for forest closeness centrality - for individual vertices and vertex groups. In: SDM, pp. 136–144. SIAM (2021)
38 SPP. van der Grinten, A., Bergamini, E., Green, O., Bader, D.A., Meyerhenke, H.: Scalable Katz ranking computation in large static and dynamic graphs. In: ESA, pp. 42:1–42:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.ESA.2018.42
39 SPP. van der Grinten, A., Meyerhenke, H.: Scaling betweenness approximation to billions of edges by MPI-based adaptive sampling. In: IPDPS, pp. 527–535. IEEE (2020). https://doi.org/10.1109/IPDPS47924.2020.00061
40 SPP. Hamann, M., Lindner, G., Meyerhenke, H., Staudt, C.L., Wagner, D.: Structure-preserving sparsification methods for social networks. Soc. Netw. Anal. Min. **6**(1), 22:1–22:22 (2016). https://doi.org/10.1007/s13278-016-0332-2
41 SPP. Hamann, M., Röhrs, E., Wagner, D.: Local community detection based on small cliques. Algorithms **10**(3), 90 (2017). https://doi.org/10.3390/a10030090
42. Herman, I., Melançon, G., Marshall, M.S.: Graph visualization and navigation in information visualization: a survey. IEEE Trans. Vis. Comput. Graph. **6**(1), 24–43 (2000). https://doi.org/10.1109/2945.841119
43 SPP. Hoske, D., Lukarski, D., Meyerhenke, H., Wegner, M.: Engineering a combinatorial Laplacian solver: lessons learned. Algorithms **9**(4), 72 (2016). https://doi.org/10.3390/a9040072

44. Jacomy, M., Venturini, T., Heymann, S., Bastian, M.: ForceAtlas2, a continuous graph layout algorithm for handy network visualization designed for the Gephi software. PLoS ONE **9**(6), e98679 (2014)
45 SPP. Kepner, J., et al.: Mathematical foundations of the GraphBLAS. In: HPEC, pp. 1–9. IEEE (2016). https://doi.org/10.1109/HPEC.2016.7761646
46 SPP. Koch, J., Staudt, C.L., Vogel, M., Meyerhenke, H.: An empirical comparison of big graph frameworks in the context of network analysis. Soc. Netw. Anal. Min. **6**(1), 84:1–84:20 (2016). https://doi.org/10.1007/s13278-016-0394-1
47. Kreutel, J.: Augmenting network analysis with linked data for humanities research. In: Kremers, H. (ed.) Digital Cultural Heritage, pp. 1–14. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-15200-0_1
48. Lancichinetti, A., Fortunato, S., Radicchi, F.: Benchmark graphs for testing community detection algorithms. Phys. Rev. E **78**, 046110 (2008). https://doi.org/10.1103/PhysRevE.78.046110
49. Langville, A.N., Meyer, C.D.: Google's PageRank and Beyond - The Science of Search Engine Rankings. Princeton University Press, Princeton (2006)
50. Leskovec, J., Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets, 2nd edn. Cambridge University Press, Cambridge (2014)
51. Livne, O.E., Brandt, A.: Lean algebraic multigrid (LAMG): fast graph Laplacian linear solver. SIAM J. Sci. Comput. **34**(4), B499–B522 (2012). https://doi.org/10.1137/110843563
52 SPP. von Looz, M., Meyerhenke, H.: Querying probabilistic neighborhoods in spatial data sets efficiently. In: Mäkinen, V., Puglisi, S.J., Salmela, L. (eds.) IWOCA 2016. LNCS, vol. 9843, pp. 449–460. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44543-4_35
53 SPP. von Looz, M., Meyerhenke, H.: Updating dynamic random hyperbolic graphs in sublinear time. ACM J. Exp. Algorithmics **23**, 1–30 (2018). https://doi.org/10.1145/3195635
54 SPP. von Looz, M., Meyerhenke, H., Prutkin, R.: Generating random hyperbolic graphs in subquadratic time. In: Elbassioni, K., Makino, K. (eds.) ISAAC 2015. LNCS, vol. 9472, pp. 467–478. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48971-0_40
55 SPP. von Looz, M., Özdayi, M.S., Laue, S., Meyerhenke, H.: Generating massive complex networks with hyperbolic geometry faster in practice. In: HPEC, pp. 1–6. IEEE (2016). https://doi.org/10.1109/HPEC.2016.7761644
56 SPP. von Looz, M., Wolter, M., Jacob, C.R., Meyerhenke, H.: Better partitions of protein graphs for subsystem quantum chemistry. In: Goldberg, A.V., Kulikov, A.S. (eds.) SEA 2016. LNCS, vol. 9685, pp. 353–368. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-38851-9_24
57. Mahmoody, A., Tsourakakis, C.E., Upfal, E.: Scalable betweenness centrality maximization via sampling. In: KDD, pp. 1765–1773. ACM (2016). https://doi.org/10.1145/2939672.2939869
58 SPP. Meyerhenke, H., Nöllenburg, M., Schulz, C.: Drawing large graphs by multilevel maxent-stress optimization. IEEE Trans. Vis. Comput. Graph. **24**(5), 1814–1827 (2018). https://doi.org/10.1109/TVCG.2017.2689016
59. Mocnik, F.B.: The polynomial volume law of complex networks in the context of local and global optimization. Sci. Rep. **8**(1), 1–10 (2018). https://doi.org/10.1038/s41598-018-29131-0
60. Mocnik, F.-B., Frank, A.U.: Modelling spatial structures. In: Fabrikant, S.I., Raubal, M., Bertolotto, M., Davies, C., Freundschuh, S., Bell, S. (eds.) COSIT 2015. LNCS, vol. 9368, pp. 44–64. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23374-1_3

61. Nastos, J., Gao, Y.: Familial groups in social networks. Soc. Netw. **35**(3), 439–450 (2013). https://doi.org/10.1016/j.socnet.2013.05.001
62. Newman, M.: Networks, 2nd edn. Oxford University Press, Oxford (2018)
63. Newman, M.E.J., Girvan, M.: Finding and evaluating community structure in networks. Phys. Rev. E **69**, 026113 (2004). https://doi.org/10.1103/PhysRevE.69.026113
64 SPP. Nocaj, A., Ortmann, M., Brandes, U.: Untangling the hairballs of multi-centered, small-world online social media networks. J. Graph Algorithms Appl. **19**(2), 595–618 (2015). https://doi.org/10.7155/jgaa.00370
65. Peel, L., Larremore, D.B., Clauset, A.: The ground truth about metadata and community detection in networks. Sci. Adv. **3**(5), e1602548 (2017). https://doi.org/10.1126/sciadv.1602548
66. Riondato, M., Kornaropoulos, E.M.: Fast approximation of betweenness centrality through sampling. In: WSDM, pp. 413–422. ACM (2014). https://doi.org/10.1145/2556195.2556224
67. Rochat, Y.: Closeness centrality extended to unconnected graphs: the harmonic centrality index. In: ASNA, Applications of Social Network Analysis (2009)
68. Rotta, R., Noack, A.: Multilevel local search algorithms for modularity clustering. ACM J. Exp. Algorithmics **16**, 27 (2011). https://doi.org/10.1145/1963190.1970376
69. Satuluri, V., Parthasarathy, S., Ruan, Y.: Local graph sparsification for scalable clustering. In: SIGMOD Conference, pp. 721–732. ACM (2011). https://doi.org/10.1145/1989323.1989399
70 SPP. Şimşek, M., Meyerhenke, H.: Combined centrality measures for an improved characterization of influence spread in social networks. J. Complex Netw. **8**(1), cnz048 (2020). https://doi.org/10.1093/comnet/cnz048
71 SPP. Staudt, C., Marrakchi, Y., Meyerhenke, H.: Detecting communities around seed nodes in complex networks. In: BigData, pp. 62–69. IEEE Computer Society (2014). https://doi.org/10.1109/BigData.2014.7004373
72. Staudt, C., Meyerhenke, H.: Engineering high-performance community detection heuristics for massive graphs. In: ICPP, pp. 180–189. IEEE Computer Society (2013). https://doi.org/10.1109/ICPP.2013.27
73 SPP. Staudt, C.L., Hamann, M., Gutfraind, A., Safro, I., Meyerhenke, H.: Generating realistic scaled complex networks. Appl. Netw. Sci. **2**, 36 (2017). https://doi.org/10.1007/s41109-017-0054-z
74 SPP. Staudt, C.L., Meyerhenke, H.: Engineering parallel algorithms for community detection in massive networks. IEEE Trans. Parallel Distrib. Syst. **27**(1), 171–184 (2016). https://doi.org/10.1109/TPDS.2015.2390633
75 SPP. Staudt, C.L., Sazonovs, A., Meyerhenke, H.: NetworKit: a tool suite for large-scale complex network analysis. Netw. Sci. **4**(4), 508–530 (2016). https://doi.org/10.1017/nws.2016.20
76 SPP. Wegner, M., Taubert, O., Schug, A., Meyerhenke, H.: Maxent-stress optimization of 3D biomolecular models. In: ESA, pp. 70:1–70:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.ESA.2017.70
77 SPP. Zweig, K.A.: Network Analysis Literacy - A Practical Approach to the Analysis of Networks. Lecture Notes in Social Networks. Springer, Vienna (2016). https://doi.org/10.1007/978-3-7091-0741-6

Generating Synthetic Graph Data from Random Network Models

Ulrich Meyer and Manuel Penschuck(✉)

Goethe University Frankfurt, Frankfurt, Germany
{umeyer,mpenschuck}@ae.cs.uni-frankfurt.de

Abstract. Network models are developed and used in various fields of science as their design and analysis can improve the understanding of the numerous complex systems we can observe on an everyday basis. From an algorithmics point of view, structural insights into networks can guide the engineering of tailor-made graph algorithms required to face the *big data* challenge.

By design, network models describe graph classes and therefore can often provide meaningful synthetic instances whose applications include experimental case studies. While there exist public network libraries with numerous datasets, the available instances do not fully satisfy the needs of experimenters, especially pertaining to size and diversity. As several SPP 1736 projects engineered practical graph algorithms, multiple sampling algorithms for various graph models were designed and implemented to supplement experimental campaigns. In this chapter, we survey the results obtained for these so-called graph generators. *This chapter is partially based on* [43 SPP].

Keywords: Random graphs · Graph generator · Sampling · Parallel · Distributed · External memory

1 Motivation

Networks are the very fabric that makes societies [5,40]. As such, humanity is seeking to understand their structures, rules, and implications for centuries (see also Chapter 1). The practical importance of networks, however, only sky-rocketed with the advent of the information age. Nowadays, modern computers offer sufficient storage and processing capacity to map out most aspects of human life and the world we inhabit. They are fed by billions of interconnected sensors and computerized personal devices that produce enormous volumes of network data to be exploited.

Computer science provides the means to face this *big data challenge*. However, a formal grammar capturing the inner structure of the data expected to be processed is required to provide tailor-made solutions. Network models are just that: a mathematical tool to describe and analyze realistic graphs. Research into and applications of these models are deeply intertwined with various fields of science.

Networks are commonly modeled by so-called *random graphs* and, therefore, represent probability distributions over the set of graphs [8]. These distributions are almost always parametrized (e.g., for the graph size or density) and typically follow implicitly from some randomized construction algorithm. Popular models are designed such that we[1] can expect certain topological properties from a randomly drawn instance: a particularly interesting goal is to reproduce the loosely defined class of *complex networks* which, among others, encompasses most social networks.

By expressing network models as random graphs, we inherit a rich set of tools from combinatorics, stochastics, and graph theory. In algorithmics we may, for instance, assume that meaningful inputs are random instances of a *suitable* network model. Then we can derive realistic formal performance predictions using average-case analysis, smoothed complexity, et cetera. In practice, such results tend to be more relevant than worst-case analysis based on pathologic structures that are implausible in applications.

Network models also enable or supplement experimental campaigns as a versatile source of synthetic data with controllable independent variables. Synthetic benchmarks are especially useful in the context of large instances where real data is typically unavailable in sufficient size, quantity, or variety. Even if the data exists, procuring and archiving it may be difficult for legal or technical reasons; this threatens the independent reproducibility of results and thus infringes on one of science's cornerstones [45].

1.1 Structure

In Sects. 1.2 and 1.3, we introduce the definitions and notation used in this chapter. The main part of the chapter is then organized by the network model type. Section 2 discusses the notation of *random graphs* in detail and introduces sampling algorithms for the $\mathcal{G}(n,p)$ and $\mathcal{G}(n,m)$ models.

Sections 3 to 5 deal with random graph classes that focus on the distribution of degrees. Preferential attachment models, and especially the *BA* model by Barabási and Albert, explain the emergence of powerlaw degree distributions in growing networks; we discuss suitable sampling algorithms, so-called (graph) generators, in Sect. 3. The *R-MAT*, also capable of producing powerlaw degree distributions, is presented in Sect. 4. In Sect. 5, we consider several solutions for the following problem: given a list of degrees, produce a uniform sample from the set of all simple graphs that satisfy these degrees. Section 6 discusses geometrically embedded random graphs including the popular *Random Hyperbolic Graphs*.

Finally, in Sect. 7, we introduce network analysis and generator software supported by the SPP 1736.

1.2 Notation

A graph $G = (V,E)$ models a set of objects (nodes) $V = \{v_1,\ldots,v_n\}$ and their connections E (edges). Throughout this chapter, we will denote the numbers of nodes and edges as $n = |V|$, and $m = |E|$ respectively. A graph class is called *sparse* if $m = \mathcal{O}(n \operatorname{polylog} n)$ and *dense* if $m = \Theta(n^2)$.

Edges can encode a direction (i.e., $E \subseteq V \times V$) or be undirected (i.e., $E \subseteq \{\{u,v\}\,|u,v \in V\}$). If not stated differently, we assume undirected graphs. An edge that

[1] In the interest of readability, "*we*" is used quite casually in this chapter. Please note that it also appears regularly in the context of work of others.

exists multiple times is called *multi-edge* and part of a *multi-graph*. A graph without multi-edges or self-loops (edges between one node) is called *simple*.

Two nodes u and v connected by an edge e are *neighbors* or said to be *adjacent*; the nodes u and v, in turn, are incident to the edge e. The number of neighbors of a given node $u \in V$ is called its *degree* $\deg(u)$. A sequence $(\deg(v_1), \ldots, \deg(v_n))$ is called a *degree sequence*. The related concept of a *degree distribution* refers to the probability distribution of the degree of a randomly sampled node (possibly in a randomly sampled graph). Many observed networks exhibit a *powerlaw* degree distribution where the probability of degree k is proportional to $k^{-\gamma}$ for some $2 < \gamma$ (and often $\gamma < 3$). A property applies *with high probability* (*w.h.p.*) if it holds with probability of at least $1 - x^{-\alpha}$ for $\alpha \geq 1$ where x depends on the context and is often the problem size n or m.

1.3 Models of Computation

The design of an algorithm is heavily influenced by the assumed model of computation. If not state differently we suppose the *unit-cost RAM* in which operations for control-flow, data access, and basic arithmetic are handled in constant time. For shared-memory parallel algorithms, its parallel variant PRAM is used. In a parallel context, we use the term *processing unit (PU)* to refer to an abstract machine executing a sequential algorithm (e.g., a core in a CPU or an individual processor in a distributed computer cluster). A problem is said to be *pleasingly parallel* if it consists of sufficiently many subproblems that can trivially be computed independently.

To model the cost of data transfer, the *external memory model* by Aggarwal and Vitter [1] assumes a two-level memory hierarchy. It consists of an internal memory of size M and an unbounded external disk which holds the algorithm's input and output. Computation is free, but is only possible on data in internal memory and therefore has to be move to and fro. Data access is block-oriented and transfers B data items per *I/O*. Reading and writing N contiguous items is referred to as *scanning* and requires $\text{scan}(N) = \Theta(N/B)$ I/Os. Sorting such items triggers $\text{sort}(N) = \Theta((N/B)\log_{M/B}(N/B))$ I/Os and constitutes a lower-bound for most intuitively hard problems.

Analogously, the cost of communication is often a bottleneck for distributed machines consisting of interconnected processors. *Communication-agnostic* algorithms are an extreme case of communication avoidance. Each PU is only aware of its rank, the total number of PUs, and some input configuration. However, exchange of any further information during the execution of the algorithm is prohibited.

2 Random Graphs and the $G(n, p)$ and $G(n, m)$ Models

A *random graph* is a probability distribution $P\colon \mathbb{G} \to [0, 1]$ where $\mathbb{G}$ is the set of all graphs. Virtually all *random graph models*[2] are parameterized and thus form families of probability distributions. The underlying distributions are typically specified implicitly, and often have a finite support defined by some combinatorial constraints.

[2] In the literature the terms *random graph* and *random graph model* are commonly used interchangeably, and may even refer to a random instance sampled from a model. We adopt the former simplification for the sake of readability.

As an example, consider the popular $\mathcal{G}(n,p)$ model introduced by E. Gilbert [20] in 1959. In its original formulation, the model's support consists of all $2^{n(n+1)/2}$ undirected graphs with exactly n nodes. The probability distribution is given indirectly via the following sampling algorithm:

> "Pick one of these graphs by the following random process. For all pairs of points [nodes] make random choices, independent of each other, whether or not to join the points of the pair by a line [edge]. Let the common probability of joining be p." [20]

In other words, in a random instance of $\mathcal{G}(n,p)$ any edge e exists independently with probability p. Observe that $G(n,1/2)$ hence implies the uniform distribution of all graphs with n nodes. It is therefore a so-called *maximum entropy model* and sometimes even referred to as *the* random graph [5].

Erdős and Rényi [17] propose the related and well-known $\mathcal{G}(n,m)$ model as the uniform distribution over all undirected graphs with n nodes and m edges. The models $\mathcal{G}(n,p)$ and $\mathcal{G}(n,m)$ with $m = \binom{n}{2}p$ are equivalent in the limit of $n \to \infty$.

Neither $\mathcal{G}(n,p)$ nor $\mathcal{G}(n,m)$ explain the non-trivial structural properties of observed networks. Since all edges are chosen (mostly) independently with identical probabilities, we do not expect the formation of any complex features. Several ways to formalize this intuition are discussed in [44 SPP]. Still, the models are commonly used to generate synthetic data, e.g., as a null-model.

2.1 Sampling from $\mathcal{G}(n,p)$ and $\mathcal{G}(n,m)$

Gilbert's sampling algorithm is designed to communicate the model's spirit to a human reader and, as such, is not optimized for performance. The generator thus requires $\Omega(n^2)$ work independently of the linking probability p which is suboptimal for non-dense graphs.

Batagelj and SPP 1736 PI Brandes [6] describe an optimal sequential generator requiring work linear in the number m of edges produced. The algorithm fixes a convenient order of all possible edges (i.e., a bijection $\pi\colon [\binom{n}{2}] \to \{\{u,v\} \mid u,v \in V \wedge u \neq v\}$) and considers them in this sequence. Since each edge in a $\mathcal{G}(n,p)$ graph is the result of an independent Bernoulli trial, the number of "non-edges" between any two successful trials follows a geometric distribution. The generator therefore draws a random geometric variate, jumps over that many non-edges, writes out the next edge, and repeats until all possible edges have been considered.

Since all edges are drawn independently, the generator can be parallelized by partitioning the sequence of possible edges into independent sub-problems of roughly equal size. Later, Bringmann and Friedrich [9] give an exact variant of the algorithm that does not require real-valued arithmetic to sample the skip distances.

Sampling from $\mathcal{G}(n,m)$ is more challenging than $\mathcal{G}(n,p)$ since faithful $\mathcal{G}(n,m)$ generators can not assume independent trials. This is due to the fact that partially sampled edges and non-edges affect the probability distribution of the remaining candidates. While Batagelj and Brandes remark that their $\mathcal{G}(n,p)$ generator can be extended to $\mathcal{G}(n,m)$ by modifying the skip distance distribution accordingly, they continue to develop a more efficient alternative requiring work linear in the number of edges produced [6].

In the following, we however focus on a parallel approach by Funke et al. [18 SPP] and showcase general divide-and-conquer techniques used to yield communication-agnostic generators. The resulting generator is a variant of a parallel sampling algorithm [47 SPP] for the related problem of randomly selecting m distinct elements from a finite universe (i.e., sampling without replacement).

For simplicity's sake, we only consider the directed variant of $\mathscr{G}(n,m)$.[3] In order to parallelize, we partition the set of nodes V into disjoint subsets $V_1,\dots,V_p$ of roughly equal size. Then, processing unit i is tasked to produce the m_i out-going edges of nodes in V_i. By definition of $\mathscr{G}(n,m)$, we require that $\sum_i m_i = m$. Observe that this is the only dependency between subproblems. Thus, if m_i is known a priori, PU i can work independently.

Consequently, we need to find a communication-agnostic way to agree on a consistent and randomly chosen $\mathbf{m} = (m_1,\dots,m_P)$ where each PU only needs to know its own value m_i. The vector $\mathbf{m}$ follows a multinomial hypergeometric distribution where the number of "positive instances" for the i-th entry are given by the number $n \cdot |V_i|$ of potential edges processed by PU i. Under the assumption that the number P of PUs satisfies $P = \mathscr{O}(n/\log n)$ the values of m_i are sufficiently concentrated to bound the complexity of the previous local sampling to $\mathscr{O}((n+m)/P)$ w.h.p..

A traditional distributed generator may sample $\mathbf{m}$ on a central PU and then broadcast the values—this is, however, not possible in a communication-agnostic setting since it incurs a communication volume $\Omega(P)$. Alternatively, each PU can independently sample $\mathbf{m}$ with pseudo-random number generators that use a common seed value. This approach requires expected time $\Theta(P)$ and, thus, dominates the total runtime for $P = \omega(\sqrt{m})$.

Thus, we rather follow a divide-and-conquer approach which works for various distributions and is also used in Sect. 6.3. Roughly speaking, each m_i corresponds to a leaf in a binary tree of depth $\mathscr{O}(\log P)$. At each inner node, we draw a random variate x from an appropriately parametrized hypergeometric distribution and interpret x as the number of edges to be produced in the left subtree. Each PU follows its unique path from the root to the i-th leaf to sample its own value of m_i. To achieve consistent values, the sampling at each inner node is carried out using a pseudo-random number generator whose seed is deterministically derived from a unique node index.

The authors show that combining these ideas yields a communication-agnostic generator with a runtime complexity of $\mathscr{O}((n+m)/P + \log P)$ w.h.p..

3 Preferential Attachment

Barabási and Albert [4] propose a simple stochastic process to explain the emergence of scale-free networks and show that two ingredients, namely growth and selection bias, suffice to yield networks with powerlaw degree distributions.[4]

[3] The undirected [18 SPP] variant only differs in the partitioning of the parallel subproblems.

[4] Earlier, Price [50] proposed a similar process inspired by Pólya urns [16]. The author applies it to citation networks with a known powerlaw in-degree distribution [46]. The more widespread *BA* model is sometimes interpreted as a special case of Price's model.

At its core, their *BA* model relies on *preferential attachment*, a positive feedback loop in dynamic systems where selecting an item at one point in time increases the probability of selecting it again in the future. It is proverbially summarized as “the rich get richer”.

Based on this idea, the authors describe the following random graph. Starting with an arbitrary seed graph G_0 with n_0 vertices and m_0 edges, we iteratively add $n - n_0$ nodes—one node at a time. For each new node, we choose d neighbors at random where the probability to select node v is proportional to the degree of v at that time.

The main algorithmic challenge of *BA* lies in this dynamic weighted sampling. Depending on the assumed model of computation, quite different solutions are available. Batagelj and Brandes [6] observe that each node with degree k appears exactly k-times in the edge list produced so far. Therefore, the underlying dynamic weighted sampling problem can be reduced to uniformly selecting entries from the edge list, leading to the linear-time generator BB-BA.

As BB-BA requires unstructured I/Os, it cannot efficiently produce graphs that do not fit into main memory. Meyer and Penschuck [36 SPP] introduce TFP-BA and MP-BA, the first two I/O-efficient sampling approaches for random graph models based on preferential attachment. The authors initially focus on *BA* graphs to demonstrate the techniques and subsequently discuss additional features such as seed graphs exceeding main memory, nodes with inhomogenous initial degrees, the inclusion of uniform node sampling, directed graphs, and edges between two randomly chosen nodes.

- TFP-BA is a simple and easily generalizable sequential generator inspired by BB-BA. Rather than reading from random positions in the edge list, TFP-BA first precomputes all necessary read operations and sorts them by the memory address they read from. As the algorithm produces the edge list monotonously moving from beginning to end, it scans through the sorted read request and forwards the still cached values to the target positions using an I/O-efficient priority queue. This approach requires $\mathcal{O}(\text{scan}(m_0) + \text{sort}(m))$ I/Os, where m_0 is the number of edges in the seed graph and m is the number of edges produced.
- MP-BA is a parallel generator that offloads a number of subtasks onto a general-purpose graphics processor. The algorithm implements dynamic weighted sampling using a binary tree T partially stored in external memory. Each node u of the generated graph corresponds to a leaf in T labeled with the degree of u; any inner node stores the total weight of all leaves contained in its left subtree.
  In order to select a neighbor, MP-BA first has to sample a leaf according to the current degree distribution and then increment the leaf's weight to account for the newly gained edge. The key insight is that we can do both in a single top-down traversal from the root to the sampled leaf. This allows us to combine the queries for sampling and updating into a single operation and, in turn, to coalesce queries into batches. MP-BA requires $\mathcal{O}(\text{sort}(n_0 + m))$ I/Os, where n_0 is the number of nodes in the seed graph and m is the number of edges produced.
  The algorithm uses two forms of parallelism: firstly, T is cut at a certain depth to process the subtrees rooted there pleasingly parallel. In order to handle the high volume of requests near T's root, a dedicated PRAM algorithm processes multiple requests

to the same tree node in parallel. MP-BA's implementation executes the latter part on a GPU for maximal throughput.

Sanders and Schulz [48 SPP] describe CA-BA, a communication-agnostic generator for distributed-memory parallelism. Their algorithm builds on top of BB-BA and uses pseudo-randomization to avoid all lookups to edges generated. By doing so, several PUs can work on the problem without exchanging information other than an initial broadcast of the seed graph and a few parameters. In contrast to the original algorithm, CA-BA does not maintain an edge list to sample from explicitly. To simplify the description, we still presume its existence as a concept for addressing.

In order to add an edge, the generator needs to place the indices of the two incident nodes into the edge list. Recall that each generated edge consists of a newly introduced node and a randomly selected neighbor. By convention, we store the former at even positions of the edge list, and the latter at odd positions. Since by definition of the *BA* model, each newly introduced node is initially incident to exactly d edges, all entries at even positions follow from a simple index transformation.

Sampling random neighbors involves a shared random hash function $h(\cdot)$ with the property that $h(i) < i$. Then, in order to choose the node index of the random neighbor to be written to the edge list's i-th position, we conceptually copy the value from position $j = h(i)$. To do so, we distinguish three cases:

1. If $j < 2m_0$, we need to retrieve a value from the seed graph which is a simple read access to the input data. It is the only case where an actual memory access needs to be carried out.
2. If $j \geq 2m_0$ and j is even, we can compute the value stored there using the aforementioned index transformation.
3. If $j \geq 2m_0$ and j is odd, we retrace the sampling carried out. To do so, we recurse on $j' = h(j) = h(h(i))$.

The first two cases imply constant work on a unit-cost RAM. Since we assume $h(\cdot)$ to be a random function, the first two cases are chosen with probability of at least $1/2$. Thus, the recursion of the last case has an expected depth of at most 2 and is $\mathscr{O}(\log m)$ with high probability. Assuming $h(\cdot)$ can be evaluated in constant time, CA-BA therefore requires expected linear work.

4 R-MAT

R-MAT [15] graphs are a well-accepted network model which is especially known for its use in the Graph500 benchmark [38]. The model is defined for graphs on 2^k nodes and m edges. To sample an edge, we recursively subdivide the adjacency matrix into four quadrants, assign them probabilities $p_a + p_b + p_c + p_d = 1$ provided as model parameters, and randomly select one. We repeat this k times until we reach a matrix of size 1×1 which corresponds to the edge. Depending on the model, we either allow multi-edges, or reject and resample to avoid duplicates. Undirected graphs are possible and typically imply additional symmetry constraints on the quadrant probabilities. For certain sets of parameters, the model exhibits similarities to observed networks such a powerlaw degree distribution [34].

Following the recursive definition, there exists a bijection between each possible edge and the set of words Σ^k over $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ where each $x \in \Sigma$ represents the quadrant chosen. A naive *R-MAT* generator explicitly samples the k characters, one after another, and thus requires $\Omega(\log n)$ work per edge.

Hübschle-Schneider and Sanders [27 SPP] propose a communication-agnostic scheme that instead samples edges in constant time under the reasonable assumption that $m = \Omega(n)$. The algorithms performs a preprocessing step to construct an urn which contains n^α path fragments (for some $\alpha < 1$) weighted by their probabilities in time $\mathcal{O}(n)$, e.g., by considering all words Σ^ℓ of fixed length $\ell = \log_2 \sqrt{n} = k/2$.

To draw an edge we sample $\lceil k/\ell \rceil = \mathcal{O}(1)$ fragments. We then concatenate them using bit-parallel shifting and masking operations available in virtually all modern computers. Both steps require only constant time per edge.

5 Simple Graphs from Prescribed Degree Sequence

The sampling of random graphs matching a prescribed degree sequence is a common task in network analysis. Its various applications range from to the construction of null-models (e.g., Chapter 3) to use-cases as building blocks in graph generators. Instances of the latter are the popular *LFR* benchmark [28] or the derived *ReCon* [51 SPP] model to generate scaled-replicas of an input graph.

The computational cost of this approach heavily depends on the exact formulation of the model. Two models with linear work sampling algorithms are the *Chung-Lu* (*CL*) model and the *Configuration Model* (*CM*). The *CL* model produces the prescribed degree sequence only in expectation (see [44 SPP] for details). The *CM*, on the other hand, exactly matches the prescribed degree sequence but permits self-loops and multi-edges. These parallel edges affect the uniformity of the model [39, p. 436] and are inappropriate for certain applications; however, erasing them may lead to significant changes in topology [49 SPP]. In the following, we focus on simple graphs (i.e., without self-loops or multi-edges) matching a prescribed degree sequence exactly. Several generators and models for such graphs were considered within the SPP 1736.

5.1 The Edge Switching Markov Chain Model

The *Fixed-Degree-Sequence-Model* (*FDSM*) is a common solution to obtain simple graphs from a prescribed degree sequence. It first manifests a biased deterministic graph (e.g., using the HAVEL-HAKIMI algorithm [23,26]) and then uses an *Edge Switching* (*ES*) Markov chain process [21] to perturb the graph. In each step, the process selects two edges uniformly at random and exchanges their incident nodes—by doing so the degrees of all nodes involved do not change. If a step were to result in a self-loop or multi-edge, it is rejected without replacement. Despite intensive research, it remains an open problem to find *practical* upper bounds on the Markov chain's mixing time; i.e., the number of steps required to obtain a uniform sample. In practice, a small multiple of the number of edges typically suffices (cf. Chapter 3).

The main issue when implementing *ES* is the large number of unstructured accesses to memory; for each switch it is necessary to identify the involved nodes, check whether the updated edges already exist, and finally to write out the updates.

Hamann et al. [25 SPP] describe EM-LFR, an I/O-efficient pipeline to sample large instances from the *LFR* model. From an algorithmic point of view, two central parts of the pipeline are EM-HH and EM-ES which together implement *FDSM*. EM-HH is designed to avoid memory accesses as best as possible especially for graphs with powerlaw distributions. EM-ES, on the other hand, batches $\Theta(m)$ individual swaps and processes them out-of-order without changing the outcome; due to the large number of swaps in each batch, we can amortize the I/O volume and stream through the whole graph a constant number of times rather than executing $\Theta(m)$ more expensive unstructured accesses.

Later, [24 SPP] propose a modification of the *FDSM* model and provide empirical evidence of faster mixing. The previous combination of EM-HH followed by EM-ES starts with a highly biased simple graph. The novel EM-CM/ES takes another route: It starts with a random but non-simple graph and switches edges until a simple random graph is obtained. It uses an I/O-efficient generator for the *Configuration Model* and a variant of EM-ES which accepts non-simple inputs without increasing its I/O complexity. The modified algorithm executes all switches that neither increase the multiplicity of a given edge nor introduce self-loops. Non-simple edges are also switched more frequently than legal edges to accelerate the repair phase. Observe, however, that it does not suffice to rewire non-simple edges using the presented variant of *ES* as it produces a biased sample [2,3]. Instead, additional *ES* steps are necessary.

Brugger et al. [12 SPP] implement *ES* in hardware (see Chapter 4 for details). Their design maintains the graph in a hybrid data structure combining an adjacency list to efficiently sample edges and an adjacency matrix for fast edge existence queries. Then, the authors investigate two cases:

- First they perturb several independent graphs pleasingly parallel where each graph is stored in a dedicate physical region (DRAM channel) of common memory chips. In this setting, the authors optimize the memory controller to address channels independently and interleave these requests.
- In a second step, parallelism within a single *ES* run is exposed using the following observation: If the graph is sufficiently large, the probability that a short run of multiple switches target a common edge is small (cf. birthday problem). The authors therefore describe a hardware design that checks whether 12 contiguous switches are collision-free and if so, execute them in parallel.

5.2 Curveball

Curveball (*CB*) [52] is a more recent process but structurally similar to *ES*; instead of selecting random edges, *CB* selects two random nodes $u \neq v$, and *trades* their neighborhoods as follows. *CB* begins by freezing all edges that either connect u and v themselves or link to neighbors which u and v have in common. Then, the remaining neighbors are randomly shuffled while maintaining the degrees of u and v. A single *CB* trade can therefore inflict “more change” to a graph than a single edge switch; depending on the processed graph, a state in *CB*’s Markov chain may have up to $2^{\Theta(n)}$ neighbors while the degrees in *ES*’s chain are bounded by $\mathcal{O}(n^4)$ [13]. Empirical data suggests that fewer trades are necessary to mix a graph (though each trade may require more work).

CB exposes more data locality than *ES* since all information required to carry out a trade is contained in the two neighborhoods. This is in contrast to *ES*, which requires additional unstructured reads to prevent a switch from introducing multi-edges. Note, however, that an undirected edge is classically stored twice—once for each endpoint. In this scenario, frequent unstructured updates are necessary and negate the previously mentioned locality benefits.

The I/O-efficient EM-CB algorithm [14 SPP] thus relies on a dynamic data structure and assigns each edge only to the endpoint that is traded next. EM-CB uses the external-memory technique *Time Forward Processing* (*TFP*, see [35]) to ensure that the complete neighborhood of a node is available when needed.

The algorithm works in batches. At the beginning of each batch, it samples the node pairs to be traded within the batch and organizes them in dedicated indices. These auxiliary data structures are used to address the *TFP* messages and to determine which endpoint of an edge will be traded first. EM-CB requires $\mathcal{O}(r[\text{sort}(n) + \text{sort}(m)])$ I/Os to carry out r global trades (see below).

Carstens et al. [14 SPP] generalize *Global Curveball* (*G-CB*) to undirected graphs. An *undirected global trade* is a sequence of $\lfloor n/2 \rfloor$ single trades such that the neighborhood of each node is traded at most once. They show that the process converges to a uniform distribution over the set of all graphs and give empirical evidence of its superior performance compared to *CB*.

Since each node participates once[5] in a global trade, we can interpret a global trade as a random permutation of nodes where we trade pairwise adjacent nodes. The authors then propose an algorithm that eliminates the auxiliary data structures by maintaining the permutation implicitly using a collision-free (on the relevant domain) and invertible hash function, and finally give a parallel version of it.

6 Geometrically Embedded Random Graphs

Random Hyperbolic Graphs (*RHGs*) are a popular network model which naturally exhibits many features commonly observed in complex networks. *RHG* assigns each node a position on a two-dimensional hyperbolic disk of radius R. These positions are conveniently expressed in polar coordinates where each point is located in terms of its distance r (radius) to the disk's center and an angular coordinate θ.

In the so-called *Threshold RHG* [22], we connect all pairs of points (r_i, θ_i) and (r_j, θ_j) with $i \neq j$ whose hyperbolic distance $d(p_i, p_j)$ is smaller than R, where

$$\cosh(d(p_i, p_j)) = \cosh(r_i)\cosh(r_j) - \sinh(r_i)\sinh(r_j)\cos(\theta_i - \theta_j). \tag{1}$$

Thus, the hyperbolic distance is a function of the relative and absolute positions of both points; the closer a point is to the disk's center, the more neighbors it is expected to have. We obtain a powerlaw degree distribution with a controllable exponent by choosing an appropriate radial density for the randomly placed points.

[5] For simplicity, we assume here that n is even.

Binomial RHG extends *Threshold RHG* by adding a positive *temperature* parameter T that affects the local cohesion. In the binomial variant, each pair of nodes $p_i \neq p_j$ is independently connected by an edge with probability $p_T(d(p_i, p_j))$ defined as follows:

$$p_T(d) = \left[\exp\left(\frac{d-R}{2T}\right) + 1\right]^{-1} \tag{2}$$

Binomial RHG contains *Threshold RHG* as p_T becomes a step function for $T \to 0$. Looz and Meyerhenke [31 SPP] propose an extension of the *RHG* model to generate dynamic graph data sets: their model adds movement of nodes which in turn translates to a stream of edge insertions and deletions.

6.1 Efficient Generators Based on Geometric Data Structures

A naive *RHG* generator that checks each node pair for an edge requires $\Omega(n^2)$ work and little parallel depth[6]. All efficient generators we are aware of reduce the computational complexity in a two step process: they cheaply identify a set of edge candidates (i.e., a super-set of the true result), and then filter the candidates more carefully. The identification typically exploits geometrical or stochastic arguments, while the filtering process tends to involve costly per edge distances computations.

All geometric generators discussed in the remainder of this chapter use one of two geometric partitioning schemes, namely a quad-tree or a band structure.

- Looz et al. [32 SPP] describe NkQuad, the first sub-quadratic work *RHG* generator. NkQuad is based on a polar quad-tree which recursively subdivides the space into four quadrants each (i.e., each inner tree-node introduces two cuts, one in the angular and one in radial dimension, respectively). The generator then iterates over all nodes and computes for each $v \in V$ the neighbor candidates C_v. The set C_v consists of all nodes in quad-tree leaf cells which intersect the hyperbolic circle of radius R around v. The identification of such leafs is simplified by working in the Poincare projection which translates hyperbolic circles into (radially shifted) Euclidean circles. The authors show that such a query examines $\mathcal{O}(\sqrt{n} + |C_v|)$ leafs w.h.p., leading to total work of $\mathcal{O}((n^{3/2} + m)\log n)$ w.h.p..
  Later, Looz and Meyerhenke [30 SPP] generalize the data structure and extend the generator to *Binomial RHG* while maintaining the asymptotic complexity. The efficient sampling of low-probability edges is implemented by bounding the probability to connect to any edge within a leaf from above. These bounds are used to carry out geometric jumps (cf. Sect. 2) followed by rejection sampling to account for the over-estimation.
- Looz et al. [33 SPP] improve NkQuad by proposing NkBand featuring a novel partitioning scheme. NkBand covers the hyperbolic disk with $\Theta(\log n)$ disjoint concentrical bands where each band is maintained as an array of points sorted by their angles. To find the neighbor candidates of a node v in band b_i, the algorithms considers b_i and all bands containing larger radii. For each such band b_j, the smallest and largest angular coordinate of a potential neighbor of v in b_j is computed; then

[6] Dependencies may arise from the output format, e.g., from a need for compaction.

two binary search yield the left- and right-most candidates in the sorted array. By doing so, the authors effectively over-estimate the upper half of the hyperbolic circle around node v by a discrete stack of shrinking band-segments. The generator has an empirical runtime of $\mathcal{O}(n \log n + m)$. Later, Looz [29 SPP] extends NkBand to *Binomial RHG* using ideas similarly to the generalization described for NkQuad.

6.2 A Fast and Memory-Efficient Streaming Generator for RHG

As the geometric data structures discussed for NkQuad, NkBand, and HyperGirgs have a large memory footprint that can render them unsuitable for accelerator hardware with a small dedicated memory, [42 SPP] presents HyperGen, a streaming generator for *Threshold RHGs* which instead samples the points on demand. The generator requires $\mathcal{O}([n^{1-\alpha}\bar{d}^{\alpha} + \log n]\log n)$ words of memory w.h.p.. For realistic average degrees $\bar{d} = o(n/\log^{1/\alpha}(n))$ this is a significant asymptotic reduction over classical approaches.

HyperGen executes a sweep-line algorithm and stores the set of nodes that may still find neighbors in its sweep-line state; we refer to them as *candidates*. Roughly speaking, the algorithm randomly samples points with non-decreasing angular coordinates.[7] For each new point, the algorithm identifies all sufficiently close candidates and emits edges to them. The generator then marks the point a candidate itself and advances the sweep-line. HyperGen stops the sweep-line at additional points, e.g., to prune candidates whose distances to the sweep-line are so large that they cannot find neighbors anymore.

To manage the computational cost of maintaining the sweep state, HyperGen includes conservative approximations that do not infringe on the generator's faithful reproduction of *RHGs*. They exploit the distribution of points as well as properties of the hyperbolic distance function. The majority of points can be quickly pruned from the algorithm's state. In contrast, the few points that have small radii stay candidates for a significantly longer period of time. To accommodate the different requirements, HyperGen partitions the hyperbolic disk into $\Theta(\log n)$ concentrical bands. Each band has its own sweep-line and state which remain synchronized with the states of its adjacent bands.

Observe that, due to the angular periodicity of the hyperbolic disk, points sampled late (i.e., with angles near 2π) can be adjacent to points discovered and pruned much earlier. HyperGen accounts for this by restarting the sampling process until all candidates of the first phase are processed. It exploits pseudorandomness to obtain consistent point coordinates in both phases.

Parallelization is possible by splitting the disk into segments of equal size. Some care has to be taken to manage the dependencies near the segments' borders. HyperGen also significantly accelerates the frequent distance computations by preparing auxiliary values per point. This removes all transcendental functions (here sinh, cosh, and cos) from Eq. (1). Refined versions of these techniques carry over to Sects. 6.3 and 6.4.

The implementation of HyperGen is designed with SIMD (Single-Instruction-Multiple-Data) in mind and is explicitly vectorized. It uses SIMD instructions to com-

[7] This is an over-simplification of the sweep-line's behavior (cf. [42 SPP]).

pute eight hyperbolic distances simultaneously (which is only possible because we first removed the aforementioned transcendental functions).

6.3 Communication-Agnostic Generators for RHG

Funke et al. [19 SPP] present RHG, a communication-agnostic generator for *Threshold RHG*. The generators RHG and HYPERGEN were developed independently at roughly the same time, and share ideas to sample specific subsections of the hyperbolic disk using pseudorandomization. While HYPERGEN uses a monotonous sweep-like motion optimized for memory usage, RHG uses less structured queries. These “random” queries are answered using a fine-grained partitioning of the hyperbolic space which ingeniously allows random access to any cell (the geometry is similar to the one discussed in Sect. 6.1).

For huge graph instance, the number of nodes may be too large to sample —let alone store— all nodes on every distributed machine. Fortunately, a key property of relevant *RHG* graphs is that most nodes only have a very local neighborhood, i.e., a hyperbolic circle around each node suffices to compute all its links. Observe that many of these subsets overlap due to common edges. In general, there is no balanced mapping of nodes to processing units without overlaps. Thus, any two PUs with overlapping subsets have to have a consistent view of the underlying region of hyperbolic space.

We achieve this by partitioning the hyperbolic space into k cells. Then, the following process reproducibly samples points within a cell. First, a hash function f is used to seed a pseudorandom number generator with the value $f(i)$. For each cell i, we seed a pseudorandom generator with a value deterministically derived from the cell’s index i and, subsequently, use the generator to sample the n_i points contained within the cell. By construction, this process yields consistent results even if executed by multiple independent processing units.

The only information missing is the number n_i of points in cell i. The vector $\mathbf{N} = (n_1, \ldots, n_k)$ follows a multinomial hypergeometric distribution due to the side condition that exactly n points need to be scattered in total, i.e., $\sum_i n_i = n$. All PUs obtain consistent values for $\mathbf{N}$ using common seeds for their pseudorandom generators analogously to the divide-and-conquer approach in Sect. 2.1.

In [18 SPP], this techniques is combined with HYPERGEN (see Sect. 6.2) yielding the communication-agnostic sweep-line generator sRHG which consistently outperforms RHG. We demonstrate its scalability to up to 32 768 cores and produce a graph with $n = 2^{39}$ nodes in less than a minute.

6.4 GIRG-Based Generator

Bringmann et al. propose Geometric Inhomogenous Random Graphs as a flexible and simple model, that asymptotically contains *RHG* [11]. Roughly speaking, the model embeds a graph into an d-dimensional torus and uses node weights to control the degree sequence similarly to the Chung-Lu model. The authors also give an expected linear time sampling algorithm for GIRGs [10] which we engineer adapt[8] it to *Binomial RHGs*

[8] Bringmann et al. already discuss the applicability to *RHG*. The models are however not identical [7 SPP], and HYPERGIRGS closes this gap.

in [7 SPP]. We refer to our algorithms as GIRGS and HYPERGIRGS, respectively. To the best of our knowledge, GIRGS is the first practically efficient generator for the *GIRG* model. Here, we focus on *RHGs* since the algorithmic treatment of both models is very similar.

HYPERGIRGS first samples all points and builds a data structure that can be interpreted as a polar quad-tree. While the structure is similar to the previous state-of-the-art generator NKQUAD (see Sect. 6.1), differences in details result in a polynomial gap in their running times. In the following, we refer to nodes of the quad-tree as *tree-nodes* (to distinguish them from the hyperbolic nodes contained).

Bringmann et al. propose the following neighborhood search which is adapted by HYPERGIRGS. For simplicity, we initially restrict ourselves to *Threshold RHGs*. The generator enumerates all pairs of tree-nodes that may contain point pairs sufficiently close to imply an edge. This is done in a pessimistic and oblivious fashion, i.e., without considering the actual points represented by the tree-nodes. HYPERGIRGS then emits edges by testing all point pairs contained in each previously enumerated pair of tree-nodes. To avoid asymptotically significant overheads, the algorithm pairs tree-nodes as high up in the quad-tree as possible without adding unintended distance computations.

The quad-tree needs to support efficient random access to all points contained within any tree-node at any depth. Similarly to [10], HYPERGIRGS achieves this using z-order space-filling curves [41] to map the tree to memory. This choice allows us to efficiently build and query the quad-tree using Morton codes [37].

In case of *Binomial RHGs* with $T > 0$, any node pair has a positive (yet mostly negligible) probability $p_T(d)$ to be connected. HYPERGIRGS therefore has to consider all tree-node pairs—even those with a tiny connection probability. In the latter case, the connection probability is bounded from below. Then, we use geometric jumps followed by rejection sampling to prune the search space. The authors also engineer an exact look-up table-based sampling scheme to reduce the evaluation of transcendental functions during the computation of linking probabilities $p_T(d)$.

HYPERGIRGS processes the tree-node pairs pleasingly parallel. As a special feature, its implementation guarantees reproducibility in the sense that two runs with the same set of parameters and seed values output the same set of edges (though not necessarily in the same order). At the time of writing, the implementation of HYPERGIRGS is the fastest sequential *RHG* generator and competitive for shared-memory parallelism.

7 Software Packages

From a practical point of view, it is crucial that a generator interacts well with the software used to analyze the emitted graphs. A common choice is to write the produced graph into a file which then can be processed by a tool of choice. There are, however, notable drawbacks of this approach; for one, there are a plethora of file formats which may be incompatible. Also reading and writing files can have surprisingly high overheads (e.g., [42 SPP]).

The network analysis framework NetworKit (partially supported by the SPP 1736) includes generators for all network models that are discussed in depth in this chapter. As detailed in Chapter 1, this software package combines various types of graph algorithms efficiently implemented in C++ with an easy to use Python interface. The tight interaction between network generation and analysis promises a fast and convenient processing pipelines.

KaGen is a graph generator suite for distributed computing and contains a number of communication-agnostic generators [18 SPP]. The suite includes generators for the following models accessible via a common interface $\mathcal{G}(n,p)$, $\mathcal{G}(n,m)$, *Kronecker Graph*, *Random Geometric Graph*, *Random Delaunay Triangulation*, *Barabási-Albert*, and *Threshold RHG*.

Acknowledgements. The authors thank Mario Holldack and Hung Tran for valueable discussions and their insightful comments.

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM **31**(9), 1116–1127 (1988). https://doi.org/10.1145/48529.48535
2. Allendorf, D.: Implementation and evaluation of a uniform graph sampling algorithm for prescribed power-law degree sequences. Master's thesis. Goethe University Frankfurt, Germany (2020)
3. Arman, A., Gao, P., Wormald, N.C.: Fast uniform generation of random graphs with given degree sequences. In: FOCS, pp. 1371–1379. IEEE Computer Society (2019). https://doi.org/10.1109/FOCS.2019.00084
4. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439), 509–512 (1999)
5. Barabási, A.L., et al.: Network Science. Cambridge University Press, Cambridge (2016)
6. Batagelj, V., Brandes, U.: Efficient generation of large random networks. Phys. Rev. E **71**(3), 036113 (2005). https://doi.org/10.1103/physreve.71.036113

7 SPP. Bläsius, T., Friedrich, T., Katzmann, M., Meyer, U., Penschuck, M., Weyand, C.: Efficiently generating geometric inhomogeneous and hyperbolic random graphs. In: ESA, pp. 21:1–21:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ESA.2019.21

8. Bollobás, B.: Random Graphs, 2nd edn. Cambridge Studies in Advanced Mathematics, vol. 73. Cambridge University Press, Cambridge (2011). https://doi.org/10.1017/CBO9780511814068
9. Bringmann, K., Friedrich, T.: Exact and efficient generation of geometric random variates and random graphs. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013. LNCS, vol. 7965, pp. 267–278. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39206-1_23
10. Bringmann, K., Keusch, R., Lengler, J.: Sampling geometric inhomogeneous random graphs in linear time. In: ESA, pp. 20:1–20:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.ESA.2017.20
11. Bringmann, K., Keusch, R., Lengler, J.: Geometric inhomogeneous random graphs. Theor. Comput. Sci. **760**, 35–54 (2019). https://doi.org/10.1016/j.tcs.2018.08.014

12 SPP. Brugger, C., et al.: A memory centric architecture of the link assessment algorithm in large graphs. IEEE Des. Test **35**(1), 7–15 (2018). https://doi.org/10.1109/MDAT.2017.2750900

13. Carstens, C.J., Berger, A., Strona, G.: Curveball: a new generation of sampling algorithms for graphs with fixed degree sequence. CoRR abs/1609.05137 (2016)

14 SPP. Carstens, C.J., Hamann, M., Meyer, U., Penschuck, M., Tran, H., Wagner, D.: Parallel and I/O-efficient randomisation of massive networks using global curveball trades. In: ESA, pp. 11:1–11:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.ESA.2018.11
15. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: a recursive model for graph mining. In: SDM, pp. 442–446. SIAM (2004). https://doi.org/10.1137/1.9781611972740.43
16. Eggenberger, F., Pólya, G.: Über die Statistik verketteter Vorgänge. ZAMM-J. Appl. Math. Mech./Zeitschrift für Angewandte Mathematik und Mechanik **3**(4), 279–289 (1923)
17. Erdős, P., Rényi, A.: On random graphs I. Publicationes Mathematicae Debrecen (1959)
18 SPP. Funke, D., et al.: Communication-free massively distributed graph generation. J. Parallel Distrib. Comput. **131**, 200–217 (2019). https://doi.org/10.1016/j.jpdc.2019.03.011
19 SPP. Funke, D., Lamm, S., Sanders, P., Schulz, C., Strash, D., von Looz, M.: Communication-free massively distributed graph generation. In: IPDPS, pp. 336–347. IEEE Computer Society (2018). https://doi.org/10.1109/IPDPS.2018.00043
20. Gilbert, E.N.: Random graphs. Ann. Math. Stat. **30**(4), 1141–1144 (1959). https://doi.org/10.1214/aoms/1177706098
21. Gkantsidis, C., Mihail, M., Zegura, E.W.: The Markov Chain simulation method for generating connected power law random graphs. In: Workshop on Algorithm Engineering and Experiments, pp. 16–25. Society for Industrial and App. Math. SIAM (2003)
22. Gugelmann, L., Panagiotou, K., Peter, U.: Random hyperbolic graphs: degree sequence and clustering - (extended abstract). In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012. LNCS, vol. 7392, pp. 573–585. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31585-5_51
23. Hakimi, S.L.: On realizability of a set of integers as degrees of the vertices of a linear graph. I. J. Soc. Ind. App. Math. **10**(3), 496–506 (1962). https://doi.org/10.1137/0110037
24 SPP. Hamann, M., Meyer, U., Penschuck, M., Tran, H., Wagner, D.: I/O-efficient generation of massive graphs following the LFR benchmark. ACM J. Exp. Algorithmics **23**, 1-33 (2018). https://doi.org/10.1145/3230743
25 SPP. Hamann, M., Meyer, U., Penschuck, M., Wagner, D.: I/O-efficient generation of massive graphs following the LFR benchmark. In: ALENEX, pp. 58–72. SIAM (2017). https://doi.org/10.1137/1.9781611974768.5
26. Havel, V.: Poznámka o existenci konečných grafů. Časopis pro pěstování matematiky **080**(4), 477–480 (1955)
27 SPP. Hübschle-Schneider, L., Sanders, P.: Linear work generation of R-MAT graphs. Netw. Sci. **8**(4), 543–550 (2020). https://doi.org/10.1017/nws.2020.21
28. Lancichinetti, A., Fortunato, S.: Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. Phys. Rev. E **80**(1), 016118 (2009). https://doi.org/10.1103/physreve.80.016118
29 SPP. von Looz, M.: High-performance graph algorithms. Ph.D. thesis. KIT - Karlsruhe Institute of Technology (2018)
30 SPP. von Looz, M., Meyerhenke, H.: Querying probabilistic neighborhoods in spatial data sets efficiently. In: Mäkinen, V., Puglisi, S.J., Salmela, L. (eds.) IWOCA 2016. LNCS, vol. 9843, pp. 449–460. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44543-4_35
31 SPP. von Looz, M., Meyerhenke, H.: Updating dynamic random hyperbolic graphs in sublinear time. ACM J. Exp. Algorithmics **23**, 1–30 (2018). https://doi.org/10.1145/3195635

32 SPP. von Looz, M., Meyerhenke, H., Prutkin, R.: Generating random hyperbolic graphs in subquadratic time. In: Elbassioni, K., Makino, K. (eds.) ISAAC 2015. LNCS, vol. 9472, pp. 467–478. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48971-0_40

33 SPP. von Looz, M., Özdayi, M.S., Laue, S., Meyerhenke, H.: Generating massive complex networks with hyperbolic geometry faster in practice. In: HPEC, pp. 1–6. IEEE (2016). https://doi.org/10.1109/HPEC.2016.7761644

34. Mahdian, M., Xu, Y.: Stochastic kronecker graphs. In: Bonato, A., Chung, F.R.K. (eds.) WAW 2007. LNCS, vol. 4863, pp. 179–186. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77004-6_14

35. Maheshwari, A., Zeh, N.: A survey of techniques for designing I/O-efficient algorithms. In: Meyer, U., Sanders, P., Sibeyn, J. (eds.) Algorithms for Memory Hierarchies. LNCS, vol. 2625, pp. 36–61. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36574-5_3

36 SPP. Meyer, U., Penschuck, M.: Generating massive scale-free networks under resource constraints. In: ALENEX, pp. 39–52. SIAM (2016). https://doi.org/10.1137/1.9781611974317.4

37. Morton, G.M.: A comp. oriented geodetic data base and a new technique in file sequencing. Technical report. Int. Business Machines Company, New York (1966). https://domino.research.ibm.com/library/cyberdig.nsf/0/0dabf9473b9c86d48525779800566a39?OpenDocument

38. Murphy, R.C., Wheeler, K.B., Barrett, B.W., Ang, J.A.: Introducing the graph 500. Cray Users Group (CUG) **19**, 45–74 (2010)

39. Newman, M.E.J.: Networks: An Introduction. Oxford University Press, Oxford (2010). https://doi.org/10.1093/ACPROF:OSO/9780199206650.001.0001

40. Newman, M.E.J., Girvan, M.: Finding and evaluating community structure in networks. Phys. Rev. E **69**(026113), 1–16 (2004). http://link.aps.org/abstract/PRE/v69/e026113

41. Orenstein, J.A., Merrett, T.H.: A class of data structures for associative searching. In: PODS, pp. 181–190. ACM (1984). https://doi.org/10.1145/588011.588037

42 SPP. Penschuck, M.: Generating practical random hyperbolic graphs in near-linear time and with sub-linear memory. In: SEA, pp. 26:1–26:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.SEA.2017.26

43 SPP. Penschuck, M.: Scalable generation of random graphs. Ph.D. thesis. Goethe University Frankfurt (2020)

44 SPP. Penschuck, M., et al.: Recent advances in scalable network generation. CoRR abs/2003.00736 (2020)

45. Popper, K.: The Logic of Scientific Discovery. Hutchinson, London (1959)

46. Price, D.J.D.S.: Networks of scientific papers. Science **149**(3683), 510–515 (1965). http://www.jstor.org/stable/1716232

47 SPP. Sanders, P., Lamm, S., Hübschle-Schneider, L., Schrade, E., Dachsbacher, C.: Efficient parallel random sampling - vectorized, cache-efficient, and online. ACM Trans. Math. Softw. **44**(3), 29:1–29:14 (2018). https://doi.org/10.1145/3157734

48 SPP. Sanders, P., Schulz, C.: Scalable generation of scale-free graphs. Inf. Process. Lett. **116**(7), 489–491 (2016). https://doi.org/10.1016/j.ipl.2016.02.004

49 SPP. Schlauch, W.E., Zweig, K.A.: Influence of the null-model on motif detection. In: IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining ASONAM, pp. 514–519. Association for Computing Machinery ACM (2015). https://doi.org/10.1145/2808797.2809400

50. de Solla Price, D.J.: A general theory of bibliometric and other cumulative advantage processes. J. Am. Soc. Inf. Sci. **27**(5), 292–306 (1976). https://doi.org/10.1002/asi.4630270505

51 SPP. Staudt, C.L., Hamann, M., Gutfraind, A., Safro, I., Meyerhenke, H.: Generating realistic scaled complex networks. Appl. Netw. Sci. **2**(1), 1–29 (2017). https://doi.org/10.1007/s41109-017-0054-z

52. Strona, G., Nappo, D., Boccacci, F., Fattorini, S., San-Miguel-Ayanz, J.: A fast and unbiased procedure to randomize ecological binary matrices with fixed row and column totals. Nat. Commun. **5**(1), 1–9 (2014). https://doi.org/10.1038/ncomms5114

Increasing the Sampling Efficiency for the Link Assessment Problem

André Chinazzo(✉), Christian De Schryver, Katharina Zweig, and Norbert Wehn

TU Kaiserslautern, Kaiserslautern, Germany
{chinazzo,schryver,wehn}@eit.uni-kl.de, zweig@cs.uni-kl.de

Abstract. Complex graphs are at the heart of today's big data challenges like recommendation systems, customer behavior modeling, or incident detection systems. One reoccurring task in these fields is the extraction of network motifs, which are subgraphs that are reoccurring and statistically significant. To assess the statistical significance of their occurrence, the observed values in the real network need to be compared to their expected value in a random graph model.

In this chapter, we focus on the so-called Link Assessment (LA) problem, in particular for bipartite networks. Lacking closed-form solutions, we require stochastic Monte Carlo approaches that raise the challenge of finding appropriate metrics for quantifying the quality of results (QoR) together with suitable heuristics that stop the computation process if no further increase in quality is expected. We provide investigation results for three quality metrics and show that observing the right metrics reveals so-called *phase transitions* that can be used as a reliable basis for such heuristics. Finally, we propose a heuristic that has been evaluated with real-word datasets, providing a speedup of 15.4× over previous approaches.

Keywords: Link Assessment · Edge switching · Curveball · Random graphs

1 Introduction

The *data deluge* phenomenon is ever more present. We, as a society, generate and store far more data than what we can make use of right now [7]. Among the reasons for that are: 1. new approaches to acquire data, ranging from internet traffic recordings to high throughput DNA sequencing; and 2. the reduction in price per bit of data storage technologies, which motivates companies and researchers to be less selective about what data to be stored. These are by no means disadvantages over past methodologies, instead, they open new possibilities for data analysis that require methods that are more efficient and more robust against noise.

Complex network analysis is a tool-set of methods commonly used to extract information from large amounts of data, as long as the data can be meaningfully represented as a network. One popular method is the so-called *Link Assessment (LA)*, whose goal is to refine the data based on the principle of *structural similarity* (or *homophily*), i.e., entities that are alike tend to share a large proportion of their neighbors. Although the assumption of homophily is most common in social network analysis, mainly unipartite networks, it has been shown useful in a large range of contexts, including bipartite networks (such as a user rating / movie network) [24 SPP].

For unipartite networks, such as a protein-protein interaction database [12] or social networks [24 SPP], the LA may serve as a data-cleansing method, evaluating whether each existing link is likely to be a true positive and whether each non-existing link is likely to be a true-negative. This should not be confused with the link *prediction* problem, which is already well-researched [17], but poses a slightly different question: Given a snapshot of a network at time t, which of the yet unconnected node pairs are predicted to be connected at $t+1$?

For bipartite networks, such as genes that are associated with diseases [11] or products that are bought by costumers [10], the LA is a systematic way of projecting such networks to one of their sides [27]. This so-called *one-mode projection* transforms a bipartite network into a unipartite one by connecting the nodes on one of the sides based on their connections to the other side, while the nodes of the other side are discarded (see Sect. 2). Since most methods and tools for network analysis focus on general graphs, the one-mode projection of bipartite networks is a particularly useful pre-processing step to their analysis [27]. In this chapter, therefore, we focus on the LA for bipartite networks.

The LA is closely related to a vast body of research that includes the link prediction [17,20], recommendation systems [2], and node similarity in complex network analysis [16, 18]. Known approaches for such problems can be divided into *supervised* and *unsupervised learning*. Supervised learning approaches require a ground truth, i.e., a subset of the network whose links or labels are known to be correct. In general, these ground truths, or training sets, are manually annotated and therefore are often the bottleneck in the data-mining pipeline [26]. Moreover, the ground truths are often split into a training set and a test set, where the test set is used to estimate the quality of results (QoR). Unsupervised learning methods, on the other hand, require no ground truth, instead, they rely only on the structure (or other properties) of the data itself. Thus, in many cases, the QoR cannot be directly estimated. In conclusion, these methods should only be applied to specific types of data set for which their robustness has already been validated.

As stated earlier, the LA is an unsupervised method that assumes the relationships in the network to adhere to the notion of homophily, where alike nodes tend to have a larger common neighborhoods than one would expected from merely their degrees in a randomly constructed graph. In practice, the LA is based on Markov chain Monte Carlo (MCMC) methods that generate a large set of such random graphs. These MCMC methods are known to eventually converge, but their parameters are unknown. Whenever a ground truth is available, the QoR is assessed by, for e.g., the ratio of correctly identified pairs of alike nodes over the total number of pairs listed in the ground truth, i.e., the PPV_k (see Sect. 2.1). Else, one must ensure that the MCMC has converged.

In this chapter we summarize specific aspects for creating an LA problem solver, give insights into available metrics for measuring the QoR, and propose an appropriate heuristic that can speed up the run time by a factor of $15.4\times$ compared to a conservative approach.

2 Link Assessment Based on z*

Several node similarity measures have been proposed by different scientific communities, such as the *Jaccard index*, the *Pearson correlation coefficient*, or the *hypergeom* [12]. In [24 SPP], we have introduced a new similarity measure, the z* that has shown to be the most robust one across a range of datasets from protein-protein interactions to movie ratings to social network.

Given a bipartite graph $G((V_l,V_r),E)$ with vertices V_l and V_r and edges E, we define $coocc(u,v)$ as the number of co-occurring neighbors of nodes u and v. Most node similarity measures inherently depend on this quantity, but differ in how they are normalized based on the structure of the network. The similarity scores between nodes of the side-of-interest, say V_l, are the basis for the one-mode projection $G((V_l,V_r),E) \Rightarrow G'(V_l,E')$, where E' are edges between nodes in V_l. Some similarity measures use a simple factor based on properties of the two nodes, u and v (e.g., Jaccard index), while others are the result of a comparison to the expected value from a null-model (e.g., hypergeom).

The z* falls in the second category, as a combination of the p-value and the z-score statistics of the node-pairwise co-occurrences. Node pairs are ranked more similar if their p-value is smaller and ties are broken by their z-score [24 SPP]. Of key importance is the null-model used–the *fixed degree sequence model (FDSM)*. The FDSM is a random graph model that preserves the degree sequence of the original network while randomizing its nodes' interconnections, or edges. While it has been shown that the FDSM is a superior null-model than simpler graph models [13, 14, 27], closed-form expressions for the expected co-occurrences, $coocc_{FDSM}(u,v)$, are not known. These quantities are instead estimated by a random sampling procedure, known as a Markov chain Monte Carlo (MCMC) approach. Algorithm 1 describes the complete calculation of the z*.

2.1 Ground Truth and PPV$_k$

Throughout this chapter, we discuss a variety of results for the Link Assessment (LA) using as an example the Netflix Prize dataset[1]. By setting a threshold, the data are represented as a bipartite graph between users and movies, where an edge (u,v) means that user u liked (4 or 5 stars in the 1–5 scale) movie v. By finding significant co-occurrences between any two movies (v,w), a one-mode projection can be obtained [27]. The projection to the movies side was preferred because the users are anonymized, therefore it would be impossible to generate a ground truth of known similar users.

We quantify the quality of the LA by the positive predictive value (PPV_k) based on a ground truth dataset that contains only pairs of known non-random association, namely movie sequels like Star Wars and James Bond. The PPV_k is the fraction of correctly identified pairs from the ground truth in the set of the k highest ranked pairs of movies, where k is the number of pairs in the ground truth (see [3 SPP] for an example).

Building a ground truth for real datasets requires orthogonal information about the data (information that is not available for the LA method being tested) as well as an

[1] Available at https://www.kaggle.com/netflix-inc/netflix-prize-data.

Algorithm 1: The complete Link Assessment algorithm, calculating the similarity measure z*

```
Data: Graph $G((V_l,V_r);E)$ with vertices $V_l$ and $V_r$ and edges $E$, $V_l$ being the vertices of
      interest;
Result: A z*-score (p-value and z-score) for all pairs of vertices $(u,v) \in (V_l \times V_l)$;
1  Calculate $coocc(u,v)\ \forall\ (u,v) \in (V_l \times V_l)$; $G_0 := G$;
2  for $i := 1$ to $|samples|$ do
3      $G_i := G_{i-1}$;
4      Graph randomization:
5      for $|swaps|$ do
6          Choose two edges at random in $G_i$ and swap them, if no duplicate edge arises
           from the swap;
7      Coocc computation:
8      Calculate $coocc_i(u,v)\ \forall\ (u,v) \in (V_l \times V_l)$;
9  Calculate p-value and z-score, i.e., the z*:
10 p-value$(u,v) := (|\{i : coocc_i(u,v) > coocc(u,v)\ \forall\ i \in 1..|samples|)\}|\ \forall\ (u,v) \in (V_l \times V_l)$;
11 $coocc_{FDSM}(u,v) := \{coocc_i(u,v)\ \forall\ i \in 1..|samples|\}\ \forall\ (u,v) \in (V_l \times V_l)$;
12 z-score$(u,v) := \frac{mean(coocc_{FDSM}(u,v)) - coocc(u,v)}{stddev(coocc_{FDSM}(u,v))}\ \forall\ (u,v) \in (V_l \times V_l)$;
```

reliable method, such as assuming that movies within a sequel are non-randomly similar. Therefore, reliable ground truths are rare, limiting the range of input datasets for which the PPV_k can be measured.

Recently, however, we have discovered a systematic way of generating synthetic graphs for which the ground truth can be directly extracted, based on the benchmarks proposed in [14]. With that, we are able to conduct experiments for reliably comparing the efficiency and QoR of several LA approaches over an arbitrary range of input datasets. However, this work is still ongoing.

2.2 Random Graph Models

Network mofits are subgraphs whose occurrence in the observed data is statistically significant when compared to a random graph model (a null-model). The choice of such a null-model must be well-suited to test the investigator's hypothesis, and an inappropriate null-model can result in misinterpretation of the observed data [8]. The fixed degree sequence model (FDSM) is considered most appropriate for the identification of motifs, and in many cases, only simple graphs should be considered, i.e., no self-loops nor multi-edges. Unfortunately, closed-form expressions for the expected motif frequency over all possible simple graphs with a prescribed degree sequence are not yet known. Therefore, we commonly rely on a comparatively inefficient MCMC approach based on sequential mixing of the sampled graph states.

In [23 SPP] and [22 SPP], we have looked at different null-models, which can be more efficiently generated, as an approximation for the FDSM, as well as developed equations with the same intention. While some 3-node subgraph frequencies can be well approximated by simple equations, the case for the node-pairwise co-occurrences is more complex. For very regular degree sequences, i.e., all nodes have similar degrees, an equation based on the simple independence model is sufficient to estimate the individual node pairs co-occurrences. As the degree sequence becomes more skewed, the true values from the FDSM diverge from the approximation. Even a more intricate approximation for the individual co-occurrences [19, p. 441], whose sum almost matches the true value, becomes inaccurate for high-degree nodes. Since skewed degree sequences are abundant in real networks, such approximations cannot be widely used.

2.3 Co-occurrence Gradient in the FDSM

In another attempt to avoid the costly MCMC sampling approach for estimating the expected co-occurrences in the FDSM, we have analyzed the co-occurrence gradients throughout the Markov chains–a so-called *mean-field approach*, borrowed from statistical physics. In this approach, we first find the differential equation that describes the expected change, i.e., gradient, in co-occurrence after one single step in the graph mixing Markov chain. If the gradient is sufficient to describe the dynamics of the chain, a closed-form solution for the expected co-occurrences could be derived (see [1] for an example of a successful attempt), or at least an iterative, direct method that is not based on sampling.

In order to fully describe the dynamics of the mixing chains, the co-occurrences gradient, $\Delta\mathbf{coocc} = \Delta\mathbf{coocc}(\mathbf{coocc})$, must be a function of only the co-occurrence matrix, **coocc**. If additional parameters are needed, their dynamics must also be represented in differential equations. As it turned out, however, the co-occurrences gradient can only be found if the structure of the graph is taken into account, i.e., **coocc** is not sufficient. Table 1 exemplifies such insufficiency by showing that a centrosymmetric **coocc** matrix (middle) does not result in a equally centrosymmetric Δ**coocc** matrix (right).

Table 1. An example of a adjacency matrix (left) whose row-pairwise *co-occurrence (coocc)* matrix (middle) is not sufficient to calculate the expected *coocc* gradient (right). For the sake of clarity, the gradient values w.r.t. the edge switching chain (right) are shown without normalization by the number of possible swaps trials per step, $|E|^2 = 11^2$.

	c_1	c_2	c_3	c_4	c_5
r_1	0	0	0	1	1
r_2	0	0	1	0	1
r_3	0	1	1	1	0
r_4	1	0	0	1	0
r_5	0	0	0	1	1

	r_1	r_2	r_3	r_4	r_5
r_1	–	1	1	1	2
r_2	1	–	1	0	1
r_3	1	1	–	1	1
r_4	1	0	1	–	1
r_5	2	1	1	1	–

	r_1	r_2	r_3	r_4	r_5
r_1	–	−4	6	0	−16
r_2	−4	–	2	12	−4
r_3	6	2	–	−2	6
r_4	0	12	−2	–	0
r_5	−16	−4	6	0	–

In fact, the expected change in co-occurrence, a node pairwise relation, can only be given by the interaction between the neighborhoods of three nodes. This becomes clear once we realize that $coocc(i, j)$ can only be changed if the neighborhood of a third node k is modified since the degree sequences are fixed. In turn, the dynamics of the node

3-wise relations can only be described by 4-wise relations, and so on. Therefore, we conclude that a full description of the dynamics of the mixing chains w.r.t. the pairwise co-occurrences is not feasible.

3 The Benchmarking Problem

In general, comparing different system implementations is a non-trivial task. The reason is that plenty of parameters influence the final system behavior, such as the underlying system architecture, the selected algorithms, the chosen software implementation language, compilers, or communication and memory infrastructures. Besides, the performance of a system can heavily depend on the input data, in particular if adaptive ("*self-tuning*") methods are used. Thus, fairly comparing implementations requires an in-depth analysis of the relevant factors and the target application domains first.

In this context, we distinguish between the *application* or *problem* (the actual task to be carried out), the employed *model* or *algorithm* and its final *implementation* on a specific *architecture*. The latter three make up the final *system solution* that we are evaluating.

Let us look more closely at an example: We define the *application* or *problem* as "recommend movies to a client who has already watched several other movies", a generic task, e.g., in a video streaming service. The choice of an appropriate *algorithm* for this task is crucial for the overall system behavior: We can, e.g., select a graph-based approach as discussed in this chapter, statistical analysis, or machine learning based methods [15]. Each of those can be implemented in pure software on a generic computer architecture, in hardware, or in a hybrid hardware/software setting that combines programmable architectures such as central processing units (CPUs) with hardware accelerators. The selection of an appropriate underlying system *architecture* is strongly linked to the chosen algorithm, since there may be strong interactions between those two. Some algorithms are more friendly for being implemented in hardware or accelerators (in particular if they allow high parallel processing), while others may fit more to programmable (i.e., in general sequential or control-driven) architectures. Thus, fixing *algorithm* and *architecture* in the system design flow is an iterative and heavily interdependent process that requires a deep understanding of both domains and the target *application*, since the latter may impose additional restrictions or constraints on the other ones. In particular, low-level parameters such as selecting appropriate data structures for efficient memory accesses or custom data types with reduced-precision can lead to strong increases in performance and energy efficiency, but may also impact the QoR (see Chapter 4).

However, evaluating/comparing *systems* always requires well-defined *metrics*. In order to allow comparisons over architectural borders, those metrics need to be independent of the underlying system architecture and/or employed software. "Operations per second" for example is still a widespread metric in the high-performance computing (HPC) domain, but cannot be applied to systems that incorporate hardware accelerators (mainly data-flow architectures with hard-wired circuits) in which no "operations" exist. Thus, we propose application-level metrics that are not related to the selected algorithms or architectures. Examples are "run-time for a specific task", "consumed energy for a run", and "achieved QoR for a specific task".

In the above-mentioned example "recommend movies to a client who has already watched several other movies", we could, e.g., compare a software implementation running on a CPU-based cluster with a (hybrid) hardware-accelerated architecture. After deciding that we are going to implement a graph-based approach over other available options, we can still select the specific algorithm for the LA part (e.g., Edge Switching (ES) or Curveball (CB), see Sect. 4), the number of processing elements (PEs), the amount of hardware acceleration (if any), the memory hierarchy, communication infrastructure, the data structure (e.g., matrix vs. adjacency list), and the data types (e.g., floating-point precision) that impact both storage demands and required computational effort. It is obvious that a large number of degrees of freedom leads to an overwhelming amount of possible *system solutions* that all solve the same task, but with different characteristics.

While "run-time for a specific task" and "consumed energy for a run" can be measured or estimated with rather straight-forward approaches, determining the "achieved QoR for a specific task" is much harder to quantify. The reason is that in general multiple ways for measuring quality exist that must be investigated more specifically in order to determine which metric provides the most meaningful insights for the specified application.

In addition, stochastic parts of the selected algorithm (e.g., based on former system states, random numbers, or early stopping criteria/heuristics) may even lead to variations over different runs with the same input data. Thus, a robust quality metric not only needs to provide a meaningful quantitative statement of the achieved QoR but should also be determined in a way that minimizes stochastic impacts on the result. In the LA part of our example ("recommendation system"), we could, e.g., use the PPV_k [24 SPP] as a direct measure of the results or autocorrelation [6 SPP] or perturbation [25] as QoR measure for the mixing itself.

A generic approach for tackling these issues are *benchmark sets* that try to cover specific application areas with typical data points. Most of them consist of so-called *batteries* that combine multiple tests into larger task lists to minimize set up/initialization and read-out overhead and to reduce stochastic effects.

In order to stop a stochastic process when a sufficient QoR is achieved, we employ *heuristics* that perform online tracking of specific QoR measures together with desired target values (so-called *early stopping criteria*). Once the target is achieved, the processing is stopped. For the Link Assessment (LA) problem with the ES chain, we have analyzed how the PPV_k changes over the number of samples and swaps throughout the processing [4 SPP]. We are using two data sets, the Netflix competition data set and a medium-size MovieLens data set[2]. More detailed insights are given in Fig. 5.

Figure 4 and Fig. 5 clearly show that the PPV_k saturates abruptly when a specific number of samples or swaps is achieved (a so-called *phase transition*). From this moment on, further processing does not increase the QoR any more. Thus, we can stop when we detect the phase transition of the PPV_k and use this as an early stopping criterion for this task.

[2] The 100k MovieLens data set, available from http://grouplens.org/datasets/movielens/.

From this criterion, we can derive an appropriate heuristic that we incorporate into the final implementation. One crucial aspect for such a heuristic is its *stability*, i.e., it must be ensured that it works reliably for the allowed range of input data sets for a given application and that it stops the processing at the earliest possible time when the desired QoR is achieved. We present appropriate heuristics for the LA in Sect. 5.1.

4 Edge Switching vs. Curveball

Generating random samples from the fixed degree sequence model (FDSM) remains the most accurate method for estimating the expected co-occurrences between nodes, and therefore also for performing the Link Assessment (LA). Exact sampling schemes, where random graph samples are generated from scratch and exactly uniformly at random, were proposed but their computational complexity is $O(n^3)$ [9]. Most commonly, the random graphs are generated by sequentially mixing the original graph's edges, specifically using the Edge Switching (ES) Markov chain. Strona et al. [25] proposed a new algorithm, coined the Curveball (CB), which instead of switching a pair of edges, randomly trades the neighborhoods of two nodes. The CB was quickly proven to converge to the uniform distribution and adapted for different types of graphs [5] (see Chapter 2 for more details about the Curveball algorithm).

The mixing time of a Markov chain refers to the number of steps in the chain required to reach any possible state with equal probability[3], i.e., to disassociate the final, random state from the initial state [21]. While the true mixing time of neither Markov chain, the ES or the CB, is known, first empirical results suggested that the CB was a more efficient method of randomizing a graph [5,25]. These are based on the perturbation score and discussed in terms of the number of steps in the respective Markov chains. In practice, however, one CB step may take much longer than one ES step, so an actual runtime comparison between implementations is more meaningful.

In this section, we show the runtime comparison between two versions of the CB algorithm and an ES implementation. The Sorted-lists Curveball (SCB) iterates through two randomly selected nodes' neighborhoods (lists) in order to find, shuffle and re-assign the disjoint set of neighbors. Although finding the disjoint set is facilitated by keeping sorted lists, after the re-assignment they must be re-sorted in preparation for the next trade, so the overall complexity is $O(deg_{max} \times \log deg_{max})$ per trade, where deg_{max} is the maximum node degree of the network. The Hashed-lists Curveball avoids sorting the lists by creating a temporary hash-map of each neighborhood. The complexity of the HCB depends on the properties of the hash-map used, but in general is between $O(deg_{max})$ and $O(deg^2_{max})$ per trade. Finally, the ES uses two redundant data structures to accomplish a complexity of $O(1)$ per swap: the adjacency lists to randomly pick two existing edges; and the adjacency matrix to check whether they can be swapped.

4.1 Perturbation Score

The perturbation score [25] is the number of different entries between the adjacency matrices of the original graph and the random sample. Since each step in the edge

[3] Within an arbitrary error margin.

switching chain can only swap two edges, at most four entries of the adjacency matrix are modified in each step. The perturbation score, although maybe not a true estimator for the total mixing time, is a direct measure of the distance of the shortest path between two states of the ES chain.

The first comparison between the randomization algorithms was conducted using the Netflix Prize dataset. Figure 1 shows the relative perturbation[4] vs. the runtime for a curveball (the SCB) and the ES implementation (shown are averages of at least 10 repetitions). The bottom-most subplot refers to the complete dataset, while the first two refer to random subsets of 10000 and 100000 users, respectively. Since the curveball algorithm is sensitive to the number of adjacency lists (=number of columns in the adjacency matrix), both Movies x Users and Users x Movies representations are simulated. In this analysis, irrespective of the amount of data and number of adjacency lists, the ES implementation is at least $2\times$ faster than SCB.

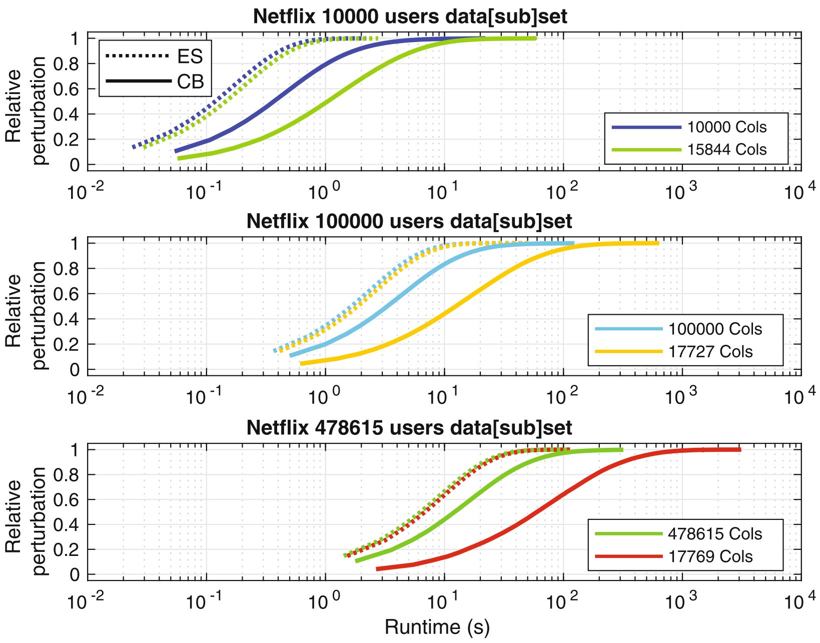


Fig. 1. Relative perturbation achieved by ES and SCB vs runtime for different subsets of the Netflix Prize dataset. Machine: Intel(R) Xeon(R) E5 2640v3 @ 2.60 GHz.

Besides the surprising results showing that the ES implementation using both the adjacency lists and matrix is faster than the CB based on sorted adjacency lists (further discussed in Sect. 4.2), another interesting effect can be seen in Fig. 1: Mixing the

[4] We define the relative perturbation as the perturbation score normalized by its maximum value among all mixing chains and repetition runs.

Users side of the bipartite network is always faster than mixing the Movies side, irrespective of the number of users and movies. An explanation for that may be in the degree distribution of the two types of nodes, users and movies, do not show the same shape. Figure 2 shows the degree distribution densities for users and movies nodes for a subset of 100000 randomly selected users of the Netflix Prize dataset. While relatively more movies have very low or very high degrees, user degrees concentrate in the middle, a trend that holds for any subset of randomly selected users (not shown). Given that a curveball trade consists of shuffling the disjoint neighborhoods of two randomly selected nodes, a higher mixing efficiency is expected when the two nodes have similar degrees. Therefore, we conclude that, when mixing bipartite graphs using the curveball Markov chain, it is advantageous to perform the trades between nodes from the least skewed degree distribution side, at least with respect to the perturbation score vs. the runtime. Note that the ratio between the number of nodes of the two sides does not play a major role w.r.t. the runtime (see Fig. 1), contrary to the belief of the original curveball algorithm inventors [25].

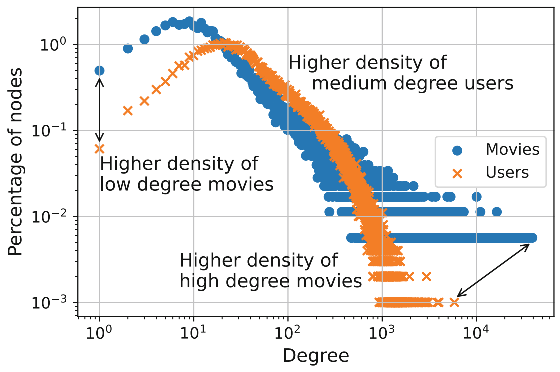


Fig. 2. The degree distribution densities of users and movies for a subset of the Netflix Prize dataset containing good ratings from 100000 random users. Users' degrees are more concentrated in the middle while movies' degrees have a more skewed distribution, with many very low and many very high degree movies.

4.2 Runtime Comparison with NetworKit

In [6 SPP], an I/O-efficient implementation of the curveball trades for simple, unipartite graphs is proposed. Its key feature is the introduction of a trade sequence that is lexicographically sorted before the curveball trades are performed, resulting in a more efficient memory access (see Chapter 2 or [6 SPP] for more details). In cooperation with the Group of Algorithm Engineering from the Goethe University Frankfurt, we compare our ES, SCB, and HCB implementations to a bipartite version of their curveball.

Figure 3 shows the runtime comparison results between our ES, SCB, and HCB and the NetworKit CB [6 SPP] implementations. Each mixing chain executes 20 super

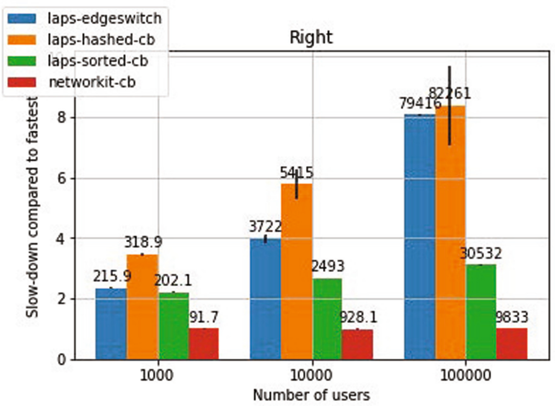


Fig. 3. Runtime comparison of the ES and three curveball implementations. The numbers over the colored bars indicate the average runtime in milliseconds for 10 super steps (see text). The black, thin bars indicate the standard deviation of 10 runs. NetworKit CB is consistently faster (at least 2x faster) and scales better than our ES, HCB, and SCB implementations. Machine: Intel(R) Xeon(R) CPU E5-2630v3 @ 2.40 GHz. (Color figure online)

steps, i.e., $10 \times \#Users$ for the curveball and $10 \times \#Edges$[5] for the edge switching, which considers, in expectation, 20 times the state of each edge, which is when both ES and CB converge in quality according to the autocorrelation thinning factor [6 SPP].

Figure 3 shows that NetworKit CB is at least $2\times$ faster than our CB implementations, as well as faster than the ES. From Fig. 1 we see that our ES is between $1.5\times$ to $2.5\times$ faster than the SCB, which in turn is between $2\times$ and $3\times$ slower than NetworKit CB. Therefore it is safe to assume that there is a CB implementation that is at least as fast as our ES even according to the perturbation[6]. Furthermore, it becomes evident from Fig. 3 that the CB–both our SCB and NetworKit's–scales consistently with the size of the graph, while the ES appears to have higher factors.

With these results, we conclude that the Curveball algorithm can indeed be efficiently implemented in software, and scales better than the ES. However, it also became clear that there can be huge differences in results interpretation depending on the quality metric being regarded (perturbation or auto-correlation), and it is not clear which is the most relevant.

5 Phase Transition and Heuristics

Instead of a steady increase in the quality of the LA with the underlying MCMC parameters, mainly the mixing length and number of samples, we actually see a flat low

[5] Notice the factor $0.5\times$ between the super step definition in [6 SPP] and here. This is due to the representation of unipartite, undirected graphs [6 SPP] requiring duplicated edges, one in each direction, while bipartite graphs do not.

[6] Unfortunately, a direct comparison w.r.t. the perturbation turned out to be difficult because of incompatibilities between the structures of the source codes.

quality followed by a sudden and steep increase. This phase transition-like behavior was first reported in [4 SPP], further discussed in [3 SPP], and is shown in Fig. 4. The LA quality, measured by the PPV_k, is plotted against the number of samples, the main parameter w.r.t. the total runtime of the method.

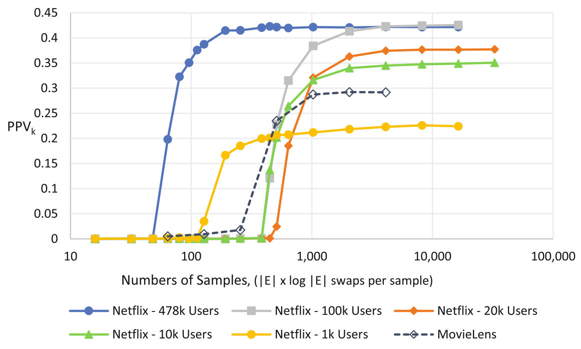


Fig. 4. Link assessment quality (PPV_V) over number of samples. For a wide variety of data sets, narrow phase transitions are present. Similar phase transitions are also seen for the number of edge swaps (see Fig. 7 in [3 SPP]). (Color figure online)

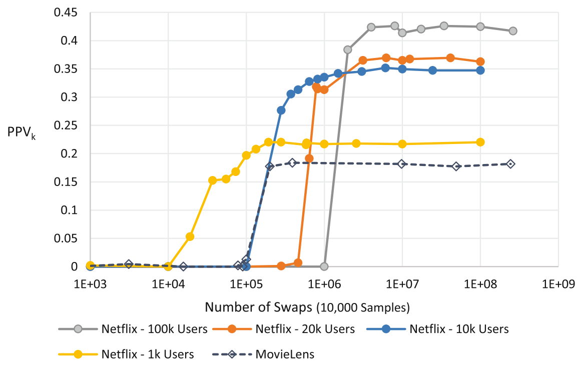


Fig. 5. Quality over number of swaps. For a wide variety of data sets, narrow phase transitions are present, similarly to what can be observed varying the number of samples. (see Fig. 2 in [4 SPP]). (Color figure online)

The complete Netflix dataset (blue), e.g., requires 384 samples to reach a PPV_k of 0.4206 ± 0.0019, while 16,384 samples only increase this value to 0.4217 ± 0.0012. While, on one hand, it indicates that a low number of samples is required, the steep transition also cautions us against taking too few samples, since 64 samples instead of 384 would result in roughly half the quality (0.20 ± 0.03) and 48 samples (0.001 ± 0.001) is no better than a random guess. Therefore, this is not mainly a trade-off problem, where more resources (samples) bring better quality, but rather a threshold problem, at which the quality transitions steeply from its minimum to its maximum. Finding this threshold can be done via online heuristics, as is discussed in Sect. 5.1.

5.1 Heuristics for MCMC Parameters

Continuing the sampling procedure after the maximum LA quality is reached results in increased runtime for no benefit. Therefore, being able to reliably assess when maximum quality has been reached can represent great speedups for the complete link assessment method. For example, compared to the commonly used 10,000 samples, we see in Fig. 4 that, for the complete Netflix dataset, the LA reaches maximum quality at around 384 samples, which would represent a speedup of $> 25\times$.

Figure 4 also shows that the tipping point depends on the dataset. However, we were not able to find correlations between the input dataset and the required number of samples, and so an analytic formula could not be derived.

Even though the mechanism that causes the phase transitions is not yet completely understood, we know that it must be related to the stability of the ranking of the most significant pair of nodes. We have tested multiple methods to evaluate the stability of the final result by comparing it with the previous one [4 SPP, 3 SPP]. A good estimator for the final quality should also present a phase transition-like behavior, as does the quality itself. In [3 SPP] we described in detail our successive attempts to find a reliable heuristic, going from the number of matching pairs at the very top of the ranking, to more sophisticated correlations methods, and finally the internal PPV_k method. The internal PPV_k method consists of creating an internal ground truth based on the top ranked node pairs at each iteration and using it to calculate the PPV_k after a new (group of) sample(s) is drawn. This measure turned out to be stable and well correlated to the observed phase transition of the actual quality, based on the real ground truth.

A rather similar heuristic, in the sense that it also presents a phase transition-like behavior, was devised for assessing the required amount of mixing (w.r.t. the number of edge swaps) necessary [3 SPP]. It is based on the fact that the average $coocc_{FDSM}(a,b)$ of two nodes a, b only depends on their degrees. Therefore, we expect to see a converging behavior of multiple node pairs that have the same degree pair if the amount of mixing is sufficient (see Fig. 10 in [3 SPP]).

Table 2 summarizes the speedups achieved by applying each and the two heuristics for the number of swaps and samples presented in [4 SPP, 3 SPP]. In all cases, the heuristics accelerate the overall LA (all overheads are accounted for) when compared to the safe number of swaps ($|E| \times \log |E|$) and samples (10000) without any significant degradation of its quality. The largest graph seems to benefit the most from either and both heuristics. For the smallest graph, the Movielens data, the overhead of the swaps heuristic becomes significant, shadowing the increased randomization speed during the actual sampling. Also, it still requires around one-third of the 10000 samples for the internal PPV_k to converge.

Notice that we can clearly see an interdependence between the number of swaps and the number of samples. For example, using a more conservative set of parameters for the swaps heuristic results in 8×10^7 edge swaps and 454 samples for the complete Netflix data, almost ten times more swaps but 4 times fewer samples as the less conservative alternative. This is expected since the relevance (or entropy) of each new sample is

Table 2. Runtime and quality comparisons of the LA with and without heuristics [3 SPP]. The swaps heuristic has three parameters: the convergence threshold, θ_{min}; the number of groups of node pairs with the same degree pair, N_g; and the number of node pairs per group, N_p. The samples heuristic also takes three parameters: the internal PPV_k threshold, α; the size of the internal ground truth, k'; and the number of samples between evaluations, $samples_{step}$.

Data set	Samples	Edge swaps	Runtime[a]	PPV
State-of-the-art				
Netflix, 487k users	10,000	10^9	20 h	0.422
Netflix, 100k users	10,000	2.6×10^8	5.5 h	0.425
MoviesLens	10,000	1.5×10^7	877 s	0.290
Swap heuristic:				
Netflix, 478k users[c]	10,000	6.3×10^6 (**160x**)	0.2 + 6.8 h (2.9x)	0.418 (−0.9%)
Netflix, 478k users[d]	10,000	8.0×10^7 (**13x**)	0.8 + 9.6 h (1.9x)	0.424 (+0.1%)
Netflix, 100k users[b]	10,000	6.8×10^7 (**4x**)	0.2 + 3.4 h (1.5x)	0.424 (−0.2%)
MovieLens[b]	10,000	1.5×10^6 (**10x**)	24 + 554 s (1.5x)	0.290 (0.0%)
Sample heuristic[e]:				
Netflix, 487k users	640	10^9	1.42 h (**14x**)	0.418 (−0.9%)
Netflix, 100k users	2,944	2.6×10^8	1.66 h (**3.3x**)	0.419 (−1.3%)
MovieLens	3,456	1.5×10^7	326 s (**2.7x**)	0.291 (+1.0%)
Combined heuristic[e]:				
Netflix, 487k users[c]	1,664	6.3×10^6	0.2+1.3 h (**13.3x**)	0.420 (−0.5%)
Netflix, 487k users[d]	454.4	8.0×10^7	0.8+0.5 h (**15.4x**)	0.419 (−0.7%)
Netflix, 100k users[b]	3,072	6.8×10^7	0.2+1.2 h (**3.9x**)	0.420 (−1.1%)
MoviesLens[b]	3,949	1.5×10^6	24+252 s (**3.2x**)	0.290 (0.0%)

[a]When appropriate, reported as (Swap heuristic runtime) + (LA core runtime)
[b]Swap heuristic parameters: $N_p = 4$; $N_g = 24$; $\theta_{min} = 0.01$
[c]Swap heuristic parameters: $N_p = 36$; $N_g = 36$; $\theta_{min} = 0.05$
[d]Swap heuristic parameters: $N_p = 36$; $N_g = 36$; $\theta_{min} = 0.005$
[e]Sample heuristic parameters: $samples_{step} = 128$; $k' = 0.2\% \, |V_R|^2$; $\alpha = 0.95$.

directly related to the amount of mixing between samples - the more independent the samples, the more information each of them adds to the poll. In this case, the more conservative choice is slightly faster.

The trade-off between the amount of mixing between samples and the required total number of samples provides an effective measure of the quality of each random sample. As the quality measure is independent of the mixing chains, we have used it to compare the effectiveness of different chains and their runtimes. An online heuristic that optimizes this trade-off is also under investigation.

5.2 Phase Transitions as Mixing Quality Estimation

Earlier results from the implementation of heuristics to find appropriate parameters for the MCMC sampling showed that there exists a trade-off between the amount of mixing

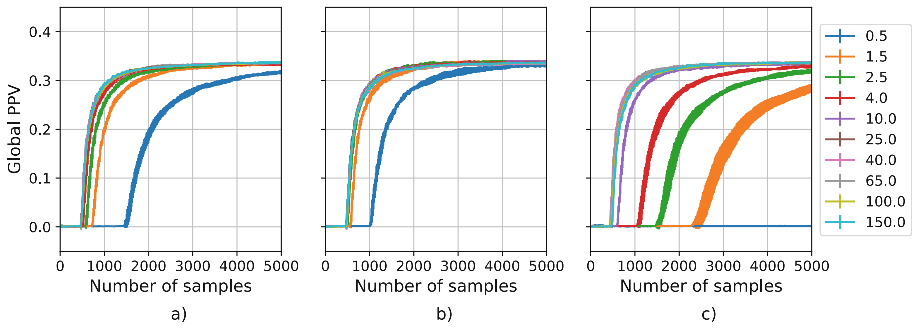


Fig. 6. The phase transitions in the link assessment quality of the Netflix 10k dataset for increasing mixing lengths of a) Edge Switching, b) Curveball in Movies, and c) Curveball in Users. The vertical line-width represent the standard deviation of 10 independent simulations. Mixing lengths are given in super steps, where 1 super step (Sect. 4.2) is a) $|E|/2$, b) $|M|/2$, and c) $|U|/2$.

and the number of samples required for convergence (Table 2). This insight led us to investigate how the PPV_k phase transitions behave when we vary the mixing length of the chains.

Figure 6 shows the LA quality phase transitions w.r.t. the number of samples for increasing mixing chain lengths and three different mixing chains. As expected, the phase transitions are shifted to the right (more samples are required) as the mixing length decreases. If the mixing length is sufficient, however, increasing it further may not significantly change the result. This behavior can be explained by the correlation (or level of independence) between consecutive samples, which is expected to decrease exponentially with the amount of mixing. If enough mixing is performed, the sampling procedure became virtually as efficient as it can be, as if each sample was drawn uniformly at random. If the correlation between each consecutive sample is measurable, the sampling efficiency is degraded, therefore the phase transitions both start later and become less steep.

Figure 6 also shows that there can be a great difference in the efficiency of the Curveball algorithm whether we choose to mix (b) movies or (c) users, even if the mixing lengths are normalized w.r.t. the adjacency matrix dimensions. Similar behavior was already observed in Fig. 1, where mixing the users' neighbors was faster in reaching the maximum perturbation. Surprisingly, however, mixing the movies side instead of users is more effective when the LA quality is regarded. To better expose this controversy, we can find the number of samples required to reach a certain PPV_k threshold, say 0.3 in this case. If the perturbation was a good predictor of the sampling efficiency, we would expect the number of samples to be the lowest only when the maximum perturbation is reached.

Figure 7 shows the perturbation and the number of samples required to reach the threshold PPV_k of 0.3 versus the amount of mixing. For the edge switching (ES) and the Curveball in users (CB_C), there seems to exist a strong (inverse) correlation between the perturbation and the number of samples. When the Curveball mixes the movies neighborhoods (CB_R), however, the perturbation is far too conservative. While the

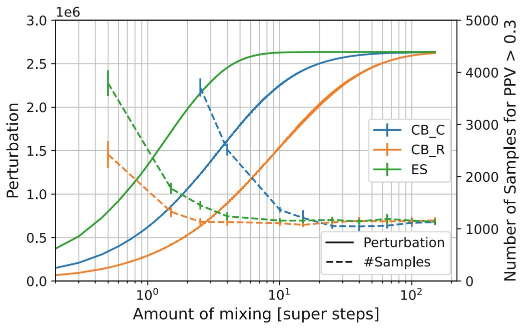


Fig. 7. The perturbation (left axis) and the number of samples until the PPV_k reaches 0.3 (right axis) vs the mixing length for the Netflix 10k dataset. The error bars represent the standard deviation of 10 independent simulations. Mixing chains: Curveball in Users (CB_C); Curveball in Movies (CB_R); Edge Switching (ES).

minimum number of samples of around 1200 is reached at 2.5 super steps, only at 150 super steps does the perturbation reaches its maximum value - an overdo of 60×.

6 Summary and Conclusion

In this chapter, we summarize strategies for increasing the sampling efficiency for the Link Assessment (LA) problem. Although we provide specific results for the latter one, our approach is generic and can be applied to related applications. Since in many cases no closed-form solutions exist, stochastic Markov chain Monte Carlo (MCMC) need to be employed. However, their main drawbacks are the high computational demand and the lack of reproducibility of exact numerical results due to their stochastic components. In addition, different algorithms may be used for generating the MCMC samples (edge switching vs. curveball), and their performance on different compute architectures can vary strongly. In many cases, it is not clear which algorithm is the best one for a specific data set on a specific architecture with respect to performance or quality.

Thus, we highlight the importance of application-level benchmark sets together with application-level, numerical measures for the quality of results (QoR) of specific runs. Such benchmarks allow a fair and quantitative comparison of non-architecturally linked metrics such as “energy per run”, “time per run”, or “quality of the results”. In addition, for many real-world data sets, we do not have any kind of ground truth that could serve as a test oracle when evaluating the achieved quality. In order to overcome this issue, we propose to construct meaningful benchmark batteries (together with ground truths) for specific application domains artificially that cover the main tasks and the corner cases equally.

Furthermore, we consider the PPV_k being a good quality measure for the LA problem. However, since it is strongly linked to the ground truth not available in many cases, we have investigated alternative metrics such as autocorrelation or perturbation. Those also allow the construction of unsupervised systems that are able to determine a “sufficient” QoR on their own. We clearly observe correlations between the latter ones and the PPV_k in Fig. 7, but a comprehensive analysis of any formal causalities between them is currently ongoing.

Finally, we propose a working heuristic for an LA solver based on edge switching that exploits observable phase transitions of the PPV_k as an early stopping criterion for the MCMC process. This heuristic provides speedups of up to 15.4× compared to solvers that use conservative approaches.

References

1. Barabási, A.L., Albert, R., Jeong, H.: Mean-field theory for scale-free random networks. Physica A: Stat. Mech. Appl. **272**(1), 173–187 (1999). https://doi.org/10.1016/S0378-4371(99)00291-5
2. Bobadilla, J., Ortega, F., Hernando, A., Gutiérrez, A.: Recommender systems survey. Knowl. Based Syst. **46**, 109–132 (2013). https://doi.org/10.1016/j.knosys.2013.03.012

3 SPP. Brugger, C., et al.: Increasing sampling efficiency for the fixed degree sequence model with phase transitions. Soc. Netw. Anal. Min. **6**(1), 1–14 (2016). https://doi.org/10.1007/s13278-016-0407-0

4 SPP. Brugger, C., et al.: Exploiting phase transitions for the efficient sampling of the fixed degree sequence model. In: ASONAM, pp. 308–313. ACM (2015). https://doi.org/10.1145/2808797.2809388

5. Carstens, C.J., Berger, A., Strona, G.: Curveball: a new generation of sampling algorithms for graphs with fixed degree sequence. CoRR abs/1609.05137 (2016). https://doi.org/10.1016/j.mex.2018.06.018

6 SPP. Carstens, C.J., Hamann, M., Meyer, U., Penschuck, M., Tran, H., Wagner, D.: Parallel and I/O-efficient randomisation of massive networks using global curveball trades. In: ESA, pp. 11:1–11:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.ESA.2018.11

7. Duranton, M., et al.: HiPEAC vision 2019. In: European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC) (2019)
8. Fosdick, B.K., Larremore, D.B., Nishimura, J., Ugander, J.: Configuring random graph models with fixed degree sequences. SIAM Rev. **60**(2), 315–355 (2018). https://doi.org/10.1137/16M1087175
9. Genio, C.I.D., Kim, H., Toroczkai, Z., Bassler, K.E.: Efficient and exact sampling of simple graphs with given arbitrary degree sequence. CoRR abs/1002.2975 (2010)
10. Gionis, A., Mannila, H., Mielikäinen, T., Tsaparas, P.: Assessing data mining results via swap randomization. ACM Trans. Knowl. Discov. Data **1**(3), 14 (2007). https://doi.org/10.1145/1297332.1297338
11. Goh, K.I., Cusick, M.E., Valle, D., Childs, B., Vidal, M., Barabási, A.L.: The human disease network. Proc. Natl. Acad. Sci. **104**(21), 8685–8690 (2007). https://doi.org/10.1073/pnas.0701361104
12. Goldberg, D.S., Roth, F.P.: Assessing experimentally derived interactions in a small world. Proc. Natl. Acad. Sci. **100**(8), 4372–4376 (2003). https://doi.org/10.1073/pnas.0735871100
13. Gotelli, N.J.: Null model analysis of species co-occurrence patterns. Ecology **81**(9), 2606–2621 (2000). https://doi.org/10.1890/0012-9658(2000)081[2606:NMAOSC]2.0.CO;2
14. Gotelli, N.J., Ulrich, W.: The empirical Bayes approach as a tool to identify non-random species associations. Oecologia **162**(2), 463–477 (2010). https://doi.org/10.1007/s00442-009-1474-y

15. Isinkaye, F., Folajimi, Y., Ojokoh, B.: Recommendation systems: principles, methods and evaluation. Egypt. Inform. J. **16**(3), 261–273 (2015). https://doi.org/10.1016/j.eij.2015.06.005
16. Leicht, E.A., Holme, P., Newman, M.E.J.: Vertex similarity in networks. Phys. Rev. E **73**, 026120 (2006). https://doi.org/10.1103/PhysRevE.73.026120
17. Lü, L., Zhou, T.: Link prediction in complex networks: a survey. Physica A: Stat. Mech. Appl. **390**(6), 1150–1170 (2011). https://doi.org/10.1016/j.physa.2010.11.027
18. Newman, M.: Networks. OUP, Oxford (2018)
19. Newman, M.E.J.: Networks: An Introduction. Oxford University Press, Oxford (2010). https://doi.org/10.1093/ACPROF:OSO/9780199206650.001.0001
20. Rapti, A., Tsolis, D., Sioutas, S., Tsakalidis, A.K.: A survey: mining linked cultural heritage data. In: EANN Workshops, pp. 24:1–24:6. ACM (2015). https://doi.org/10.1145/2797143.2797172
21. Rechner, S., Berger, A.: Marathon: an open source software library for the analysis of Markov-chain Monte Carlo algorithms. CoRR abs/1508.04740 (2015). https://doi.org/10.1371/journal.pone.0147935
22 SPP. Schlauch, W.E., Horvát, E.Á., Zweig, K.A.: Different flavors of randomness: comparing random graph models with fixed degree sequences. Soc. Netw. Anal. Min. **5**(1), 1–14 (2015). https://doi.org/10.1007/s13278-015-0267-z
23 SPP. Schlauch, W.E., Zweig, K.A.: Influence of the null-model on motif detection. In: ASONAM, pp. 514–519. ACM (2015). https://doi.org/10.1145/2808797.2809400
24 SPP. Spitz, A., Gimmler, A., Stoeck, T., Zweig, K.A., Horvát, E.: Assessing low-intensity relationships in complex networks. PLoS ONE **11**(4), 1–17 (2016). https://doi.org/10.1371/journal.pone.0152536
25. Strona, G., Nappo, D., Boccacci, F., Fattorini, S., San-Miguel-Ayanz, J.: A fast and unbiased procedure to randomize ecological binary matrices with fixed row and column totals. Nat. Commun. **5**(1), 4114 (2014). https://doi.org/10.1038/ncomms5114
26. Sun, C., Shrivastava, A., Singh, S., Gupta, A.: Revisiting unreasonable effectiveness of data in deep learning era. In: ICCV, pp. 843–852. IEEE Computer Society (2017). https://doi.org/10.1109/ICCV.2017.97
27. Zweig, K.A., Kaufmann, M.: A systematic approach to the one-mode projection of bipartite graphs. Soc. Netw. Anal. Min. **1**(3), 187–218 (2011). https://doi.org/10.1007/s13278-011-0021-0

A Custom Hardware Architecture for the Link Assessment Problem

André Chinazzo(✉), Christian De Schryver, Katharina Zweig, and Norbert Wehn

TU Kaiserslautern, Kaiserslautern, Germany
{chinazzo,schryver,wehn}@eit.uni-kl.de, zweig@cs.uni-kl.de

Abstract. Heterogeneous accelerator enhanced computing architectures are a common solution in embedded computing, mainly due to the constraints in energy and power efficiency. Such accelerator enhanced systems dispatch data- and computing-intensive tasks to specialized, optimized and thus efficient hardware units, leaving most control flow tasks for the more generic but less efficient central processing units (CPUs). Nowadays, also high-performance computing (HPC) systems are becoming more heterogeneous by incorporating accelerators into the computing nodes.

In this chapter, we introduce the concept of heterogeneous computing and present the design of a hardware accelerator for solving the Link Assessment (LA) problem, in introduced Chapter 3. The hardware accelerator integrates its main dedicated processing units with a customized cache design and light-weight data path. We provide detailed area, energy, and timing results for a 28 nm application specific integrated circuit (ASIC) process and DDR3 memory devices. Compared to an CPU-based cluster, our proposed solution uses 38x less memory and is 1030x more energy efficient for processing a users-movies dataset with half a million edges.

Keywords: Link assessment · Application specific · Custom hardware · DRAM

1 Introduction

Nowadays, we live in the era of the so-called *data deluge*, i.e., the increase in produced data supersedes the progress in the available compute performance. This poses heavy challenges on data-centric (statistical) methods, algorithms, and compute systems [18]. Among others, selecting the appropriate data structures, heterogeneity, and parallelization schemes are crucial for achieving high computing performances with low energy demands. For example central processing unit (CPU)-based systems can only access data stored in memory as complete words (cache lines) and work with fixed data types. In contrast, dedicated hardware accelerators allow custom bit widths and data types. This can not only save energy due to avoiding unnecessary data transfers and operations but also allowing direct bit-wise operations like, e.g., accessing one-bit-column entries in a matrix.

In general, standard computing architectures based on CPUs and graphics processor units (GPUs) are moving data around heavily. However, in modern technologies, data transfers and storage in general consume much more power than the actual computing [5]. In particular, accessing (off-chip) dynamic random-access memory (DRAM) is a very time- and energy-consuming task. This leads to the concept of the so-called *data-driven* or *dataflow computing*, e.g., employed in the Google TensorFlow architecture [5]. Such architectures focus on the data stream and manipulate data on-the-fly, avoiding unnecessary storage and data transfers.

In addition, in data centers, servers alone only consume around one-third of the total power, while the rest is required for cooling, communication, storage, and building supply [8]. Seen from a different perspective, the maximum available power budget of a system (or a data center) is a hard limit for the available computing power. The latter can only be increased by installing compute systems with a higher power efficiency (e.g., incorporating special hardware accelerators, for instance with a dataflow architecture). Thus, reducing the power demand of the compute servers in combination with the smart reduction of inter-server communication can lead to a total of 2-3x power savings in the data center itself.

Modern system on chips (SoCs) in the mobile, embedded, and Internet-of-Things (IoT) domain are heavily heterogeneous systems with plenty of custom components for dedicated purposes such as audio decoding, video en- and decoding, radio transmission, or sensor data pre-processing in a mobile phone. In particular for mobile devices, there are hard limits for both energy (battery capacity) and power (maximum heat dissipation). However, over the last decades we see more and more heterogeneity also in the data centers [1,5]. Examples are general purpose graphics processor units (GPGPUs), the Intel Xeon Phi accelerator cards, or the field programmable gate array (FPGA)-based Amazon EC2 F1 instances released in 2017[1]. One of the major reason is the so-called *Dark Silicon* phenomenon: In modern chip technologies, only a small amount of transistors can be active at a time in order to avoid overheating (and thus destruction) of the device [7]. This also poses a heavy challenge for the classical multi-core approach - more cores of the same type do not provide more computation power if they cannot be powered up all at the same time.

Nevertheless, end-users are not at all interested in the underlying technology of the *services* they use. Nowadays, most services are distributed over an information technology (IT)-infrastructure from IoT nodes, mobiles, edge servers, and data centers [13]. Thus, the overall application is partitioned and disseminated on various parts of the IT-infrastructure, all with probably different computing architectures and characteristics. As an example, consider a real-time navigation service from Google or Apple: The Global Positioning System (GPS) coordinates collected by (maybe external) GPS receivers are sent to the SoC of the mobile that acts as a human-machine interface (HMI), displaying the route. However, the route itself is calculated in a data center of the service provider. In addition, GPS data from other service users is employed for estimating traveling times and traffic jams, and incorporated in the route calculation.

[1] See https://aws.amazon.com/ec2/instance-types/f1/. Last accessed on 24/11/2022.

In this chapter, we give an overview of hardware-assisted compute systems for applications based on the *Link Assessment (LA)* algorithm. The LA algorithm can be used to clean up large network data sets with noisy data. It assesses the structural similarities between the nodes, and thus differentiates meaningful relationships between nodes from noisy ones [19 SPP]. The LA algorithm as presented in Chapter 3 can be employed on a large scale of applications, e.g., recommendation systems, protein-protein interaction analyses in biology, or business analytics and marketing [3 SPP].

In Sect. 2 we give a short overview about the fundamentals of hardware (HW) and hardware/software (HW/SW) design both for custom application specific integrated circuit (ASIC) and FPGA architectures. Section 3 provides detailed insights in our proposed HW architecture for the Link Assessment (LA) algorithm. Performance data and comparisons are given in Sect. 4. Section 5 concludes this chapter.

2 Basics of Hardware and Systems Design

Custom, dedicated hardware compute architectures are substantially different from standard programmable architectures such as CPUs or GPUs. They are tailored for a specific task, avoiding all unnecessary overhead in storing/moving data, for control architectures, and over-precision data types. This increases both compute performance and power/energy efficiency, at the cost of low to zero flexibility after design. In contrast to a program written for CPUs, hardware architectures, in general, do not receive and execute instructions. Instead, their behavior is encoded in the circuit itself.

Hardware accelerators are electrical (abstracted: digital) circuits that focus on data manipulation. They can be realized in three ways:

- As circuits with various discrete components on a printed circuit board (PCB),
- As a fixed geometry on silicon (a so-called *application specific integrated circuit (ASIC))*), or
- On an underlying configurable hardware architecture such as a *programmable logic device (PLD)*, in particular an *field programmable gate array (FPGA).*

Nowadays, most systems are realized on a so-called *system on chip (SoC)*. In contrast to discrete circuits realized on PCBs, a SoC combines most components on a single piece of silicon. For that purpose, various *processing elements (PEs)* are attached to a communication infrastructure (a bus or a *network on chip (NoC))*. In addition, external input/output (I/O) interfaces are provided for receiving from and sending data to the outside world. An example for such a SoC structure is given in Fig. 1.

In general, not all PEs are developed by the system designer (team) on their own. Instead, many component architectures are available for purchasing as so-called *intellectual property (IP)*, i.e., as hardware geometry or as design data given in a hardware description language (HDL) or a logical netlist. They mostly ship with an equivalent software model that can be used for behavioral analysis, testing, and debugging purposes. IP cores can somehow be compared to software libraries in programming since they offer predefined functionalities that can be incorporated into the overall systems. However, most IP cores are closed-source and only available on a commercial basis. In contrast to software projects, open-source hardware platforms such as *opencores.org* are very limited, both from their available contents and their technology.

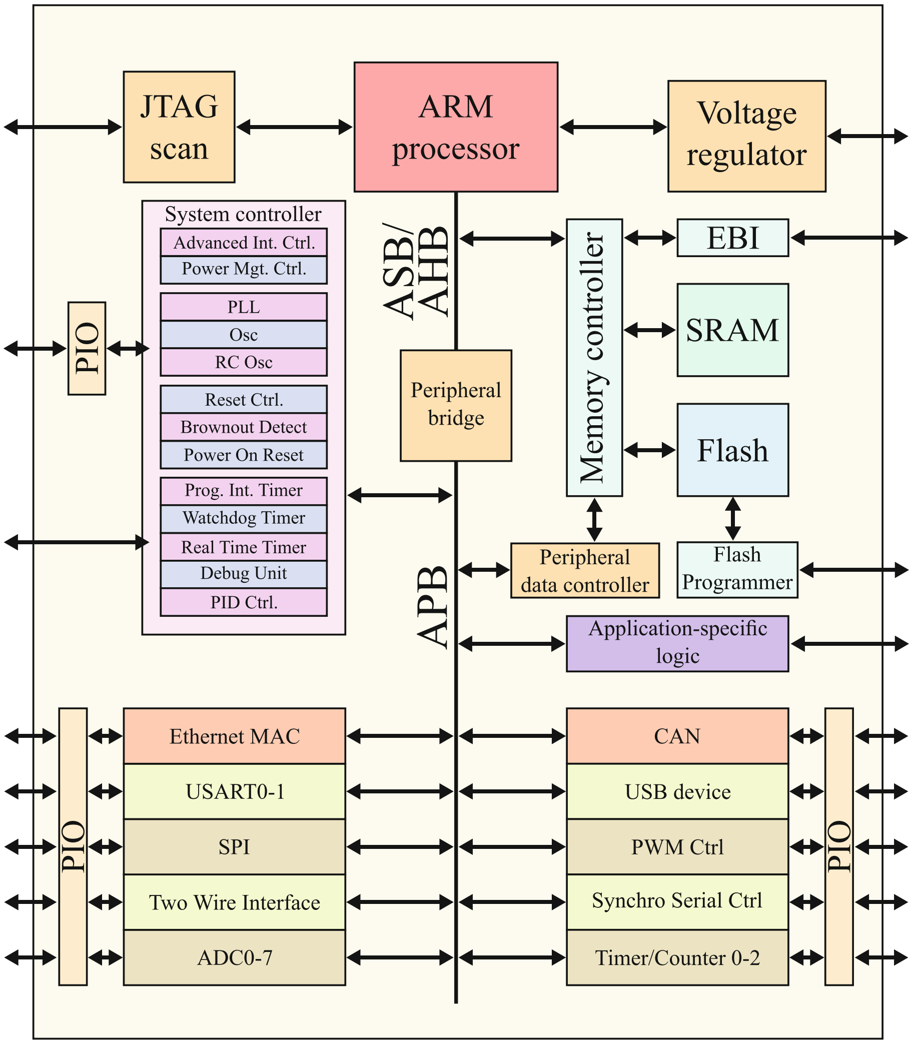


Fig. 1. Example for a SoC with processing elements, interconnect, and interfaces (By en:User:Cburnett - Own work in Inkscape based on en:Image:ARMSoCBlock Diagram.gif, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2866881)

2.1 Hardware/Software System Design Flow

The generic (classic) *design flow*[2] for custom computing systems is shown in Fig. 2. It is much more complex than a pure software development flow. The flow starts with a so-called *hardware-software-partitioning* that determines which parts of the overall behavior will be realized in hardware or software. While considering available hardware and software IP in conjunction with functional and non-functional requirements such as throughput, energy/power limitations, or quality aspects, the system (architecture) platform is determined. After a preliminary simulation, the actual implementation of the hardware and software components starts. Finally, the system components, their interaction, and the final system behavior are validated.

Since we expect software development flows to be well-known by the readers of this chapter, we will focus on the hardware development part in the following.

2.2 FPGA Basics

Hardware architectures realized in an *application specific integrated circuit (ASIC)* can no longer be changed after production (they are fixed geometries in silicon). In contrast, a *programmable logic device (PLD)* is shipped as a device with plenty of available hardware units that can be connected after production. This *programming* or *configuration* can be either one-time[3] or multiple times. A prominent example for the latter is a *field programmable gate array (FPGA).*

FPGAs are hardware devices that come with a large amount of flexible small hardware units, so-called lookup tables (LUTs). They are basically very small random access memorys (RAMs) that are written during the boot process (*"configuration"*) of the FPGA. Besides, FPGA provide a complex and flexible interconnect system that is configured together with the LUTs. Furthermore, special components such as Block RAMs (BRAMs), fixed bitwidth multiply-accumulate (MAC) units, multipliers, and I/O components are available.

FPGAs do not have a functional behavior before being initially configured. Some types can even be (partially) re-configured during operation, i.e., changing (parts of) the circuit while the rest of the system continues running. Thus, systems equipped with FPGAs allow a very high level of flexibility and dynamics (however, at the cost of an immensely complex design flow, see Fig. 2). In addition, combined CPU/GPU-FPGA systems are available, both in the high-performance computing (HPC)/data center and the embedded SoC domain.

The acquisition of the FPGA vendors Altera by Intel in 2015 and Xilinx by AMD in 2020 shows the potential of this technology for the future of the computing landscape.

[2] A lot of different elaborate system design flows exist [2,11,17] that are omitted here for the sake of clarity.

[3] One-time programmable devices are physically modified during the programming, e.g., by burning connections or melting so-called *antifuses* that create a conducting connection afterwards.

The proposed hardware architecture for computing the Link Assessment (LA) algorithm can be realized both on ASICs and FPGAs. In the following, we present our architecture in detail and illustrate the differences compared to classical CPU implementations.

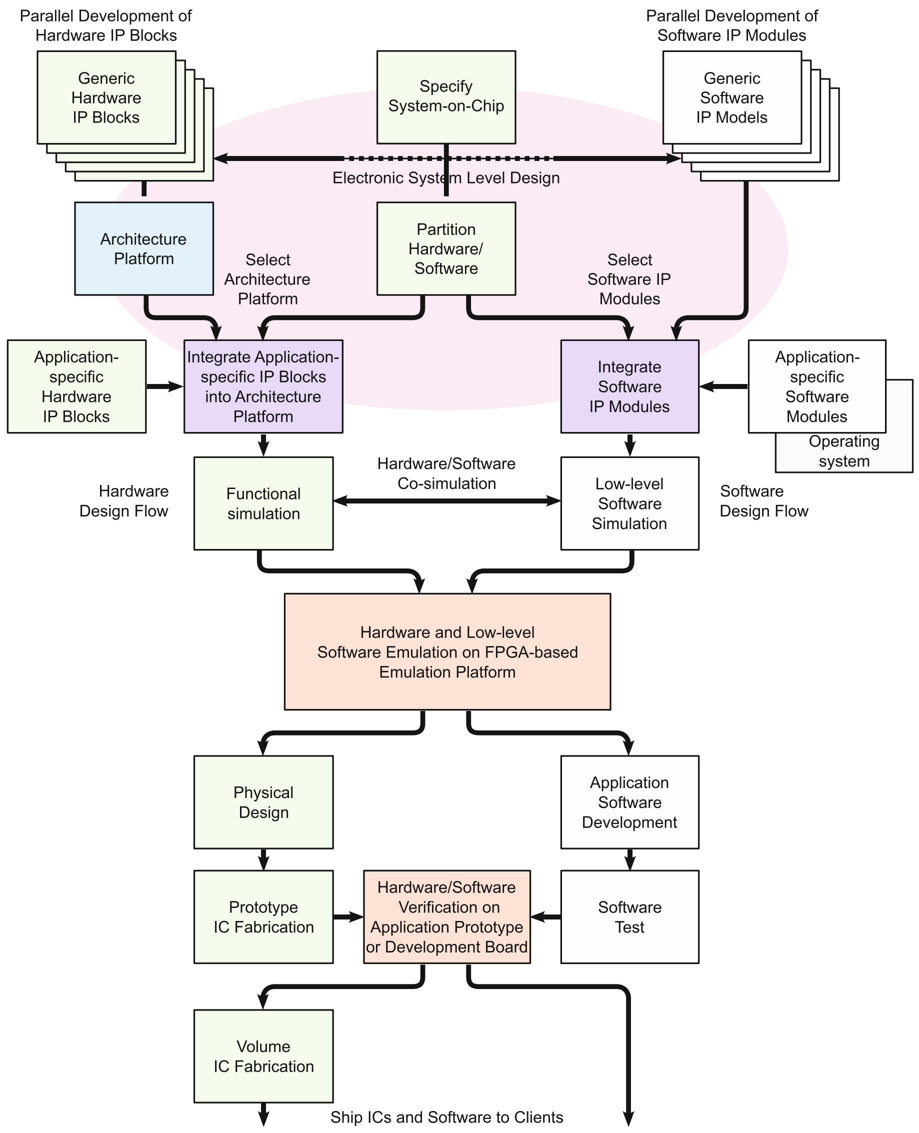

Parallel Development of Hardware IP Blocks
Generic Hardware IP Blocks
Specify System-on-Chip
Parallel Development of Software IP Modules
Generic Software IP Models
Electronic System Level Design
Architecture Platform
Select Architecture Platform
Partition Hardware/ Software
Select Software IP Modules
Application-specific Hardware IP Blocks
Integrate Application-specific IP Blocks into Architecture Platform
Integrate Software IP Modules
Application-specific Software Modules
Operating system
Hardware Design Flow
Functional simulation
Hardware/Software Co-simulation
Low-level Software Simulation
Software Design Flow
Hardware and Low-level Software Emulation on FPGA-based Emulation Platform
Physical Design
Application Software Development
Prototype IC Fabrication
Hardware/Software Verification on Application Prototype or Development Board
Software Test
Volume IC Fabrication
Ship ICs and Software to Clients

Fig. 2. Generic design flow for a SoC (By Traced by User:Stannered - en:Image:SoCDesignFlow.gif, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1864027)

3 Hardware Architectures for the Link Assessment Computation

Many applications in the big data context are based on fast and reliable identification of so-called *network motifs* in large networks, i.e., those subgraphs whose occurrence is significantly higher than expected in a random graph model [15]. This enables analyzing large-scale biological data in bioinformatics, connections in social networks, incident detection, and general graph data cleaning procedures by LA [22 SPP].

Network motif detection is actively investigated in current research, but mainly from the algorithmic point of view. From the implementation side, nearly all available work deals with mapping the motif detection problem on parallel CPU and GPU based clusters [9,14].

For the Link Assessment (LA) algorithm, we consider a special variant of motifs, the so-called *co-occurrence (coocc)* which is defined as the number of common neighbors between two nodes of graph. Formally, $coocc\ (u, v) = |N(u) \cap N(v)|$ for any pair of nodes $u, v \in G$, where $N(u)$ is the neighborhood of node u in graph G. Throughout this chapter, we use the shorthand "*coocc* matrix of a network/graph" in place of "the set of all node-pairwise *cooccs* of a network/graph," or $coocc(G) = \{coocc(u, v)\ \forall u, v \in G\}$. For a bipartite graph, $G = G(V_l, V_r; E)$, with vertex partitions V_l and V_r and edges $E \subset (V_l \times V_r)$, the *coocc* matrix can be defined for either partition, e.g., $coocc(V_l) = \{coocc(u, v) \forall (u, v) \in (V_l \times V_l)\}$, in which case V_l is called the side of interest. In this chapter, we focus on bipartite graphs.

The $coocc(u, v)$ by itself is a way of quantifying the similarity of nodes u and v. However, it is a strongly biased quantifier, e.g., w.r.t. the degree of the nodes. The LA algorithm reduces such biases by comparing the observed *coocc* of the real network with its expected value for a random graph model (null-model), namely the fixed degree sequence model (FDSM) [22 SPP,19 SPP]. As the name suggests, the FDSM is the set of all graphs configurations that share the same degree sequence as the observed graph, and it has been shown to provide more robust results than simpler null-models [22 SPP]. Since closed-form solutions for the expected co-occurrences, $coocc_{FDSM}(u, v)$, are not known, these quantities are estimated by a random sampling procedure, known as a Markov chain Monte Carlo (MCMC) approach.

The MCMC approach is divided in two main steps: (1) the randomization of the graph by repeatedly swapping its edges until an uncorrelated, and hence unbiased sample of the FDSM is reached, and (2) the computation of the sample's *cooccs*. Of key importance are (a) the number of swap trials between samples and (b) the number of samples drawn from the FDSM. For the interested reader, Chapter 3 presents the LA in more detail, including an in depth analysis of the effect of those parameters, (a) and (b), on the final quality of the results as well as on the total runtime of the algorithm. In fact, MCMC sampling is the most time consuming part of the LA algorithm.

Once enough samples have been created and evaluated, the node-pairwise similarities are calculated as the probability of finding, in the FDSM, a *coocc*(u, v) greater or equal than that of the original graph. The higher the probability, the lower the similarity between (u, v). The probability is estimated first by the p-value and ties are broken by the z-score (see Chapter 3).

In Sect. 3.2 we show that the LA performance is strongly bounded by the speed of the random accesses to the main memory. Aiming to reduce the effects of this unavoidable constraint, in 2015 we have presented the first dedicated embedded hardware accelerator optimized for this task [4 SPP]. Precisely tailored cache memories and computational units for the *coocc* calculation help reducing the number of random accesses by using a rather naive representation of the graph, which is not optimal for CPUs. This work is the basis for a granted patent [21 SPP].

In a follow-up work [3 SPP], we exploit the granularity of DRAM devices to increase the efficiency of main memory accesses during the random graph creation (the null model). We demonstrate the performance of our design with the Netflix Prize data set[4] and show that a single ASIC instance has a speedup of 5.6x compared to a 10-node Intel cluster while requiring 38x less memory and 1030x less energy.

3.1 Data Structures

The Link Assessment (LA) requires two main pieces of information: The graph and the co-occurrence and similarity measures matrices.

The graph is used by both compute kernels, i.e., the edge swapping (see Chapter 2) and the *coocc* calculation. The edge swapping kernel consists of randomly selecting two edges, (u, w) and (v, x) for $u, v \in V_l$ and $w, x \in V_r$, and swapping their connections, to get (u, x) and (v, w), if this does not modify the degree sequences of V_l and V_r. For the edge swapping to have a constant compute complexity, the data structures must provide direct access to existing edges of the graph (random edge selection) and a constant time check for the existence of the new, swapped edges (to preserve the degree sequences). While the adjacency list representation of the graph solves the first task, its adjacency matrix solves the second. Using only one of the data structures would drastically slow the edge swapping procedure. Therefore, we make use of both graph representations, as formalized next.

Given a bipartite graph $G(V_l, V_r; E)$ consisting of the vertex partitions V_l and V_r and the edges $E \subset (V_l \times V_r)$, an adjacency matrix $A = (V_l \times V_r)$ is stored. An entry in the matrix is $A_{u,w} = 1$ if $(u, w) \in E$, with nodes $u \in V_l, w \in V_r$. It is sufficient to store A with one bit per entry and a total storage requirement of $|V_l| \cdot |V_r|$ bits. The adjacency list representation is simply the list of all edges E, requiring $|E|(\lceil \log_2 |V_l| \rceil + \lceil \log_2 |V_r| \rceil)$ bits.

One *coocc* half-matrix is necessary for storing the real graph *cooccs*. It is a half-matrix since *coocc*(u, v) = *coocc*(v, u), and each pair of nodes $(u, v) \in (V_l \times V_l)$ must be evaluated. A second and identical structure is necessary for

[4] Available at https://www.kaggle.com/netflix-inc/netflix-prize-data. Last accessed on 24/11/2022.

storing the *cooccs* of each random graph sample. Instead of keeping as many *coocc* half-matrices as the number of samples, the similarity measures, p-value and z-score, are updated after each sample. For the p-values, a single half-matrix is required. For updating the z-score, it is sufficient to keep the sum and the sum-of-the-squares of the samples' *coocc*.

A summary of the memory footprint of each data structure is shown in Table 1.

Table 1. Memory footprint of the data structures for the LA

Variable	Required bits
Adj. matrix	$\|V_l\| \cdot \|V_r\|$
Adj. list	$\|E\|(\lceil \log_2 \|V_l\| \rceil + \lceil \log_2 \|V_r\| \rceil)$
$coocc_{ori}(u, v)$	$\lceil \log_2(\|V_r\|) \rceil$
$coocc_i(u, v)$	$\lceil \log_2(\|V_r\|) \rceil$
p-value count	$\lceil \log_2(\|samples\|) \rceil$
$\sum_i coocc_i(u, v)$	$\lceil \log_2(\|V_r\| \cdot \|samples\|) \rceil$
$\sum_i coocc_i(u, v)^2$	$\lceil \log_2(\|V_r\|^2 \cdot \|samples\|) \rceil$

3.2 Memory Boundedness

In order to demonstrate the memory boundedness of the LA, we use the roofline model [20] to profile a parallel, optimized CPU implementation of the algorithm. The roofline model is a visualization tool intended to evaluate the efficiency of computation kernels w.r.t. the underlying hardware. The maximum performance of the hardware is bounded, of course, by its maximum number-crunching speed, but also by the memory access bandwidth. These bounds are represented by the black lines (the Rooflines) in Fig. 3. The performance of a computing kernel is measured in operations per second, i.e., how busy the processor really is. Only integer operations (INTOP) are considered because the LA does not use floating-point numbers, and the G in GINTOP stands for Giga, i.e.,, billions of integer operations. The arithmetic intensity is defined as ratio between the number of operations over the total memory traffic, being measured in operations per byte.

The performance and arithmetic intensity of the edge swapping and the *coocc* computation kernels were measured by Intel Advisor[5]. They are presented in Fig. 3. We can see that the performance of the kernels are 1.3 GINTOP/s for edge swapping and 2.2 GINTOP/s for the *coocc* calculation. This is far from the attainable value by the CPU (109 GINTOP/s). This is because of the low arithmetic intensity of both kernels, as is expected from their tasks. The edge swapping kernel, for example, needs to access multiple random memory locations to only check for the existence of an edge, hence many bytes are accessed but

[5] https://software.intel.com/content/www/us/en/develop/articles/intel-advisor-roofline.html.

very little processing happens. Most of the time, this kernel is simply waiting for the data to be loaded, what we call a memory stall. During the stall, no processing occurs.

The impact of the stalls on the total runtime are given per memory hierarchy level. We can see that more that half of the total runtime is spend waiting for the DRAM. Moreover, the DRAM stalls account for almost 80% of the edge swapping runtime. This is expected from the intrinsically random memory access pattern of the edge swapping, which means that the cached data is hardly ever used.

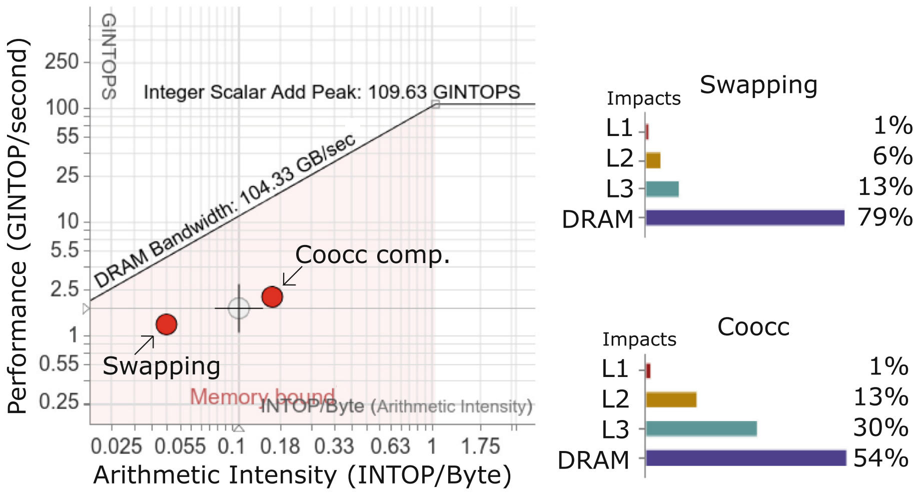


Fig. 3. Roofline analysis of the main compute kernels for the Link Assessment: Edge swapping and co-occurrence computation. Both kernels are strongly memory bounded, with 79% and 54% of the runtime spent in DRAM stalls for the edge swapping and *coocc* kernels, respectively. Machine: Intel Xeon E5-2640 v3 (16 cores at 2.6 GHz) with 2×32 GB DRAM.

3.3 Co-occurrence Calculation

Calculating the node-pairwise *coocc* of a given graph is the most time consuming part of the LA. Using the adjacency matrix, we iterate through each pair of rows (nodes in V_l) and count the number of columns (nodes in V_r) where both elements are 1, i.e., both edges exist. The computational complexity of this procedure is, therefore, $O(|V_l|^2 \cdot |V_r|)$.

Through the adjacency list, the complexity can be amortized to $O(\sum_{V_r} deg(w)^2)$, where $deg(w)$ is the degree of node $w \in V_r$. This particularly benefits networks whose degrees follow a power-law distribution, as is the case of most real networks [22 SPP]. For a CPU implementation of the LA, the adjacency list approach is preferred, even though the memory access pattern is unstructured (see Sect. 3.2).

From a hardware architecture design perspective, however, the adjacency matrix approach can be easily implemented with blocks of bit-wise ANDs followed by an adder tree, what we call *coocc* module. Due to the small size of such an operational block, it can be replicated multiple times, reaching a degree of parallelism that is not feasible in CPUs. To make use of such high parallelism without being constrained by the DRAM bandwidth requires a well-designed cache layout.

Calculating the *coocc* between all pairs of vertices in V_l in a naive way requires to load the same data many times. For example, calculating the *coocc* between $u, v \in V_l$ requires edges connected to u and v, or in other words the two rows u and v of the matrix A. When the *coocc* is later calculated between u and w, the same row A_u needs to be loaded. This leaves huge potential for an optimized memory hierarchy and algorithms to minimizing data transfer.

We presented an appropriate solution for this issue in 2015 [4 SPP]: The key idea was to add a row-cache to the *coocc* module. The row-cache must be able to store one complete row of the adjacency matrix.

Having k parallel *coocc* units, we use their caches to store a consecutive block of k rows $A_u, .., A_{u+k-1}$. Then we stream one by one all following rows through the *coocc* modules, starting with A_{u+k}. With each new row A_v the modules can calculate the *coocc* of all pairs of the cached rows $(u, v), .., (u + k - 1, v)$. Algorithm 1 formalizes this scheme.

Algorithm 1: Implementation of the *coocc* computation step for K *coocc* modules

```
Data: Graph $G((V_l, V_r); E)$ stored as adjacency matrix $A = (V_l \times V_r)$, $V_l$ being
      the side of interest
Result: coocc for all pairs of vertices $(u, v) \in (V_l \times V_l)$
1 for $u := 1$ to $|V_l|$ step $K$ do
2     k := 0
3     for $v := u$ to $|V_l|$ do
4         Stream row $A_v$ from DRAM
5         if $k \geq 1$ then
6             Compare the streamed row with all previously cached rows 1 to $k$
              and calculate the coocc for the pairs: $(u, v), .., (u + k - 1, v)$
7         if $k < K$ then
8             k := k + 1
9             Store the streamed row in cache $k$
```

The main advantage of this scheme is solving the scaling problem. While adding m times more modules reduces the runtime by a factor of m, it does not increase the requirements for external bandwidth since only one row has to be streamed through all the blocks at each given time. This allows us to place hundreds, if not thousands, of *coocc* units next to each other, providing massive speedups.

Figure 4(a) shows the data path tailored to this task, consisting of an adder tree and accumulator. Each edge cache has a capacity of 64 kB, targeting a frequency of 400 MHz. For a 64 bit double data rate (DDR) channel at 800 MHz, we get 256 edges per cycle when running the *coocc* units at 400 MHz. That means the adder tree has a width of 128 adders at the top and a depth of seven stages. Four *coocc* modules are synthesized in a single cell and combined in a grid of 5 times 12, for a total of 240 *coocc* modules. To distribute the data to the caches or to stream further rows of the matrix a tree-like replication network is used,

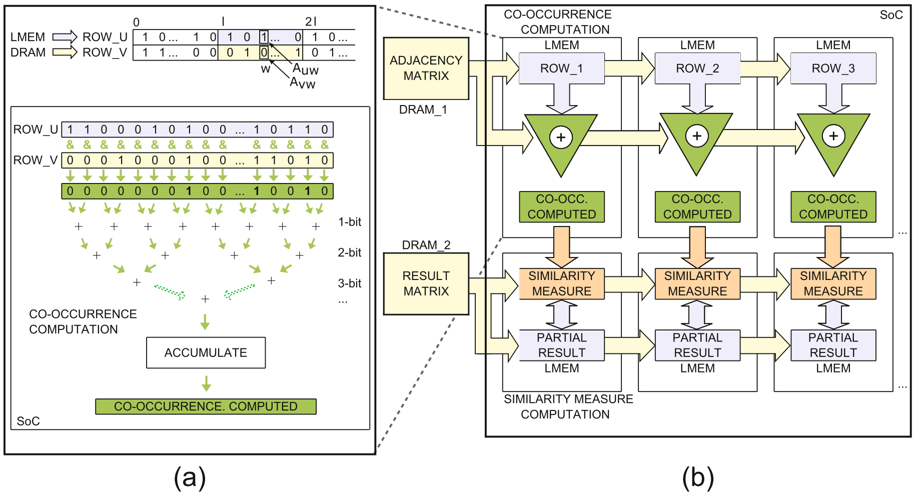


Fig. 4. The *coocc* and result module (b) works on one dataset after another, always updating the same result. It loads one row of the graph into the caches (local memory (LMEM)) and first calculates the *coocc* before calculating the similarity measures. The *coocc* module (a) consists of an efficient adder tree operating on blocks of l edges per cycle. While the similarity measures, lower half in (b), consists of several arithmetic blocks and it is only called once per row, making it possible to share most of the resources.

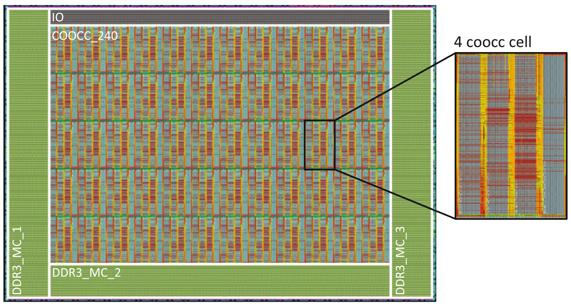


Fig. 5. ASIC layout in 28 nm technology. It consists of 240 *coocc* modules, three DRAM controllers (green) and IO logic. The swap randomization block is not visible here due to its small size .(Color figure online)

while for the results a shift register over the whole chip is used. That makes the architecture perfectly scalable.

In total, this architecture accumulates the *coocc* from $240 \times 256 = 61,440$ matrix columns per cycle, or $\sim 24.5 \times 10^{12}$ columns per second. In a comparison with the fastest CPU based population count [16] running at 3.4 GHz, that represents a speed up of $\sim 59\times$.

The rest of the design is occupied by memory controllers and IO, see Fig. 5. For the memory controllers, we have estimated the numbers based on the corresponding publications [6,10]. The whole ASIC has a size of 51.2 mm^2 and average power consumption of 11.7 W.

Partial-Line Cache Optimization. In a follow-up work [3 SPP], we further increased the efficiency of the hardware architecture by introducing the concept of partial line caches. Since the area of the *coocc* modules are dominated by their cache, reducing the cache size enables much higher degrees of parallelism. However, if the *coocc* modules cannot hold an entire row of the adjacency matrix, the partial results must be temporarily stored, raising the question of the optimal cache size for achieving the best performance.

As will be detailed in Sect. 3.4, higher granularity DRAM channels (shorter word-sizes) can be used to accelerate the graph randomization step. However, they increase the latency of accessing the adjacency matrix rows, therefore presenting the worst-case for the *coocc* computation.

In Fig. 6 we have simulated the time it takes to process the adjacency matrix with the time it takes to store the partial result on average for one line segment when using a channel word-size of 8-bit (the smallest possible). Since those operations are pipelined, the optimal cache size is given by the Pareto front between the two operations. The smallest latency is reached for a cache size of 8 kB.

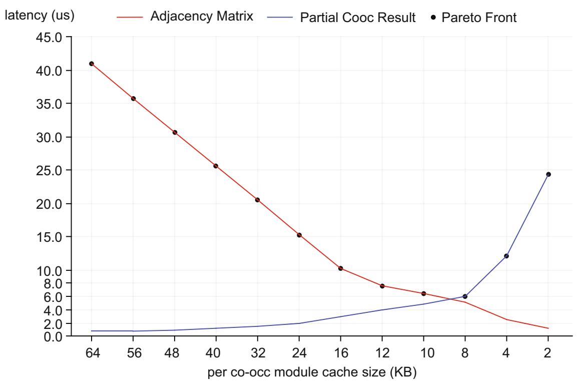


Fig. 6. Latencies of accessing the input data stored in Adjacency Matrix rows in comparison with latency for storing partial *coocc* results, assuming the same channel width for both memories involved in the design. The Pareto front is the maximum of each. Numbers are for 8-bit channel DRAMs.

While in the first design 240 *coocc* modules with 64 kB caches have been used, 8 kB caches allow us to increase the number of modules up to 1920 for the same total cache size. This results in a similar total chip area, from $51.2\,\text{mm}^2$ to $57.3\,\text{mm}^2$, as the caches dominate the *coocc* module in both cases. With this approach, we could further reduce the runtime of the *coocc* computation by a factor of $8\times$ when using the same 64-bit channels, or maintain the same speed when using 8-bit channels.

3.4 Swap Randomization

With the accelerated *coocc* computation, the generation of each sample, i.e., the randomization of the graph becomes the bottleneck. Edge swapping is a strictly sequential operation in that any swap can depend on the result of the last swap, therefore its parallelization is not as straightforward as instantiating more processing units. Nevertheless, we addressed this bottleneck by exploiting the fine-grained access to DRAM [3 SPP], what is only possible when implementing our own memory controller, as well as a collision-aware swap parallelization.

Fine Grained DRAM Access. Most modern CPU have a fixed size interface of 64 bits with the DRAM. DRAM devices, however, can have higher granularity interfaces of, e.g., 8 bits ($\times 8$), and they are physically combined into groups of 8 devices to build the 64 bits interface. A fixed burst length of 8 DRAM accesses fills up one cache line of 512 bits, or 64 bytes. For any modern CPU, one cache line, i.e., 512 bits, is the minimum amount of data that can be loaded from DRAM.

Since the swap randomization operates only on single integers and single bits, reducing the word length of the DRAM interface increases the "computations per loaded bit" (the arithmetic intensity, see Fig. 3) immensely. Indirectly, of course, it also increases the performance because the swap randomization is bounded by the random memory access latency.

We have derived an alternative hardware architecture that slightly modifies the memory controller in order to address each of the DRAM devices (with 8 bit interfaces) independently [3 SPP]. Normally, the memory controller addresses all 8 DRAM devices of a memory channel as if it was a single device, i.e., it sends the same commands and addresses to all devices. This allows the memory channel to share the command and address lines for all devices, saving energy and area at the cost of having a common address space. The data lines (8 or 64 per $\times 8$ or $\times 64$ device), on the other hand, cannot be shared, as the data in each DRAM device must be transferred independently. By introducing a *chip select signal* and interleaving the commands to each DRAM device, we can transform the common address space into 8 independent ones. This works because, during the DRAM latency (data request to data ready), the DRAM device ignores address and command lines, as they get internally saved at the request moment. That way, we can load only $8 \times 8 = 64$ bits instead of $8 \times 64 = 512$ bits in one DRAM device access. This is a slight modification in the memory controller and channel, but one that could not be accomplished without custom hardware design.

For that scheme to be the most efficient, it requires that the data stored in each DRAM device to be independent. That is, each DRAM chip holds its own copy of the graph, as shown in Fig. 7(b). With that we can read or write 8 random numbers in the same time with a ×8 channel compared to a single with one ×64 channel, as shown in Fig. 7(c)(d). This scheme speeds up the swap randomization by a factor of 4× up to 8×.

Figure 7(a) shows the alternative architecture using two ×64 memory channels. This design is more suitable whenever the *coocc* calculation is the bottleneck of the algorithm, while the design in Fig. 7(b) provides faster graph randomization. This trade-off is depicted in Sect. 4.

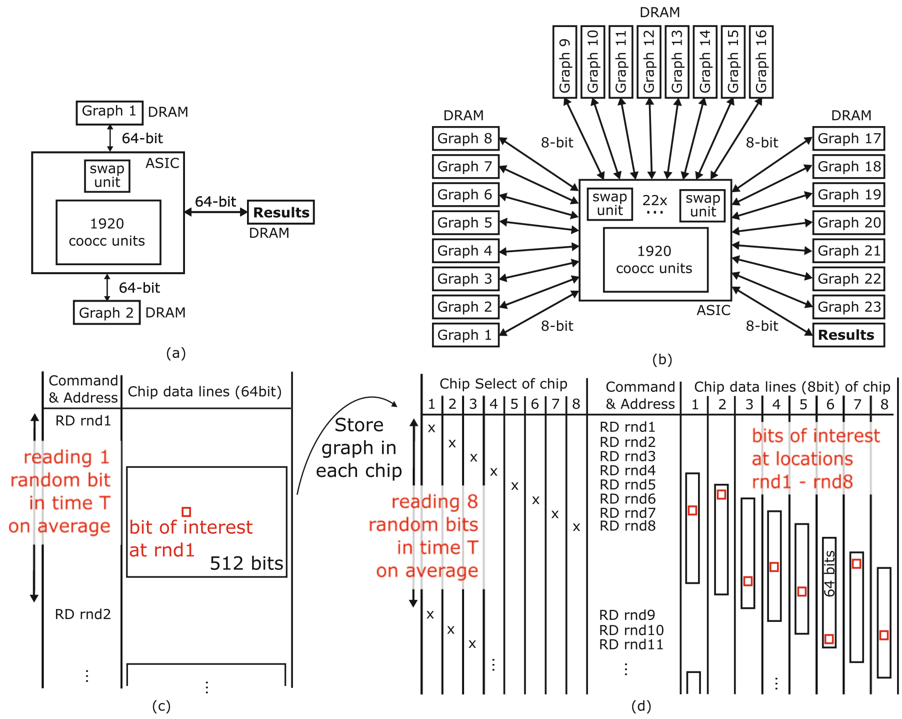


Fig. 7. Showing the ASIC for two memory configurations: ×64 (a) and ×8 (b) channels. In the case (a) only two graphs are stored and one swap unit is active, while in case (b) 23 graphs are stored and 22 swap units are active. Architecture (a) is useful for small number of swaps, while architecture (b) is useful for high number of swaps. Showing how the different random reads are performed for a ×64 channel (c) and ×8 channels (d). By interleaving the random accesses of 8 swap units with chip select over one command and address channel, 8 reads can be performed for (d) in the same time as one read for (c). This results in an 8× speedup.

Collision-Aware Swap Parallelization. Edge swapping is an inherently sequential operation in that every step can depend on the previous ones. For large graphs with millions of edges, we access the memory at random locations for billions of chained swaps. Even then, we can divide the edge swapping chain into chunks that can be processed in parallel, if we make sure that none of the swaps depend on the previous ones in the same chunk. These chunks can be reordered by the memory controller in order to ensure the minimum amount of random accesses.

We have simulated the performance of the swap parallelization for different chunk sizes with the DRAMSys tool [12]. For that, we created trace files that describe the access pattern to the DRAM. The speedup saturates at 2.5× for a chunk size of $N = 12$ parallel swaps. Since N is small, checking for collisions between swaps is much faster than writing the swapped edges back to DRAM, therefore it does not incur any time overhead.

Table 2. Cluster ASIC Comparison

Implementation	Memory [GB]	Runtime [hour]	Power [W]	Energy [MJ]
Low number of swaps (|nodes| ln |nodes|):				
ASIC (1920 modules, 64-bit channels)[a]	5.3	1.51	20.1	0.11
10 node Intel cluster[b]	202	8.5 (5.6x)	3700	114 (1030x)
ASIC (240 modules, 64-bit channels)[c]	4.6	9.0 (6.0x)	15.8	0.51 (4.6x)
High number of swaps (|edges| ln |edges|):				
ASIC (1920 modules, 8-bit channels)[a]	30.9	11.1	13.3	0.53
10 node Intel cluster[b]	202	16 (1.4x)	3300	190 (360x)
ASIC (240 modules, 64-bit channels)[c]	4.6	483 (44x)	10.9	19 (36x)

[a]node including: ASIC with 1920 *coocc* modules, 28 nm; 48 GB DDR3 memory (×64 or ×8 channels); board (ethernet, clocks), power supply.
[b]each node: 2×Intel Xeon X5680 @ 12 × 3.33 GHz, 32 nm; 48 GB DDR3 memory
[c]node including: ASIC with 240 *coocc* modules, 28 nm; 8 GB DDR3 memory (×64 channel); board (ethernet, clocks), power supply.

4 Performance Comparison

For demonstrating the performance of our design we have calculated the similarity measures for the Netflix Prize data set[6], specifically the good ratings (4 or 5 stars) from users to movies. The resulting graph has 17,769 movies, 478,615 users, and 56,919,190 edges. In this case, V_l are the the movies, V_r the users.

In practice, the number of swaps in the randomization process is chosen between $|nodes| \ln |nodes| = 6{,}259{,}639$ and $|edges| \ln |edges| = 1{,}016{,}414{,}121$. To demonstrate that our design qualifies for the full range, we compare it for both of those extremes. The exact number in practice usually depends on the

[6] Available at https://www.kaggle.com/netflix-inc/netflix-prize-data. Last accessed on 24/11/2022.

nature of the graph. A heuristic for determining the optimal number of swaps is discussed in Chapter 3.

Table 2 compares our ASIC and our optimized cluster implementations of the LA algorithm. The cluster implementation was developed specifically for this reference work and tested on two Intel Xeon X5680 @ 12×3.33 GHz, 32 nm server nodes. Optimization involved the selection of an algorithm that minimizes computing time for the given memory resources, removing locks by data partitioning, and data access linearization [4 SPP,3 SPP].

Our first ASIC design (240 *coocc* modules) has a runtime performance comparable to the cluster implementation if a low number of swaps is necessary. Notice, however, that it becomes almost useless (takes 20 days to complete) if $|edges| \ln |edges|$ swaps are required. This is clear since in this first architecture we only focused on accelerating the *coocc* calculation. Still, the total energy consumption is 10x lower (notice that the total energy takes into account the total runtime). This goes to show the amount of energy overhead for software implementations, or how much energy can be saved by task specific ASICs. This conclusion is interesting for both ends of the computing spectrum: The embedded computing systems that are limited by battery capacity, size, and power constraint, and the high performance computing, limited by energy expenses and power dissipation issues.

Our second design shows how reconfigurability can address data-dependent bottlenecks (i.e., the *coocc* or the edge swapping). By using smaller word-sizes ($\times 8$ channels), we can accelerate both the *coocc* and edge swapping in such a way that the Link Assessment becomes 45% faster than the cluster implementation while consuming 360x less energy. When fewer swaps are necessary, the word-size can be increased to $\times 64$ channels, further reducing the *coocc* computation time (the primary bottleneck), reaching a speed up of 5.6x compared to software. The total energy economy, in this case, is even more impressive: From 114 MJ in software to only 0.11 MJ in the custom design. This is partially due to the large reduction of 38x in main memory footprint, from 202 GB to 5.3 GB.

5 Conclusion

Further increasing computational performance in modern technologies has become a key challenge for the whole hardware and software industry. Phenomena such as *Dark Silicon* force system designers to move to highly heterogeneous systems, consisting of a large amount of highly dedicated hardware accelerators in combination with classical programmable architectures such as CPUs and GPUs. Since hardware accelerators focus on specific tasks, they can be much more power/energy and compute efficiently than the latter ones.

In this chapter, we present a hardware architecture for the Link Assessment (LA) algorithm, used for cleaning up noise data in large graphs. Processing and analyzing large graphs will remain a key application in HPC for the next decades. Since the current bottleneck for speeding up this task is fast random access to memory, with standard DRAM architectures and controllers on commodity

HPC nodes we experience a hard performance limit, together with high energy consumption.

Our proposed architecture uses custom data structures and exploits bit-wise access to the data in order to overcome these limitations. On a 28 nm ASIC device with a DDR3 controller it is 1030x more energy efficient compared to a standard compute cluster, using 38x less memory in total. We show multiple optimization techniques that are specific to custom hardware designs, such as a slight memory controller modification that reduces the average random access latency; and a tailored cache design that enables scalable parallelism w.r.t. memory bandwidth. The architecture is fully flexible and can also be ported as an FPGA accelerator solution. This clearly illustrates the potential of hardware accelerators for the LA in particular and the graphs analysis domain in general.

Transferring the concepts to other algorithms such as Curveball (see Chapter 2) is the subject of ongoing work.

References

1. Asanovic, K., et al.: A view of the parallel computing landscape. Commun. ACM **52**(10), 56–67 (2009). https://doi.org/10.1145/1562764.1562783
2. Brugger, C.: A new approach to efficient heterogeneous computing = Ein neuer Ansatz für effiziente, heterogene Datenverarbeitung. Ph.D. thesis, University of Kaiserslautern, Germany (2016)

3 SPP. Brugger, C., Grigorovici, V., Jung, M., de Schryver, C., Weis, C., Wehn, N., Zweig, K.A.: A memory centric architecture of the link assessment algorithm in large graphs. IEEE Des. Test **35**(1), 7–15 (2018). https://doi.org/10.1109/MDAT.2017.2750900

4 SPP. Brugger, C., et al.: A custom computing system for finding similarties in complex networks. In: ISVLSI, pp. 262–267. IEEE Computer Society (2015). https://doi.org/10.1109/ISVLSI.2015.78

5. Duranton, M., et al.: Hipeac vision 2019. European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC) (2019)
6. Dutoit, D., et al.: A 0.9 pJ/bit, 12.8 GByte/s WideIO memory interface in a 3D-IC NoC-based MPSoC. In: Symposium, VLSIT, pp. C22–C23. IEEE (2013)
7. Esmaeilzadeh, H., Blem, E.R., Amant, R.S., Sankaralingam, K., Burger, D.: Dark silicon and the end of multicore scaling. IEEE Micro **32**(3), 122–134 (2012). https://doi.org/10.1109/MM.2012.17
8. Garraghan, P., Al-Anii, Y., Summers, J., Thompson, H., Kapur, N., Djemame, K.: A unified model for holistic power usage in cloud datacenter servers. In: UCC, pp. 11–19. ACM (2016). https://doi.org/10.1145/2996890.2996896
9. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the GPU using CUDA. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 197–208. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77220-0_21
10. Howard, J., et al.: A 48-core IA-32 message-passing processor with DVFS in 45 nm CMOS. In: ISSCC, pp. 108–109. IEEE (2010). https://doi.org/10.1109/ISSCC.2010.5434077

11. Jung, M.: System-level modeling, analysis and optimization of dram memories and controller architectures. Ph.D. thesis, University of Kaiserslautern, Germany (2017)
12. Jung, M., Weis, C., Wehn, N.: Dramsys: a flexible DRAM subsystem design space exploration framework. IPSJ Trans. Syst. LSI Des. Methodol. **8**, 63–74 (2015). https://doi.org/10.2197/ipsjtsldm.8.63
13. Lee, E.A., et al.: The swarm at the edge of the cloud. IEEE Des. Test **31**(3), 8–20 (2014). https://doi.org/10.1109/MDAT.2014.2314600
14. Miller, B.A., et al.: A scalable signal processing architecture for massive graph analysis. In: ICASSP, pp. 5329–5332. IEEE (2012). https://doi.org/10.1109/ICASSP.2012.6289124
15. Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., Alon, U.: Network motifs: simple building blocks of complex networks. Science **298**(5594), 824–827 (2002). https://doi.org/10.1126/science.298.5594.824
16. Mula, W., Kurz, N., Lemire, D.: Faster population counts using AVX2 instructions. Comput. J. **61**(1), 111–120 (2018). https://doi.org/10.1093/comjnl/bxx046
17. de Schryver, C.: Design methodologies for hardware accelerated heterogeneous computing systems. Ph.D. thesis, University of Kaiserslautern, Germany (2014)
18. Slavakis, K., Giannakis, G.B., Mateos, G.: Modeling and optimization for big data analytics: (statistical) learning tools for our era of data deluge. IEEE Signal Process. Mag. **31**(5), 18–31 (2014). https://doi.org/10.1109/MSP.2014.2327238
19 SPP. Spitz, A., Gimmler, A., Stoeck, T., Zweig, K.A., Horvát, E.: Assessing low-intensity relationships in complex networks. PLoS ONE **11**(4), 1–17 (2016). https://doi.org/10.1371/journal.pone.0152536
20. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009). https://doi.org/10.1145/1498765.1498785
21 SPP. Zweig, K.A., Brugger, C., Grigorovici, V., De Schryver, C., Wehn, N.: Automated determination of network motifs (2015)
22 SPP. Zweig, K.A., Kaufmann, M.: A systematic approach to the one-mode projection of bipartite graphs. Soc. Netw. Analys. Min. **1**(3), 187–218 (2011). https://doi.org/10.1007/s13278-011-0021-0

Graph-Based Methods for Rational Drug Design

Andre Droschinsky[1], Lina Humbeck[2], Oliver Koch[3], Nils M. Kriege[4], Petra Mutzel[5](✉), and Till Schäfer[5]

[1] Department of Computer Science, TU Dortmund University, Dortmund, Germany
andre.droschinsky@tu-dortmund.de

[2] Computational Chemistry, Medicinal Chemistry, Boehringer Ingelheim Pharma GmbH & Co. KG, Ingelheim am Rhein, Germany
lina.humbeck@boehringer-ingelheim.com, lina.humbeck@tu-dortmund.de

[3] Institute of Pharmaceutical and Medicinal Chemistry, and Center for Multiscale Theory and Computation, University of Münster, Münster, Germany
oliver.koch@uni-muenster.de

[4] Faculty of Computer Science, University of Vienna, Vienna, Austria
nils.kriege@univie.ac.at

[5] Institute for Computer Science, University of Bonn, Bonn, Germany
petra.mutzel@cs.uni-bonn.de, till.schaefer@uni-bonn.de

Abstract. Rational drug design deals with computational methods to accelerate the development of new drugs. Among other tasks, it is necessary to analyze huge databases of small molecules. Since a direct relationship between the structure of these molecules and their effect (e.g., toxicity) can be assumed in many cases, a wide set of methods is based on the modeling of the molecules as graphs with attributes.

Here, we discuss our results concerning *structural* molecular similarity searches and molecular clustering and put them into the wider context of graph similarity search. In particular, we discuss algorithms for computing graph similarity w.r.t. maximum common subgraphs and their extension to domain specific requirements.

Keywords: Drug discovery · Cheminformatics · Graph similarity · Molecular similarity · Maximum common subgraph · Maximum similar subgraph · Structural graph set clustering · Subgraph mining · Molecular library · BRD4

1 Introduction

The era of big data has reached academic and industrial pharmaceutical drug research in the last decade, which has changed how drugs are developed. Nowadays, large collections of bioactivity data and large databases of potentially synthesizable molecules exist. Publically available bioactivity databases like ChEMBL [4] or Pubchem [25] contain over 16 million data points about molecules that modulate protein or drug target functions. This allows data-driven decision via in-depth data mining and knowledge discovery approaches, e.g., the identification of similar molecules for the prediction of a known protein target or unwanted side effects. The extraction of molecular features enables an increasingly reliable prediction of properties such as toxicity or oral availability.

The chemical space of drug-like molecules provides another source of big data. Theoretical analysis of the comprehensive chemical space estimates around 10^{62} molecules with a typical drug size. Among those, around 166 billion molecules are described by the chemical universe database GDB-17 that was build up using 17 standard atoms that occur within drugs [47]. The REAL space[1], a large collection of commercially available chemical compounds, contains about 15.5 billion molecules that are potentially synthesizable. Finally, the current version of the ZINC database [55] contains over 750 million purchasable compounds that were already synthesized.

Several established tools and work-flows are available that utilize bioactivity data or the chemical space for the rational development of bioactive molecules [20 SPP]. These approaches are based on the common basic assumption that similar molecular structures have similar bioactivities. A classical approach for identifying similarity between molecules is to use molecular fingerprints that are fixed size vectorial representations of structural characteristics, e.g., extended-connectivity fingerprints [46]. Although this form of representation allows fast comparisons and the usage of fast vector-based tools, vectorization suffers from an information loss and can lead to inaccurate discrimination of similar molecules. This becomes a problem for the above described 'big' molecular databases since the available similarity measures do not discriminate enough. Thus, similarity searches in such databases contain too many false positives, which hampers further processing.

A more accurate comparison of molecules is directly based on the graph representation of the chemical structures. This representation allows the modeling of the molecules as graphs with attributes and the use of graph-theoretic concepts, algorithms and tools to analyze molecular databases. Figure 1 shows two similar molecules and their graph representation. The atoms are modeled as vertices, and the bonds between the atoms as edges. Attributes for the graph could be, e.g., a label for the vertices providing the atom name and a label for the edges encoding the binding type.

Unfortunately, the use of molecular graphs to compare two molecules based on the concept of isomorphism is notoriously much more time consuming compared to molecular fingerprint-based similarity search. Additionally, a comparison based on the maximum common substructure (maximum common subgraph) between two molecules may fail in the identification of molecules with similar chemical properties since the classical definition of a common substructure is too strict under some circumstances. Therefore, novel methods are urgently needed for the analysis of the still increasing amount of molecular data. The focus of the interdisciplinary project “Graph-based Methods for Rational Drug Design” has been the development of new structural approaches w.r.t. molecular similarity search and molecular clustering. This chapter presents some of the main results and puts them into a wider context of graph similarity.

Preliminaries and mathematical definitions are provided in Sect. 2. State-of-the-art methods for comparing graphs w.r.t. the size of their *Maximum Common Subgraph (MCS)* in the context of molecular graphs are discussed in Sect. 3. For drug design,

[1] https://enamine.net/library-synthesis/real-compounds.

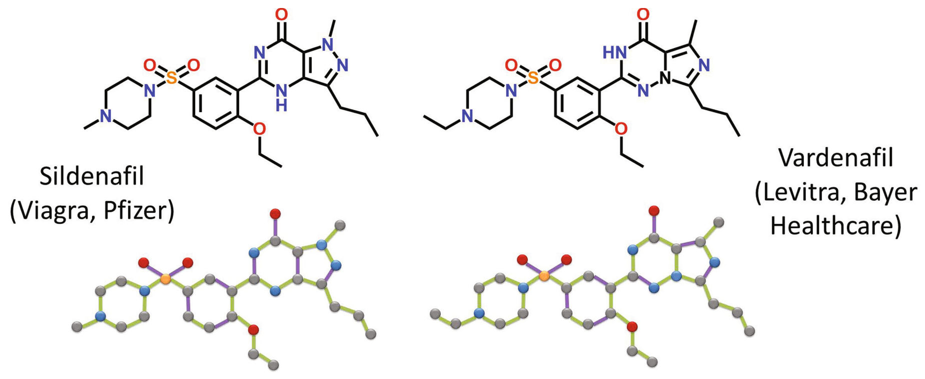


Fig. 1. Two similar molecules (Sildenafil and Vardenafil) and their corresponding graphs. The colors display the atom types (nodes) and bond types (edges).

it is often advisable to preserve certain molecular substructures –such as rings, blocks, or bridges– in comparisons since they have special biochemical properties as a whole. This method for comparing molecules can be further improved by incorporating chemical knowledge about reasonable atom or substructure substitutions that presumably do not affect bioactivity considerably. Figure 1 shows an example of two drugs with atom substitutions in the bicyclic structure that do not affect bioactivity. In our model of similarity, it is allowed to change certain structures of the graphs and still mark them as *structurally equivalent*. Thus, we have introduced the *Maximum Similar Subgraph (MSS)* problem. Our findings, including algorithms and experimental results, are discussed in Sect. 3.2.

Clustering analysis is used for a variety of tasks in drug discovery. This includes complexity reduction, structure activity relationship reasoning in visual analytics, novelty analysis of de novo databases (see Sect. 5.3), diversity analysis, structured sampling, and many more. Cluster analysis on huge molecular databases is the topic in Sect. 4. First, we discuss computational and information-theoretic challenges before we present a scalable state-of-the-art structural clustering algorithm (*StruClus*) that tackles these challenges. In Sect. 5, we discuss some selected successful applications in rational drug design in the context of this priority program. In a scaffold-focused analysis of bioactivity data, we discovered an unexpected similarity in ligand binding between two important drug targets (BRD4 and PPARγ) in cancer therapy (cf. Sect. 5.2). This discovery was possible using Scaffold Hunter, an open-source tool developed in our group to support the drug discovery process (cf. Sect. 5.1). In Sect. 5.3, we present CH*I*PMUNK, a new virtual database of more than 95 million synthesizable small molecules. Using *StruClus*, it was possible to demonstrate the novelty of the database in comparison to existing molecular libraries.

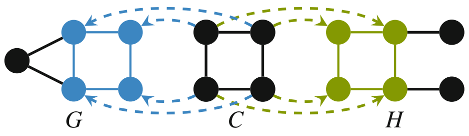


Fig. 2. Example: A common subgraph C of the graphs G and H. Dashed arrows indicate the subgraph isomorphism.

2 Preliminaries

An *undirected labeled graph* $G = (V,E,l)$ consists of a finite set of *vertices* $V(G) = V$, a finite set of *edges* $E(G) = E$ and a labeling function $l : V \uplus E \to L$, where L is a finite set of *labels*. An edge $\{u,v\}$ connects two vertices $u,v \in V$, $u \neq v$. A (simple) *path* of length n is a sequence of vertices $(v_0,\dots,v_n)$ such that $\{v_i,v_{i+1}\} \in E$ and $v_i \neq v_j$ for $i \neq j$, $i,j = 0,\dots,n-1$. A *tree* is a graph in which any two vertices are connected by a unique path. A graph is called *planar* if it admits a drawing on the plane without edge crossings, and it is *outerplanar* if such a drawing is possible in which every vertex lies on the boundary of the outer face.

For our similarity approaches based on subgraph isomorphisms, we need the following definitions. Let G and H be two undirected labeled graphs. A *(label preserving) subgraph isomorphism* from G to H is an injection $\psi : V(G) \to V(H)$, where $\forall v \in V(G) : l(v) = l(\psi(v))$ and $\forall u,v \in V(G) : \{u,v\} \in E(G) \Rightarrow \{\psi(u),\psi(v)\} \in E(H) \wedge l(\{u,v\}) = l(\{\psi(u),\psi(v)\})$. If there exists a subgraph isomorphism from G to H, we say H *supports* G, G is a *subgraph* of H, H is a *supergraph* of G or write $G \subseteq H$. If additionally $\{u,v\} \in E(G) \Leftarrow \{\psi(u),\psi(v)\} \in E(H)$ for all $u,v \in V(G)$, then ψ is an *induced* subgraph isomorphism. If there exists a subgraph isomorphism from G to H and from H to G, the two graphs are isomorphic. A *common subgraph* (cf. Fig. 2) of G and H is a graph C that is subgraph isomorphic to G and H. A *maximum common subgraph* (MCS) is a common subgraph of maximum size (vertices plus edges).

A graph $G = (V,E)$ with $|V| \geq 3$ is called *biconnected* if $G \setminus \{v\}$ is connected for each $v \in V$. A maximal biconnected subgraph of a graph G is called *block*. An edge $\{u,v\} \in E(G)$ not contained in any block of G is a *bridge*. A vertex v of G is called *cutvertex*, if $G \setminus \{v\}$ consists of more connected components than G. A *BC-tree* BC^G of a graph G consists of a node for each block and bridge in G and all the cutvertices of G. Two nodes (blocks or bridges) b,b' in a BC-tree are connected through the path bcb' if they share the cutvertex $c \in V(G)$. Figure 3 exemplifies a graph and its BC-tree. Let S and G be graphs, and $\psi : V(S) \to V(G)$ be a subgraph isomorphism. Then ψ is *block-and-bridge-preserving* (BBP) if any two edges in different blocks in S map to different blocks in G, and each bridge in S maps to a bridge in G.

The *support* $\mathrm{supp}(G,\mathcal{G})$ of a graph G over a set of graphs $\mathcal{G}$ is the fraction of graphs in $\mathcal{G}$ that support G. G is said to be *frequent* if its support is larger or equal than a *minimum support threshold* $\mathit{supp}_{\mathrm{min}}$. A frequent subgraph G is *maximal* if there exists no proper frequent supergraph of G.

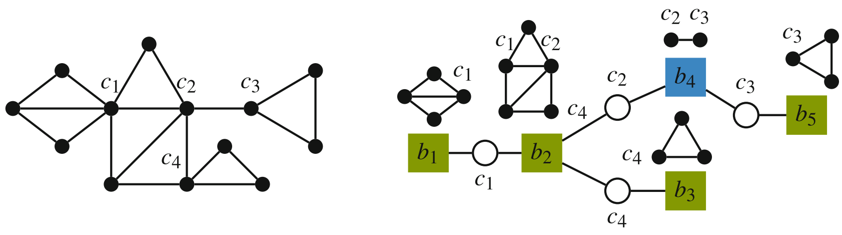


Fig. 3. A connected graph (left side) and its BC-tree (right side). The BC-trees' block nodes are depicted as green squares; the bridge nodes as blue squares. The white filled circles are the cutvertices. The associated subgraphs of G are depicted above the blocks and bridges. (Color figure online)

3 Molecular Similarity Based on Graphs

An essential criterion of molecular similarity in drug design is not only the similarity in chemical structure but also the similarity in biological activity or bioactivity. In order to obtain molecular similarities meeting this requirement, we introduce a graph-based method, which addresses the following problem.

Definition 1. *Given two molecular graphs G and H, the* maximum similar subgraph problem *is to find chemical meaningful subgraphs of G and H with equivalent bioactivity.*

Starting from this informal description, we introduce clearly defined graph-theoretical problems extending the maximum common subgraph paradigm. Since scalability is a critical concern, algorithmic aspects and complexity results must be taken into account and related to the specific properties of molecular graphs. These graphs are almost always planar and often outerplanar [18]. Since the number of bonds per atom is limited, the vertex degrees are bounded. It can be observed that all the graphs representing small molecules have a small tree width. The *tree width* of a graph essentially measures the similarity of a graph to a tree structure. Trees have tree width 1, and graphs that can be constructed via parallel or serial merges (series-parallel graphs) have tree width 2. Typically, molecular graphs have vertex and edge attributes that are either discrete labels or numerical values.

We proceed with a discussion of similarity approaches based on the maximum common subgraph paradigm and the specific challenges when applied to molecular graphs. Then, new graph-based methods are introduced, which address these challenges as part of the maximum similar subgraph problem.

3.1 Challenges and Approaches in Comparing Molecular Graphs

The *maximum common subgraph problem* is to find a common subgraph in two given graphs of maximum size. In the domain of cheminformatics, the maximum common subgraph problem has been extensively studied [12,44,50]; see [28 SPP] for a recent survey. In this domain, it is often referred to as the maximum or *largest common substructure problem*. This problem is known to be $\mathcal{NP}$-hard. With trees as input and output, the problem was shown to be polynomial-time solvable [35], but bioactive molecular graphs are not trees in general. The fact that they are mostly outerplanar does not directly lead to efficient algorithms since the maximum common subgraph problem restricted to outerplanar graphs remains $\mathcal{NP}$-hard. Instead of developing maximum common subgraph algorithms for more general graph classes, which has been proven difficult, a different approach represents molecules simplified as trees [41]. Then, vertices typically represent groups of atoms, and their comparison requires rating the similarity of two vertices by a weight function. However, similar to fingerprints, this goes along with a loss of information. Especially when comparing to large molecular databases, e.g., to rank the molecules regarding their similarity, this loss of information can lead to a reduced distinctiveness [21 SPP].

For molecular graphs, there is a variation of the maximum common subgraph problem of high practical relevance. There, the block (i.e.,connected set of molecular rings) and bridge (i.e.,molecular chain) structure of the input graphs must be retained by the common subgraph, i.e.,the underlying subgraph isomorphism is *block-and-bridge preserving* (BBP). This variation is denoted *block-and-bridge preserving maximum common subgraph problem* (BMCS) and requires the common subgraph to be connected and the associated subgraph isomorphisms to be BBP. There is a variant of the problem where the subgraphs are not necessarily (vertex) induced. This edge induced variant is denoted as *BMCES*. For both variants, it has been shown that they yield meaningful results for cheminformatics and are computable in polynomial time on outerplanar graphs [50, 21 SPP, 10 SPP].

In [50], a BMCES algorithm was proposed for outerplanar molecular graphs. Contrary to the original claim of $\mathcal{O}(n^{2.5})$ for a graph with n vertices, the algorithm allows no better bound than $\mathcal{O}(n^4)$ on its running time [30 SPP]. A previously suggested algorithm regarding the BMCS problem for input graphs with tree width $k \leq 2$ has a running time of $\mathcal{O}(n^6)$ [32 SPP]. In the case of outerplanar input graphs, the running time can be reduced to $\mathcal{O}(n^5)$. An essential part of this algorithm is the decomposition of the graphs into their *BC*- and *SPQR-trees*, which decompose the graphs into their biconnected and three-connected components. A maximum solution is then computed via a dynamic programming approach on the blocks and bridges.

Following the above result, we presented a faster approach tailored to outerplanar graphs [10 SPP]. On such graphs G and H, this algorithm achieves a running time of $\mathcal{O}(|G|\,|H|\,\Delta(G,H))$, where $\Delta(G,H) = 1$ if G or H is biconnected; otherwise, $\Delta(G,H) = \min\{\Delta_C(G), \Delta_C(H)\}$, where $\Delta_C(G)$ and $\Delta_C(H)$, respectively, is the maximum degree of all cutvertices in G and H, respectively. For outerplanar molecular graphs, the time bound is $\mathcal{O}(|G|\,|H|)$ since they have bounded degree. The first major ingredient is a fast dynamic programming approach on the BC-trees of the input graphs, where we exploit the similarity between the maximum weight matching instances that we have to solve [9 SPP]. Here, we use an algorithm for the maximum weight matching problem with a running time depending on the smaller vertex set. The second ingredient is a quadratic time algorithm to find a biconnected maximum common subgraph between two blocks b_1 and b_2. This is realized by enumerating all maximal (with respect to inclusion) biconnected common subgraphs between the two blocks. Each maximal solution C can be computed in time $\mathcal{O}(|C|)$. The total size of all maximal solutions per block pair (b_1, b_2) is $\mathcal{O}(|b_1|\,|b_2|)$; hence the total algorithm's running time is $\mathcal{O}(|G|\,|H|)$. Along the edges and vertices with different labels, the maximal solutions are split into smaller biconnected components. Among all those components, we keep one of maximum size.

For non-outerplanar graphs, we use a clique reduction to compute biconnected maximum common subgraphs between two blocks if at least one is not outerplanar. In the reduction, we enumerate c-cliques as presented in [8,27]. Among them, we keep a biconnected c-clique of maximum size. This approach reduces the practical running time compared to a pure clique-based algorithm operating on the whole graphs since the computational demanding clique problem must be solved for small components only. In contrast to the BMCES algorithm of [50], the above-described technique enables our algorithm to compute a solution for any two molecular graphs and lower the practical running time for graphs with multiple blocks, even if they are not outerplanar.

We evaluated the practical running time of our algorithm [10 SPP] by comparing it to the BMCES algorithm from [50]. In our experiments, we used a dataset of 29 000 randomly chosen pairs of outerplanar molecular graphs from the NCI Open Database, GI50[2], with an average of 22 vertices (atoms) and a maximum of 104 vertices. Our algorithm outperformed the competitor by a factor of 84 on average. The experimental results align with our theoretical correction [30 SPP] of the running time analysis given in [50]. It should be noted that the BMCES algorithm is already much faster than a general clique-based MCS algorithm [50]. Our BMCS algorithm outperforms such a general algorithm by several orders of magnitudes. The practical differences of the results w.r.t. the vertex and edge induced variants is marginal, and we observed a disagreement in only 0.4% of the comparisons.

While our basic BMCS algorithm is fast in theory and practice, the primary goal is to find a meaningful common subgraph. It was observed that allowing disconnected common subgraphs improves the validity, given that the connected components are arranged consistently in both graphs [34,51]. However, solving the general disconnected variant is $\mathcal{NP}$-hard even in trees. Moreover, small variations of the chemical elements (vertex labels) might be tolerated. We tackle these challenges in the next subsection.

3.2 Maximum Similar Subgraph Based Similarities for Molecules

This subsection presents several problem fields where the classical MCS definition is too strict w.r.t. molecular bioactivity. We show how these problem fields can be theoretically approached under the MSS definition and how they can be solved programmatically by integration in the MCS algorithms. Subsequently, we evaluate our MSS approach in comparison with several established molecular similarity measures.

From a chemical point of view, the two drugs shown in Fig. 1 are almost identical and are expected to have nearly identical properties w.r.t. bioactivity. However, an MCS-based comparison would interpret a large part of the molecules as different due to the nitrogen switch in the bicyclic ring system. In other words, the exchange of a nitrogen and carbon atom in an aromatic ring should influence the molecular similarity only to a small extend under the maximum similar subgraph problem definition. In addition,

[2] http://cactus.nci.nih.gov.

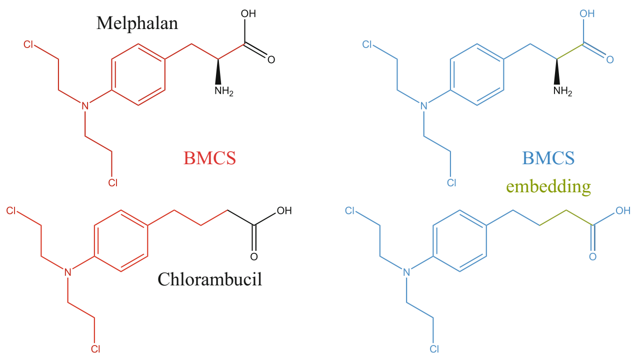


Fig. 4. Molecular graphs of Melphalan (top) and Chlorambucil (bottom). The BMCS on the left (red) maps less vertices than the BMCS embedding on the right (blue, green). The atoms on the right side (O, O, H) may be added to the embedding by mapping the green paths to each other. (Color figure online)

atom types like aromatic nitrogen or carbon can be grouped by their properties and such atom type groups can be used as representation instead. Thus, by softening the matching constraints in the MSS problem, a much larger substructure should be identified in the two molecules in comparison to the MCS approach. This problem can be solved with an atom type group representation [39] and a score in the range $[0,1] \cup \{-\infty\}$ to group mappings (mapping of vertices with atom type group labels), where $\{-\infty\}$ forbids the mapping. Hence, the objective is to maximize the weight of all mapped groups instead of the number of mapped vertices. The complete weight matrix is listed in Table III.2.3 of [19].

Additionally, we allow the mapping of disjoint paths of bridges (more precisely, the path's endpoints while skipping the inner vertices) to each other [11 SPP] in our MSS approach, i.e.,we allow some kind of disconnection. We denote this technique *embedding*, following [17]. This is useful, e.g., if two molecules differ only in the length of a chain connecting similar or identical substructures. To prevent arbitrary long paths, we introduce a linear penalty depending on the length of such paths. An example of two molecular graphs that profit from the described approach is depicted in Fig. 4.

In summary, we developed an algorithm applicable to molecular graphs that addresses the maximum similar subgraph problem by *(i)* using the established BMCS concept, *(ii)* allowing disconnectivity by mapping paths to edges, and *(iii)* supporting weight functions between labels. Moreover, our algorithm is efficient in theory and practice for the vast majority of molecular graphs.

In order to evaluate the quality of our MSS approach, we used a similar setup as in [40] and compared it to state-of-the-art chemical fingerprint methods. Our main question was whether the MSS approach produces meaningful results when used to rank molecules. In the following, we present the key evaluation results for the single-assay benchmark, which consists of rather similar molecules that have been ranked by the authors w.r.t. decreasing order of activity.

First, we analyzed different layers to represent the molecules. Among them are the chemical elements representation (e.g., N for nitrogen) and the file conversion (fconv) atom type groups [39]. We discovered that the latter representation based on the weight matrix of Table III.2.3 of [19] performed best for the single-assay benchmark. As similarity coefficient, we used Bunke and Shearer's [43], which performed best among the tested ones. It is defined as $W/\max\{k,l\}$, where W is the weight of the maximum common subgraph, and k, l are the sizes of the input graphs.

Compared to other methods, the very popular ECFP4 fingerprint showed the best match with the reference ranking followed by our MSS embedding approach. This is followed by RDKit7 (fingerprint of all subpaths up to path length 7), MSS without embedding, RDKit6, and the BMCS approach. Other fingerprint methods ranked in between. Extended-Connectivity Fingerprints (ECFPs) capture the neighborhood of the non-hydrogen atoms in circular layers up to a given diameter (e.g., 4 in the case of ECFP4). Thus, their features, similarly to the MSS, also represent the presence of particular small substructures. However, the advantage of our MSS approach is that it explicitly computes the similar substructures of the molecules and a concrete mapping between the atoms (vertices). It also achieved a high distinctiveness between the results, which is important to virtually screen large (big data) molecular libraries. The additional feature of mapping disjoint paths to each other showed improved results on the ranking benchmarks. More detailed results, as well as additional tests, can be found in [19].

4 Clustering Analysis

As mentioned in the introduction, clustering is used for a variety of use cases in drug discovery. In the following, we will focus on the task to cluster large scale molecular datasets of labeled graphs. An application of the presented approach is given in Sect. 5.3.

Definition 2. *A* clustering *of a graph dataset —i.e.,a multiset of labeled graphs— $\mathscr{X}$ is a partition $\mathscr{C} = \{C_1,\ldots,C_n\}$ of $\mathscr{X}$, that maximizes cluster homogeneity and often separation.*

The concrete definitions of homogeneity and separation differ in different clustering methods. Common measures for homogeneity are diameters or radii, density, or relative closeness to cluster representatives. Separation is often defined over the minimum distance between cluster elements or some aggregated cluster features. In contrast to homogeneity, separation is not always considered by clustering algorithms. For example, it is challenging to find a suitable definition of separation for projected clustering algorithms, since each cluster is linked to its own subspace and by that is incomparable to other clusters. Meta algorithms can be used to tune some of the clustering algorithms that do not optimize separation directly to achieve well-separated clusterings. For example, the number of clusters can be used as such a tuning parameter for the classical k-means algorithm [56].

4.1 Challenges and Approaches in Molecular Cluster Analysis

A major design decision for clustering algorithms is the data representation. Most classical clustering algorithms rely on vectorial data interpreted as points in some predefined space (e.g., $\mathscr{R}^\backslash$ with l^2-norm) or, more generally, on pairwise distances or kernels. Exchangeable distances or kernels are very versatile since it allows the clustering algorithm to be adapted to the specific clustering task. However, the explicit vector space representation with a fixed norm is often beneficial in terms of computational complexity. For example, it allows the explicit calculation of centroids, easy extraction of subspaces, or the use of binning. With these methods, it is often possible to avoid calculating a quadratic number of pairwise distances during the clustering process.

To fit a graph dataset into this models, the graphs must be either transformed into vectors (e.g., by using structural fingerprints or Weisfeiler-Lehman features [38]) or kernels/distances must operate directly on graph data (e.g., graph kernels [31] or the distance given in Sect. 3). However, while preferable in terms of generality, these generalized methods have weaknesses in the discussed domain.

First, both methods tend to produce (intrinsic) high-dimensional datasets [29]. While a high dimensionality may even be beneficial in supervised learning, intrinsic high dimensional datasets are linked to the so-called concentration effect [5] in the unsupervised setting. This effect causes the pairwise distances to lose their relative contrast, i.e.,the distances converge towards a common value. The concentration effect is closely related to a bad clusterability [1,57]. Furthermore, it causes metric index structures to be inefficient. Subspace or projected clustering methods, which are usually used in such a setting, come with an extra computational burden and are usually limited to vector space.

Second, the transformation to reasonably sized vectors is lossy and non-reversible. This causes the clustering results to be hard to interpret since cluster features, centroids, or subspaces are not in the application domain. Thus, these methods fail to provide a domain specific explanation about cluster commonalities.

As a consequence of these issues, structural clustering methods have been developed, which provide cluster descriptions or interpretations directly in the graph domain. This is accomplished by various constructs, including subgraph isomorphisms, (maximum) common subgraphs [6], frequent subgraphs [57], graph edit operations [23], and set medians [14,23]. For example, a cluster description can be given in the form of common subgraphs. Since most of these sub-problems are themselves challenging $\mathscr{N}\mathscr{P}$-hard problems, structural clustering algorithms are often limited to small datasets (e.g., [14,23,58]) or very special graph classes (e.g., trees [3]). As a consequence of the computational complexity, some of these clustering algorithms are hybrid approaches, which utilize approximations in vector space in order to map the results back into the graph domain. For example, the clustering algorithms in [14,23] calculate a cluster median in vector space but assign graphs to clusters w.r.t. the graph edit distance. A hierarchical k-means clustering in vector space is used as a starting point in [6]. It is later refined in order to increase the size of the common substructures. To the best of our knowledge and besides our own work, the only structural clustering algorithm for larger-scale datasets of general labeled graphs is presented in [54]. In this algorithm, each partition element of a vectorial pre-clustering is further partitioned with a struc-

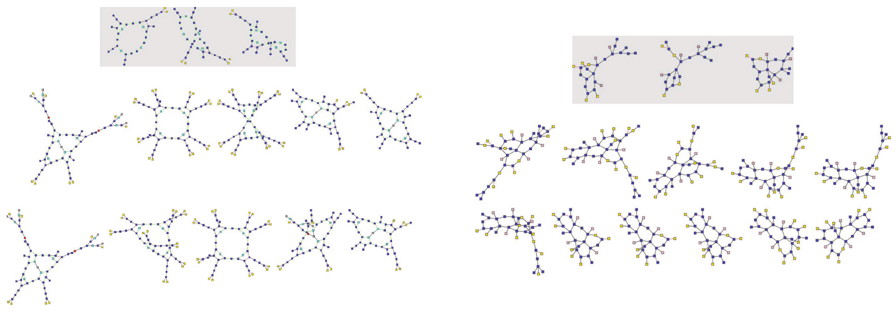


Fig. 5. Real world clusters with representatives (grey boxes) generated by *StruClus*. Colors represent node labels. (Color figure online)

tural algorithm. The pre-clustering is designed to only separate graphs that are also separated by the structural variant with a high probability if the structural clustering would be applied to the whole dataset.

4.2 *StruClus*: Scalable Structural Graph Set Clustering

StruClus [49 SPP] is a structural projected clustering algorithm that is tailored towards our setting of large-scale datasets ($\gg 10^6$ graphs) of small labeled graphs (druglike molecules are limited in their maximum size for biological reasons). Its linear runtime w.r.t. to the dataset size, the usage of various sampling strategies, and a parallelizable algorithm design make *StruClus* scalable and very fast in practice. It incorporates homogeneity and separation constraints for high-quality results.

A central concept of *StruClus* is the usage of cluster representatives sets $\mathscr{R}(C)$ for each $C \in \mathscr{C}$ (cf. Fig. 5 for a real-world example) that contain frequent subgraphs of the cluster members. They are beneficial in terms of computational complexity since they enable graph-cluster comparisons without looking at the cluster members (similar to the concepts of centroids or medoids). Additionally, they lead to human interpretable clusters by explaining the cluster content in the application domain.

The main objective of *StruClus* is to maximize homogeneity in the sense that the large fraction of the nodes and edges of the cluster members are covered by some subgraph isomorphism from the representatives. Similar to the classical k-means algorithm, this is achieved by an iterative optimization procedure that updates the representatives and re-assigns the cluster members to the best fitting cluster. However, the number of clusters is not pre-defined but adapted to the dataset structure with the help of cluster splitting operations on inhomogeneous clusters. Additionally, clusters with similar representatives are merged in order to maintain a well separated clustering.

Performance-wise, the major challenge lies in the discovery of suitable representatives $\mathscr{R}(C)$ for each cluster C. Since the number of frequent subgraphs may be exponential w.r.t. the maximal graph size in the cluster, *StruClus* utilizes a randomized maximal frequent subgraph sampling method. This is implemented by a random exploration of the frequent subgraphs of each C, which form a meet-semilattice with partial ordering derived from the sub and supergraph relation (cf. Fig. 6). Each random exploration starts

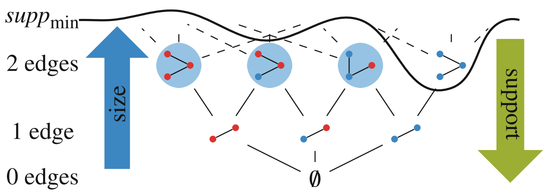


Fig. 6. Example for a meet-semilattice of subgraphs ordered by the subgraph isomorphism relation. Node colors indicate labels. Maximal frequent subgraphs are marked with a blue background color. (Color figure online)

with the empty graph and moves up in the lattice until a maximum frequent subgraph is reached. Since the support is monotonically decreasing for the supergraph relationship, it is possible to prune the search space with the minimum support threshold $supp_{\min}$.

The above-described maximum frequent pattern sampling is complemented with a new error bound stochastic sampling strategy over the cluster members to determine whether a graph pattern is frequent. A subset of the maximal frequent subgraphs given by this twofold sampling procedure is then selected by ranking the frequent subgraphs w.r.t. to the above homogeneity criteria.

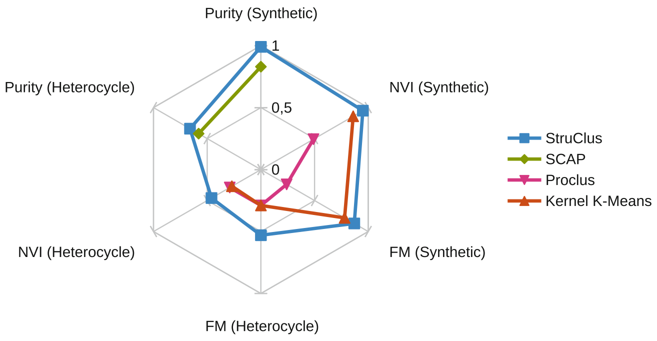
In comparison with structural clustering competitors, such as [14,23,53,54,58], *StruClus* is able to raise the maximum dataset size by multiple orders of magnitude, reaching into the domain of large-scale *de novo* databases. At the same time, *StruClus* outperforms structural competitors with a suitable performance for medium to large-scale datasets in terms of quality. Figure 7 shows an extract of an in-depth evaluation given in [49 SPP] w.r.t. to quality and performance on a real-world dataset (heterocycle) and a synthetic dataset. The heterocycle dataset consists of composed molecules classified by their reaction types. The synthetic dataset has common subgraphs for a class of graphs and is used to perform analysis with varying parameters. In Sect. 5.3, we present a real-world use case of *StruClus*.

5 Rational Drug Design Applications

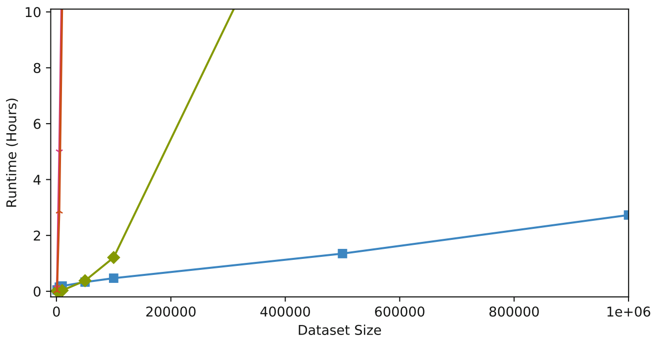
In this section, we present successful applications of the above approaches. Additionally, we present our tool Scaffold Hunter, that brings the scientific findings into the realm of practical drug design.

5.1 Scaffold Hunter

Scaffold Hunter [26,48 SPP] is open-source software for the analysis and visualization of molecular data with the aim to support the user in elucidating structure-activity-relationships. To this end, it features several structural classification schemes with dedicated visualizations and techniques to indicate chemical properties such as biological activity, e.g., by mapping values to colors, cf. Fig. 8. A fundamental structure-based concept is based on common core structures, so-called *scaffolds*, which can be organized hierarchically in a scaffold tree [52]. This approach forms the basis for several



(**a**) Quality measured with Normalized Variation of Information (NVI), Fowlkes-Mallows (FW) and Purity. Higher is better.



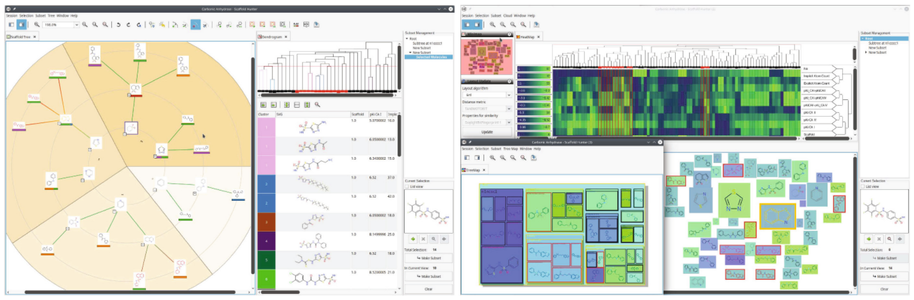
(**b**) Runtime scaling on the synthetic dataset.

Fig. 7. *StruClus* evaluation in comparison with SCAP [54], Proclus [2], and Kernel K-Means [16]. Graphlet –i.e.,small induced subgraph– frequencies are used for Proclus and Kernel K-Means.

views, which show the scaffold tree in a radial layout, in the form of a tree map or a set of scaffolds as a molecule cloud [13]. The view is inspired by the popular *word cloud* method, where the importance of words is indicated by their size. Here, scaffolds are scaled according to the number of molecules in the dataset containing them.

Following a different concept, structure-based hierarchical clustering is supported by means of chemical fingerprint similarity. Specifically for very large data sets, we have developed a heuristic method based on metric indexing [29]. The result can be visualized as a dendrogram that can be linked to a table or a heatmap. The heatmap visualizes property values in a matrix using color coding, where the columns are ordered in accordance with the dendrogram. This allows identifying whether chemical properties align with the structural similarity.

Several publications have shown that Scaffold Hunter is useful in various research tasks such as scaffold hopping, target prediction, chemical space analysis, and natural product simplification [7,26,33,45,21 SPP].



(a) Scaffold tree and dendrogram view (b) Heatmap, treemap and cloud view

Fig. 8. Scaffold Hunter allows to visualize molecular data in various linked views.

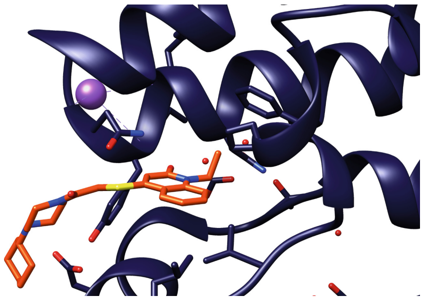


Fig. 9. Co-crystal structure of BRD4 in complex with one of the identified novel inhibitors (6g0e@pdb).

5.2 BRD4

In this study, an unexpected similarity in ligand binding between the bromodomain-containing protein 4 (BRD4) and the peroxisome-proliferator activated receptor gamma (PPARγ) was identified. Both are important drug targets in cancer therapy, cardiovascular diseases, and inflammation processes [15, 24]. The starting point was a scaffold-focused analysis of bioactivity data using the command-line version of Scaffold Hunter [48 SPP]. This analysis revealed a bicyclic scaffold that can be found, amongst others, in known ligands for BRD4 and PPARγ. Compounds with similarity to known PPARγ ligands were subsequently selected and tested on BRD4. Interestingly, the hit rate, which means the number of actives on BRD4, was unexpectedly high. Some of the novel inhibitors were successfully co-crystallized. One example is shown in Fig. 9. Further analyses of both proteins support the discovery of an unexpected relationship between the two drug targets [21 SPP] because they also show a high similarity of their binding sites. Based on this result, it seems possible to develop a drug that modulates both proteins with synergistic effects. Such a dual modulator would have the potential to have implications for the prevention or treatment of resistances against BRD4 inhibitors, which could already be observed [42]. Thus, this study demonstrates the successful application of a graph-based method in a prospective drug discovery study.

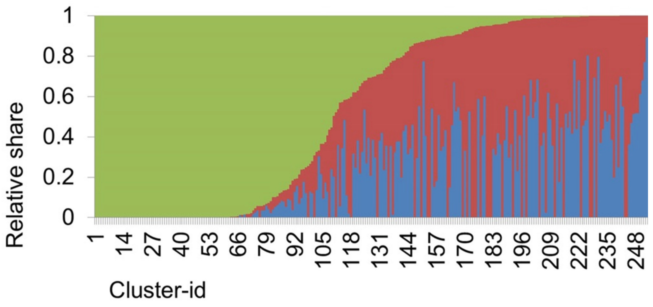


Fig. 10. Per cluster database distribution for the novelty analysis of CH*I*PMUNK. The green share is the MCR-CH*I*PMUNK sublibrary, blue is ChEMBL, red are commercially available compounds. The plot shows that some clusters are (almost) exclusively covered by CH*I*PMUNK. [taken from [22 SPP], printed with permission from Wiley] . (Color figure online)

5.3 Chipmunk

CH*I*PMUNK (CHemically feasible *In silico* Public Molecular UNiverse Knowledge base) [22 SPP] is a novel virtual library of small molecules which are synthesizable from purchasable reactants. The goal of such *de novo* libraries is the expansion of the known chemical and bioactivity space in order to enable virtual analytical processes to extract meaningful novel molecular structures, e.g., for drug discovery. The *in silico* simulated reactions are chosen such that they are synthesizable in reality with a high probability. Altogether, CH*I*PMUNK covers over 95 million compounds.

In the evaluation of CH*I*PMUNK, it was shown that the content of the library has interesting chemical properties and that the library covers previously undiscovered regions of the chemical and bioactivity space. The former aspect was analyzed using descriptor-based methods. It revealed that CH*I*PMUNK nicely covers the physicochemical space of protein modulators and protein-protein interaction modulators. *StruClus* (cf. Sect. 4.2) was used for the evaluation of the latter aspect, the novelty analysis. Additionally, *StruClus* itself was evaluated to prove that it creates useful clusterings w.r.t. to chemical properties (refer to [22 SPP] for further details). Thus, molecules of the same cluster exhibit similar chemical and biological properties with a high probability.

To analyze the novelty of CH*I*PMUNK, several libraries of commercially available compounds (ZINC [55], MolPort[3], and eMolecules[4]) as well as the large scale ChEMBL [4] bioactivity database were clustered in conjunction with CH*I*PMUNK. The former libraries serve as known chemical space, whereas the latter serves as known bioactivity space. The clustering revealed a large portion of clusters consisting purely of CH*I*PMUNK compounds (cf. Fig. 10 for an example).

Thus, it was displayed that CH*I*PMUNK encompasses regions that are uncovered by existing databases but yet with protein modulator or protein-protein interaction mod-

3 https://www.molport.com/.
4 https://www.emolecules.com/.

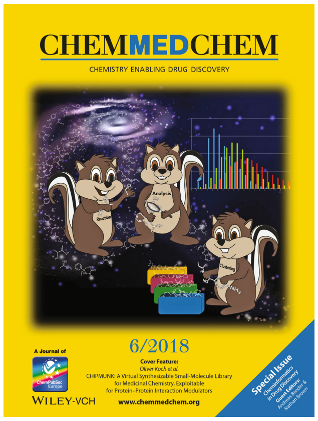

CHEMMEDCHEM
CHEMISTRY ENABLING DRUG DISCOVERY
Analysis
Reaction
Clustering
6/2018
Cover Feature:
Oliver Koch et al.
CHIPMUNK: A Virtual Synthesizable Small-Molecule Library for Medicinal Chemistry, Exploitable for Protein–Protein Interaction Modulators
www.chemmedchem.org
A Journal of
ChemPubSoc Europe
WILEY-VCH
Special Issue
Cheminformatics in Drug Discovery
Guest Editors:
Andreas Bender & Nathan Brown

Fig. 11. The Cover Feature shows three chipmunks involved in the creation, analysis, and clustering of the synthesizable virtual molecule library CH*I*PMUNK. Nearly 100 million compounds were generated with *in silico* reactions on accessible building blocks, and their descriptor profile was analyzed. [taken from [22 SPP], printed with permission from Wiley]

ulator like physicochemical properties. It can be concluded that CH*I*PMUNK offers the potential to contain future drugs.

The CH*I*PMUNK library is publicly available together with the clustering results. Areas of the chemical space –i.e.,clusters– that overlap with the ChEMBL library can be used to relate novel molecules given in CH*I*PMUNK to already known molecules from ChEMBL (in terms of structural similarity). This is helpful to relate already existing knowledge to the CH*I*PMUNK library. Thus, it may help in identifying the biological targets for the CH*I*PMUNK compounds.

6 Conclusion and Outlook

Graph-based methods for the analysis of molecular data sets are particularly appealing because they can reveal subtle structural differences and allow interpretation in terms of substructures. The complexity of the related graph-theoretical problems, however, makes their applications to large data sets challenging. We have developed new methods based on common substructures, which take the specific constraints in cheminformatics into account and exploit the properties of molecular graphs. Thereby, our techniques become efficient in both theory and practice. The application to molecular similarity search shows that our approach produces chemically meaningful rankings of molecules. Thus, it is well suited for virtual screening in large molecular databases. Moreover, we have developed a structural clustering algorithm, which represents clusters by common substructures and scales to very large databases with millions of molecules. Our methods have been proven to be useful in various research tasks in rational drug design. The success of our approaches has also been appreciated in 2018, when we were invited to the cover feature of the June issue of ChemMedChem (cf. Fig. 11).

During the writing of this survey, our project is still ongoing. Currently, we develop a distributed algorithm to mine representative sets of subgraphs for a variety of different use cases, including but not limited to the development of a fully distributed structural clustering algorithm. For this, the discussions and results within the SPP have been very useful (cf. Chap. 14).

Within our project, we have also developed other approaches to algorithmic data analysis. E.g., we have studied Graph Neural Networks (GNN) and their use to generate molecular representations for application in virtual screening approaches. Here, GNNs performed worse than fingerprint-based multilayer perceptrons, which questions the use of simple GNNs to obtain molecular representations [36 SPP, 37 SPP]. Future work will show if more complex graph-based representations will be able to replace molecular fingerprints as suitable input. For these learning approaches, it will be helpful to also learn with large generated graph families (cf. Chap. 2 and Chap. 3). Together with Christian Schulz, we investigate the applicability of kernelization (cf. Chap. 5), i.e., the iterative reduction of the problem to smaller instances, to common subgraph problems in large graphs. Matching problems build a connection with Chap. 13, which also is concerned with life science applications. Jointly we have worked on new streaming algorithms approximating the bipartite matching problem.

Acknowledgement. This work was supported by the German Research Foundation (DFG), priority programme *Algorithms for Big Data (SPP 1736)*. At the start of this project, all authors were members at TU Dortmund University. Permission for reprinting Fig. 11 and Fig. 10 has been permitted by Wiley.

References

1. Ackerman, M., Ben-David, S.: Clusterability: a theoretical study. In: Dyk, D.A.V., Welling, M. (eds.) Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics. JMLR Proceedings, AISTATS 2009, vol. 5, pp. 1–8. JMLR.org, Clearwater Beach, Florida (2009). https://www.jmlr.org/proceedings/papers/v5/ackerman09a.html
2. Aggarwal, C.C., Procopiuc, C.M., Wolf, J.L., Yu, P.S., Park, J.S.: Fast algorithms for projected clustering. In: Delis, A., Faloutsos, C., Ghandeharizadeh, S. (eds.) COMAD, ACM SIGMOD 1999, pp. 61–72. ACM Press, Philadelphia (1999). https://doi.org/10.1145/304182.304188
3. Aggarwal, C.C., Ta, N., Wang, J., Feng, J., Zaki, M.J.: Xproj: a framework for projected structural clustering of xml documents. In: Berkhin, P., Caruana, R., Wu, X. (eds.) Proceedings of the 13th International Conference on Knowledge Discovery and Data Mining, pp. 46–55. ACM Press, San Jose (2007). https://doi.org/10.1145/1281192.1281201
4. Bento, A.P., et al.: The ChEMBL bioactivity database: an update. Nucleic Acids Res. **42**(D1), D1083–D1090 (2013). https://doi.org/10.1093/nar/gkt1031
5. Beyer, K.S., Goldstein, J., Ramakrishnan, R., Shaft, U.: When is "nearest neighbor" meaningful? In: Proceedings of the 7th International Conference on Database Theory, ICDT 1999, pp. 217–235. Springer-Verlag, London (1999). https://doi.org/10.1007/3-540-49257-7_15, https://dl.acm.org/citation.cfm?id=645503.656271
6. Bocker, A.: Toward an improved clustering of large data sets using maximum common substructures and topological fingerprints. J. Chem. Inf. Model. **48**(11), 2097–2107 (2008)

7. Bon, R.S., Waldmann, H.: Bioactivity-guided navigation of chemical space. Acc. Chem. Res. **43**(8), 1103–1114 (2010). https://doi.org/10.1021/ar100014h
8. Cazals, F., Karande, C.: An algorithm for reporting maximal *c*-cliques. Theor. Comput. Sci. **349**(3), 484–490 (2005). https://doi.org/10.1016/j.tcs.2005.09.038

9 SPP. Droschinsky, A., Kriege, N.M., Mutzel, P.: Faster Algorithms for the Maximum Common Subtree Isomorphism Problem. In: MFCS 2016, Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, vol. 58, pp. 33:1–33:1 (2016). https://doi.org/10.4230/LIPIcs.MFCS.2016.33, https://drops.dagstuhl.de/opus/volltexte/2016/6447

10 SPP. Droschinsky, A., Kriege, N., Mutzel, P.: Finding largest common substructures of molecules in quadratic time. In: Steffen, B., Baier, C., van den Brand, M., Eder, J., Hinchey, M., Margaria, T. (eds.) SOFSEM 2017. LNCS, vol. 10139, pp. 309–321. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-51963-0_24

11 SPP. Droschinsky, A., Kriege, N.M., Mutzel, P.: Largest weight common subtree embeddings with distance penalties. In: Potapov, I., Spirakis, P.G., Worrell, J. (eds.) MFCS 2018. LIPIcs, vol. 117, pp. 54:1–54:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.MFCS.2018.54

12. Ehrlich, H.C., Rarey, M.: Maximum common subgraph isomorphism algorithms and their applications in molecular science: a review. Wiley Interdisc. Rev. Comput. Molec. Sci. **1**(1), 68–79 (2011). https://doi.org/10.1002/wcms.5
13. Ertl, P., Rohde, B.: The molecule cloud - compact visualization of large collections of molecules. J. Cheminf. **4**(1), 12 (2012). https://www.jcheminf.com/content/4/1/12
14. Ferrer, M., Valveny, E., Serratosa, F., Bardají, I., Bunke, H.: Graph-based *k*-means clustering: a comparison of the set median versus the generalized median graph. In: Jiang, X., Petkov, N. (eds.) CAIP 2009. LNCS, vol. 5702, pp. 342–350. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03767-2_42
15. Ferri, E., Petosa, C., McKenna, C.E.: Bromodomains: structure, function and pharmacology of inhibition. Biochem. Pharmacol. **106**, 1–18 (2016). https://doi.org/10.1016/j.bcp.2015.12.005
16. Girolami, M.A.: Mercer kernel-based clustering in feature space. IEEE Trans. Neural Netw. **13**(3), 780–784 (2002). https://doi.org/10.1109/TNN.2002.1000150
17. Gupta, A., Nishimura, N.: Finding largest subtrees and smallest supertrees. Algorithmica **21**, 183–210 (1998). https://doi.org/10.1007/PL00009212
18. Horváth, T., Ramon, J., Wrobel, S.: Frequent subgraph mining in outerplanar graphs. Data Min. Knowl. Disc. **21**(3), 472–508 (2010). https://doi.org/10.1007/s10618-009-0162-1
19. Humbeck, L.: Betrachtung der Ähnlichkeit von niedermolekularen Verbindungen unter Berücksichtigung der biologischen Aktivität. Dissertation, TU Dortmund University (2019)

20 SPP. Humbeck, L., Koch, O.: What can we learn from bioactivity data? chemoinformatics tools and applications in chemical biology research. ACS Chem. Biol. **12**(1), 23–35 (2017). https://doi.org/10.1021/acschembio.6b00706, pMID: 27779378

21 SPP. Humbeck, L., Pretzel, J., Spitzer, S., Koch, O.: Discovery of an unexpected similarity in ligand binding between BRD4 and PPARγ. ACS Chem. Biol. **16**(7), 1255–1265 (2021). https://doi.org/10.1021/acschembio.1c00323

22 SPP. Humbeck, L., Weigang, S., Schäfer, T., Mutzel, P., Koch, O.: CHIPMUNK: a virtual synthesizable small-molecule library for medicinal chemistry, exploitable for protein-protein interaction modulators. Chem. Med. Chem. (2018). https://doi.org/10.1002/cmdc.201700689

23. Jouili, S., Tabbone, S., Lacroix, V.: Median graph shift: a new clustering algorithm for graph domain. In: 20th International Conference on Pattern Recognition, pp. 950–953 (2010). https://doi.org/10.1109/ICPR.2010.238

24. Kersten, S., Desvergne, B., Wahli, W.: Roles of PPARs in health and disease. Nature **405**(6785), 421–424 (2000). https://doi.org/10.1038/35013000
25. Kim, S., et al.: PubChem in 2021: new data content and improved web interfaces. Nucleic Acids Res. **49**(D1), D1388–D1395 (2020). https://doi.org/10.1093/nar/gkaa971
26. Klein, K., Koch, O., Kriege, N., Mutzel, P., Schäfer, T.: Visual analysis of biological activity data with Scaffold Hunter. Molec. Inf. **32**(11–12), 964–975 (2013). https://doi.org/10.1002/minf.201300087
27. Koch, I.: Enumerating all connected maximal common subgraphs in two graphs. Theor. Comput. Sci. **250**(1–2), 1–30 (2001). https://doi.org/10.1016/S0304-3975(00)00286-3

28 SPP. Koch, O., Kriege, N.M., Humbeck, L.: Chemical similarity and substructure searches. In: Ranganathan, S., Gribskov, M., Nakai, K., Schönbach, C. (eds.) Encyclopedia of Bioinformatics and Computational Biology, pp. 640–649. Academic Press, Oxford (2019). https://doi.org/10.1016/B978-0-12-809633-8.20195-7, https://www.sciencedirect.com/science/article/pii/B9780128096338201957

29. Kriege, N., Mutzel, P., Schäfer, T.: Practical SAHN clustering for very large data sets and expensive distance metrics. J. Graph Algor. Appl. **18**(4), 577–602 (2014). https://doi.org/10.7155/jgaa.00338

30 SPP. Kriege, N.M., Droschinsky, A., Mutzel, P.: A note on block-and-bridge preserving maximum common subgraph algorithms for outerplanar graphs. J. Graph Algor. Appl. **22**(4), 607–616 (2018). https://doi.org/10.7155/jgaa.00480

31. Kriege, N.M., Johansson, F.D., Morris, C.: A survey on graph kernels. Appl. Netw. Sci. **5** (2020). https://doi.org/10.1007/s41109-019-0195-3

32 SPP. Kriege, N.M., Kurpicz, F., Mutzel, P.: On maximum common subgraph problems in series-parallel graphs. Eur. J. Comb. **68**, 79–95 (2018). https://doi.org/10.1016/j.ejc.2017.07.012

33. Lachance, H., Wetzel, S., Kumar, K., Waldmann, H.: Charting, navigating, and populating natural product chemical space for drug discovery. J. Med. Chem. **55**(13), 5989–6001 (2012). https://doi.org/10.1021/jm300288g, pMID: 22537178
34. Marialke, J., Körner, R., Tietze, S., Apostolakis, J.: Graph-based molecular alignment (GMA). J. Chem. Inf. Model. **47**(2), 591–601 (2007). https://doi.org/10.1021/ci600387r
35. Matula, D.W.: Subtree isomorphism in $O(n^{5/2})$. In: Alspach, B., Miller, D.P.H. (eds.) Algorithmic Aspects of Combinatorics, Annals of Discrete Mathematics, vol. 2, pp. 91–106. Elsevier (1978). https://doi.org/10.1016/S0167-5060(08)70324-8

36 SPP. Menke, J., Koch, O.: Using domain-specific fingerprints generated through neural networks to enhance ligand-based virtual screening. J. Chem. Inf. Model. **61**(2), 664–675 (2021). https://doi.org/10.1021/acs.jcim.0c01208

37 SPP. Menke, J., Massa, J., Koch, O.: Natural product scores and fingerprints extracted from artificial neural networks. Comput. Struct. Biotechnol. J. **19**, 4593–4602 (2021). https://doi.org/10.1016/j.csbj.2021.07.032

38. Morris, C., Rattan, G., Mutzel, P.: Weisfeiler and Leman go sparse: towards scalable higher-order graph embeddings. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) NeurIPS (2020). https://proceedings.neurips.cc/paper/2020/hash/f81dee42585b3814de199b2e88757f5c-Abstract.html
39. Neudert, G., Klebe, G.: fconv: format conversion, manipulation and feature computation of molecular data. Bioinform. **27**(7), 1021–1022 (2011). https://doi.org/10.1093/bioinformatics/btr055
40. O'Boyle, N., Sayle, R.: Comparing structural fingerprints using a literature-based similarity benchmark. J. Cheminf. **8** (2016). https://doi.org/10.1186/s13321-016-0148-0
41. Rarey, M., Dixon, J.S.: Feature trees: a new molecular similarity measure based on tree matching. J. Comput.-Aided Molec. Des. **12**, 471–490 (1998). https://doi.org/10.1023/A:1008068904628

42. Rathert, P., et al.: Transcriptional plasticity promotes primary and acquired resistance to bet inhibition. Nature **525**(7570), 543–547 (2015). https://doi.org/10.1038/nature14898
43. Raymond, J., Willett, P.: Effectiveness of graph-based and fingerprint-based similarity measures for virtual screening of 2d chemical structure databases. J. Comput.-Aided Molec. Des. **16**, 59–71 (2002). https://doi.org/10.1023/A:1016387816342
44. Raymond, J.W., Willett, P.: Maximum common subgraph isomorphism algorithms for the matching of chemical structures. J. Comput.-Aided Molec. Des. **16**(7), 521–533 (2002)
45. Renner, S., et al.: Bioactivity-guided mapping and navigation of chemical space. Nat. Chem. Biol. **5**(8), 585–592 (2009). https://doi.org/10.1038/nchembio.188
46. Rogers, D., Hahn, M.: Extended-connectivity fingerprints. J. Chem. Inf. Model. **50**(5), 742–754 (2010). https://doi.org/10.1021/ci100050t
47. Ruddigkeit, L., van Deursen, R., Blum, L.C., Reymond, J.L.: Enumeration of 166 billion organic small molecules in the chemical universe database GDB-17. J. Chem. Inf. Model. **52**(11), 2864–2875 (2012). https://doi.org/10.1021/ci300415d

48 SPP. Schäfer, T., Kriege, N.M., Humbeck, L., Klein, K., Koch, O., Mutzel, P.: Scaffold Hunter: a comprehensive visual analytics framework for drug discovery. J. Cheminf. **9**(1), 28:1–28:18 (2017). https://doi.org/10.1186/s13321-017-0213-3

49 SPP. Schäfer, T., Mutzel, P.: StruClus: scalable structural graph set clustering with representative sampling. In: Cong, G., Peng, W.-C., Zhang, W.E., Li, C., Sun, A. (eds.) ADMA 2017. LNCS (LNAI), vol. 10604, pp. 343–359. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69179-4_24

50. Schietgat, L., Ramon, J., Bruynooghe, M.: A polynomial-time maximum common subgraph algorithm for outerplanar graphs and its application to chemoinformatics. Ann. Math. Artif. Intell. **69**(4), 343–376 (2013). https://doi.org/10.1007/s10472-013-9335-0
51. Schmidt, R., Krull, F., Heinzke, A.L., Rarey, M.: Disconnected maximum common substructures under constraints. J. Chem. Inf. Model. **61**(1), 167–178 (2021). https://doi.org/10.1021/acs.jcim.0c00741, pMID: 33325698
52. Schuffenhauer, A., Ertl, P., Roggo, S., Wetzel, S., Koch, M.A., Waldmann, H.: The scaffold tree - visualization of the scaffold universe by hierarchical scaffold classification. J. Chem. Inf. Model. **47**(1), 47–58 (2007). https://doi.org/10.1021/ci600338x
53. Seeland, M., Berger, S.A., Stamatakis, A., Kramer, S.: Parallel structural graph clustering. In: ECML/KDD, Athens, Greece, pp. 256–272 (2011). https://doi.org/10.1007/978-3-642-23808-6_17
54. Seeland, M., Karwath, A., Kramer, S.: Structural clustering of millions of molecular graphs. In: Symposium on Applied Computing, SAC 2014, pp. 121–128. ACM, Gyeongju (2014). https://doi.org/10.1145/2554850.2555063
55. Sterling, T., Irwin, J.J.: Zinc 15-ligand discovery for everyone. J. Chem. Inf. Model. **55**(11), 2324–2337 (2015). https://doi.org/10.1021/acs.jcim.5b00559
56. Tibshirani, R., Walther, G., Hastie, T.: Estimating the number of clusters in a data set via the gap statistic. J. Roy. Stat. Soc.: Series B (Stat. Methodol.) **63**(2), 411–423 (2001). https://doi.org/10.1111/1467-9868.00293, https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/1467-9868.00293
57. Tsuda, K., Kudo, T.: Clustering graphs by weighted substructure mining. In: ICML, pp. 953–960. ACM (2006)
58. Tsuda, K., Kurihara, K.: Graph mining with variational dirichlet process mixture models. In: SDM, pp. 432–442. SIAM (2008). https://doi.org/10.1137/1.9781611972788.39

Recent Advances in Practical Data Reduction

Faisal N. Abu-Khzam[1], Sebastian Lamm[2], Matthias Mnich[3], Alexander Noe[4], Christian Schulz[5](✉), and Darren Strash[6]

[1] Lebanese American University, Beirut, Lebanon
faisal.abukhzam@lau.edu.lb
[2] Karlsruhe Institute of Technologie, Karlsruhe, Germany
sebastian.lamm@kit.edu
[3] Hamburg University of Technology, Institute for Algorithms and Complexity, Hamburg, Germany
matthias.mnich@tuhh.de
[4] University of Vienna, Vienna, Austria
alexander.noe@univie.ac.at
[5] Heidelberg University, Heidelberg, Germany
christian.schulz@informatik.uni-heidelberg.de
[6] Hamilton College, New York, USA
dstrash@hamilton.edu

Abstract. Over the last two decades, significant advances have been made in the design and analysis of fixed-parameter algorithms for a wide variety of graph-theoretic problems. This has resulted in an algorithmic toolbox that is by now well-established. However, these theoretical algorithmic ideas have received very little attention from the practical perspective. We survey recent trends in data reduction engineering results for selected problems. Moreover, we describe concrete techniques that may be useful for future implementations in the area and give open problems and research questions.

Keywords: Data reduction · Kernelization · Fixed-parameter algorithms · Algorithm engineering

1 Introduction

Many important real-world optimization problems are NP-hard: it is believed that no polynomial time algorithm exists that always finds an optimal solution. However, many NP-hard problems have been shown to be fixed-parameter tractable (FPT): large inputs can be solved efficiently and provably optimally, as long as some problem parameter is small. Over the last two decades, significant advances have been made in the design and analysis of fixed-parameter algorithms for a wide variety of graph-theoretic problems. This has resulted in an algorithmic toolbox that is by now well-established. However, these theoretical algorithmic ideas have received very little attention from the practical perspective. Until recently, few fixed-parameter algorithms have been implemented and tested on real data sets, and their practical potential is far from understood. Traditionally, algorithms are designed using simple models of problems and machines. In turn, important results are provable, such as performance guarantees for all possible inputs. This often yields elegant solutions being adaptable to many applications with predictable performance for previously unknown inputs.

In contrast to algorithm theory, taking up and implementing an algorithm is part of application development. Unfortunately, transferring results from theory to practice is a slow process and sometimes the theoretically-best algorithms perform poorly in experiments. Hence, practitioners often do not read research papers from the theoretical algorithms community. This causes a growing gap between theory and practice: Realistic hardware with its parallelism, memory hierarchies, etc. is diverging from traditional machine models. This gap is also partially due to the fact that the research community working on algorithmic problems is fairly separated. On the one hand, there are "hard core" algorithms researchers that are focused mainly on theoretical work and rarely participate in conferences in application areas. On the other hand, researchers of application areas publish their work in conferences and journals of their respective fields, and often do not visit theory conferences. In contrast to algorithm theory, algorithm engineering uses an innovation cycle where algorithm design based on realistic models, theoretical analysis, efficient implementation, and careful experimental evaluation using real-world inputs closes gaps between theory and practice and leads to improved application code and reusable software libraries (see www.algorithm-engineering.de). This yields results that practitioners can rely on for their specific application.

On the one hand, experimental results can trigger new theoretical questions and suggest new properties of inputs that are relevant parameters to use in theoretical analysis. On the other hand, the rich toolbox of parameterized algorithm theory offers a rich set of algorithmic ideas that are challenging to implement and engineer in practical settings. By applying techniques from fixed-parameter algorithms in nontrivial ways, algorithms can be obtained that perform surprisingly well on real-world instances for NP-hard problems. The viability of this approach has been demonstrated in recent years through the Parameterized Algorithms and Computational Experiments Challenge (PACE) [28,54,55,58], in which teams compete to solve real-world inputs using ideas from parameterized algorithm design. Many researchers from all over the world have participated in that challenge. Moreover, the viability of this approach has recently been demonstrated by a wide range of papers. Since the engineering part in the area has recently gained some momentum, we survey recent results and techniques that have started to bridge the gap between theory and practice that is currently observed in the area.

Theoretical Context. All known exact and deterministic algorithms that solve NP-hard problems require time that is at least super-polynomial in the total size of the input. However, some problems can be solved by algorithms that run in time which is exponential only in the size of a fixed parameter while polynomial in the size of the input; those are called *fixed-parameter algorithms*. Here, the parameterized problem can be solved efficiently for small values of the fixed parameter. Formally, a parameterized problem is a language $L \subseteq \Sigma^* \times \mathbb{N}$, where Σ is a finite alphabet. The second component is called the parameter of the problem. A parameterized problem L is *fixed-parameter tractable* if the question $(x,k) \in L$ can be decided by an algorithm in running time

$f(k) \cdot |x|^{\mathcal{O}(1)}$, where f is a computable function depending on k only. The corresponding complexity class is called FPT.

The *W hierarchy* [56] is an important hierarchy for the complexity of parameterized problems. A parameterized problem is in class $W[i]$, if we can transform every instance (x,k) to a decision circuit (a combinatorial circuit with only a single output gate) with *weft* at most i, such that the circuit outputs true if and only if $(x,k) \in L$. The weft of a combinatorial circuit is the maximum number of gates with more than two inputs on any path from input to output. Downey et al. [56] show that FPT $= W[0]$ and that $W[0] \subseteq W[1] \subseteq W[2] \subseteq \cdots \subseteq W[poly]$.

Fixed-parameter tractability is closely related to data reduction and kernelization. *Data reduction rules*, or simply *reductions*, reduce the size of a graph while retaining the ability to compute an optimal solution. A graph on which a collection of data reduction rules have been exhaustively applied is called a *reduced* graph. In kernelization, the reduced graph is called a *kernel* $\mathcal{K}$. More formally, given an binary encoded instance $(x,k) \in \{0,1\}^* \times \mathbb{N}$ of some parameterized problem L, a *kernelization* for L produces an instance (x',k') in polynomial time that satisfies: $(x',k') \in L \Leftrightarrow (x,k) \in L$ and $|x'| + k' \leq f(k)$ where f is a computable function. Note that f only depends on the problem parameter k. So roughly speaking, kernelization can be thought of as a preprocessing routine that reduces a given problem instance to its "most difficult part". The function f measures the kernel size. If $f(k) = \mathcal{O}(k^c)$ for some constant c then the kernel is called polynomial kernel, and we say the problem admits a polynomial kernel.

Many exact algorithms for parameterized problems combine these data reductions with *branching*. These algorithms are called *branch-and-reduce algorithms*. First, the algorithm aims to reduce the graph size by exhaustively applying reduction rules until there are no further data reductions possible or they are prohibitively expensive. Then, the algorithm picks an edge $e \in E$ (or a vertex $v \in V$, depending on the problem) and *branches* the problem into multiple subproblems, one subproblem for each potential state of e in regard to the problem. As an example, for the maximum cut problem or the multiterminal cut problem, branching creates two subproblems, one in which e is part of the cut and one in which e is not part of the cut. The branch-and-reduce algorithm then continues to apply reduction rules to both of these subproblems and continues branching when there are no further reductions possible. The branch-and-reduce algorithm returns the best result over all branches.

Organization. The rest of the paper is organized as follows. We first survey recent data reduction engineering results for selected NP-hard problems, and then for problems in P. We then describe concrete techniques that may be useful for future implementations in the area. Lastly, we give open problems and research questions.

2 Recent Advances for NP-Hard Problems

2.1 Maximum Independent Set and Minimum Vertex Cover

Given an undirected graph $G = (V,E)$, the goal of the *maximum independent set* (MIS) problem is to compute a set of vertices $I \subseteq V$ such that (1) no two vertices in I are adjacent to one another, (2) the set I has maximum cardinality among all such sets. The complement of an independent set I, $V \setminus I$, is called a *vertex cover*. The MIS problem and the complementary problem of finding a minimum vertex cover (MVC) are well-studied NP-hard optimization problems [75] that attract both researchers and practitioners alike. Furthermore, there is no polynomial time algorithm that approximates the MIS size within a factor $\mathcal{O}(n^{1-\varepsilon})$ for any constant $\varepsilon > 0$, unless P = NP [173]. Finally, MIS is $W[1]$-hard [56] when parameterized by solution size k. This makes it unlikely that the problem is fixed-parameter tractable in k [56]. On the other hand, MVC is fixed-parameter tractable in solution size k [56].

Exact Approaches. In recent years, the bridge between theoretically efficient algorithms and their practical applicability has been significantly reduced. In particular, the branch-and-reduce paradigm, i.e., branching algorithms that use a wide variety of reduction rules, have been (1) shown to achieve theoretical running times that are among the best for both MIS and MVC [69, 170], and (2) are able to solve large real-world networks in practice [5]. However, most often the approaches used in practice only use a small subset of the reduction rules that have been proposed to achieve good theoretical running times.

Abu-Khzam et al. [4] introduced and analyzed the crown reduction rule (and the usage of data reduction rules in this context in practice). Even though the crown rule is not as powerful as the linear programming (LP)-based rule [133] when considering the worst-case size of the resulting kernel, they experimentally verified that it often performs as well as the LP-based rule and is significantly faster in many cases. Furthermore, they show that the LP-based rule is most useful for fairly sparse graphs and should be avoided for dense graphs, as it yields little to no reduction in size.

Later, Akiba and Iwata [5] were the first to show the practicality of the branch-and-reduce paradigm for MVC (and MIS) compared to other state-of-the-art approaches like branch-and-bound and branch-and-cut. Their algorithm uses a wide spectrum of reduction rules that form the foundation of much subsequent work. This includes both conceptually simple reduction rules like degree-1 and degree-2 vertex folding [69], as well as more complicated but practically significant rules like unconfined [169] and an LP-based rule [95, 133]. Many of these reduction rules work by removing vertices that are part of some MIS. We illustrate this by briefly covering the degree-1 and degree-2 vertex fold reduction rules: (1) In the degree-1 reduction rule (see Fig. 1) one removes vertices v of degree one (and their neighbors), as they are always in at least one MIS. To see this, note that v or its neighbor w must be in some MIS I, otherwise $I \cup \{v\}$ is an independent set of larger cardinality. If w is in I, one can obtain an independent set of the same size by removing w from I and adding v instead. (2) For the degree-2 vertex fold (see Fig. 2) one removes vertices v with exactly two neighbors u and w that are not adjacent to each other. In this case a new vertex v' is inserted and connected to the union of the neighborhoods of u and w yielding a reduction of the graph size by two vertices. Finally, if v' is part of an MIS I' of the reduced graph, then $I = (I' \setminus \{v'\}) \cup \{u, w\}$ is an MIS of the original graph. Otherwise, $I = (I' \setminus \{v'\}) \cup \{v\}$ is an MIS of the original graph. Using their branch-and-reduce algorithm, Akiba and Iwata were able to solve a large variety of instances including social networks, web graphs and road networks. A

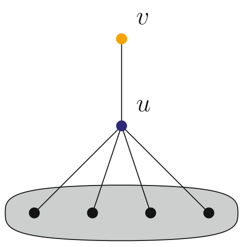


Fig. 1. Degree-1: Vertices v and u can be removed.

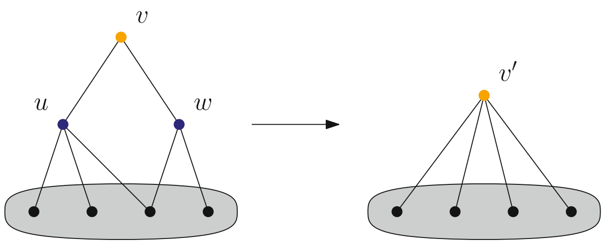


Fig. 2. Degree-2 vertex fold: Vertices v, u and w can be removed. In this case a new vertex v' is inserted.

similar approach that uses a quantum annealer to solve instances once they are small enough was recently presented by Pelofske et al. [140].

Although Akiba and Iwata [5] use a sophisticated set of reduction rules, Strash [155] showed that many of the more complicated rules are not necessary to compute an MIS in many large complex networks. Furthermore, the initial reduction rules applied to compute a reduced graph often have a bigger impact on performance, compared to further techniques used during the branch-and-reduce approach. Recently, Stallmann et al. [153] supported this idea by showing that networks G with a small normalized average degree ($\mathsf{nad}(G)$) can be efficiently handled by simple reduction rules. The $\mathsf{nad}(G)$ of a network G on n vertices is defined as the average degree of G normalized using a factor of $200/n$ if the average is larger than 20. Otherwise, if the average degree is at most 20, $\mathsf{nad}(G)$ is the same as the average degree of G. Additionally, the authors make use of the so-called degree spread t/b, where t is the degree at the 95th percentile and b at the 5th percentile. Based on these characteristics, the authors devise thresholds that indicate (1) if reductions should be used at all, (2) if more complex rules provide a significant benefit.

Open Problem 1. *What are graph characteristics and properties that determine the success of specific reduction rules?*

Recently, Hespe et al. [92] won the PACE Challenge 2019 vertex cover track by using a portfolio of exact approaches for MIS, MVC and maximum clique. In particular they use the reduction rules of Akiba and Iwata as an initial preprocessing step. Afterwards, an initial solution is computed using the state-of-the-art local search algorithm by Andrade et al. [7]. Finally, they switch between the branch-and-reduce algorithm of Akiba and Iwata [5] and the clique solver by Li et al. [119], which are applied to either the original graph or the graph resulting from the preprocessing step.

Heuristic Approaches. Reductions are also heavily used in many state-of-the-art heuristic approaches. Lamm et al. [114, 113 SPP, 150 SPP] use the same set of reductions originally used by Akiba and Iwata to develop an evolutionary algorithm that is able to compute high quality solutions for large graphs that are infeasible for branch-and-reduce. The authors use reductions for both preprocessing (to compute a kernel) and during the algorithm itself. In particular, they select vertices that are part of many highly fit individuals, which are independent sets, in their population. These vertices are then added to the resulting independent set, which includes removing them and their neighbors from the graph. Afterwards, reduction rules are applied and the evolutionary algorithm is called recursively on the resulting graph.

The idea of excluding a subset of vertices that are likely to be part of a high-quality independent set, is also explored by Gao et al. [73]. To select these vertices they perform multiple runs of a state-of-the-art local search algorithm (either NuMVC [33] or FastVC [34]). Vertices that are present in all resulting solutions are then added to the final solution and a new graph consisting of the remaining vertices and their corresponding edges is constructed. Afterwards, a final run of the local search on this graph is executed and its solution is combined with the previously removed vertices.

Dahlum et al. [51, 150 SPP] combine both simple exact reduction rules as well as inexact reductions with the ARW local search algorithm [7]. In particular, they remove cliques of up to size three (an exact reduction) and the top 1% high-degree vertices (an inexact reduction). The reasoning behind their inexact reduction is that high-degree vertices are not likely to be in a large independent set. Additionally, these vertices pose a significant bottleneck for local search. The authors also compare their algorithm against an algorithm that uses the data reduction rules of Akiba and Iwata as a preprocessing step. A similar preprocessing approach that only uses a subset of reduction rules is also presented by Cai et al. [37]. In particular, they use the degree-0, degree-1, degree-2 and domination rules.

Chang et al. [42] also make use of the idea of combining simple reduction rules that can be applied in (near-)linear time with an inexact reduction rule that removes high-degree vertices. For this purpose, they introduce the reducing-peeling framework that switches between the two types of reductions. Furthermore, they present a set of degree-2 path reductions that are special cases of the folding reduction. Combining these new rules with the degree-0, degree-1, dominance and an LP-based reduction rule, they propose an efficient preprocessing algorithm that is then combined with the ARW local search algorithm.

Open Problem 2. *Can one derive (near-)linear time special cases of the more complex reductions like the unconfined reduction that are not covered by existing reductions?*

In order to quickly achieve smaller reduced graphs than what is possible by using simple reduction rules, Hespe et al. [93, 150 SPP] provided the first shared-memory data reduction based on the rules of Akiba and Iwata. For this purpose they make use of both graph partitioning and parallel bipartite maximum matchings. The graph partitioning library KaHIP [148] is used to compute a partition of that graph which allows parallel execution of reduction rules that only need to check highly localized subgraphs, where bipartite maximum matchings are used to enable the parallel execution of the LP-based reduction rule. Furthermore, the authors present two speedup techniques for kernelization: (1) dependency checking that prunes applicability checks for certain reductions and (2) reduction tracking that stops their algorithm once the application of reduction rules only decreases the graph size by a negligible amount.

Open Problem 3. *Can the techniques used by Hespe et al. [93] be extended to a distributed memory setting? How can one efficiently apply reductions in distributed memory?*

Alsahafy and Chang [6] recently proposed an algorithm that combines the reducing-peeling framework with the exact clique solver MoMC by Li et al. [119]. Their algorithms splits reduction rules into two sets: ones that can be updated and applied incrementally (similar to Hespe et al. [93]), and ones that can not. Additionally, they continuously compute and maintain the connected components of the graph, which are then reduced individually. If a reduced component is small enough, it is then transformed into its complement and solved by MoMC. To ensure that components continue to get smaller, they use the same inexact reduction rule as Chang et al. [42] and then continue recursively on the resulting components. The authors also present a new exact reduction rule called the pyramid reduction.

Lastly, Lavallee et al. [118] evaluated a structural rounding approach for vertex cover. The main idea is to first edit a graph to a well-structured graph which can be solved more easily, and then apply a “lifting” algorithm to the partial solution to recover an approximation on the input network. Lavallee et al. find that their algorithm can outperform standard 2-approximation algorithms and that simpler lifting strategies are highly competitive with more sophisticated strategies.

Weighted MIS. Due to the significant practical results achieved for the unweighted case, there has been an increasing interest in generalizing these techniques for the weighted *maximum independent set* (WMIS) and *weighted minimum vertex cover* (WMVC) problems. For both problems, one is given an additional real-valued vertex weighting function $w : V \to \mathbb{R}^+$. In case of the WMIS problem one is then tasked with finding an independent set, such that the sum of the weights of its vertices is maximum among all possible independent sets. Analogously, for the WMVC one is tasked with finding a vertex cover of minimum weight.

Recently, Li et al. [120] used a set of four reduction rules during the initial construction phase of a local search algorithm. In particular, they use weighted reduction rules that are able to remove degree one and degree two vertices. They then use these reduction rules exhaustively in the beginning of their algorithm to obtain an improved initial solution. Their local search algorithm called NuMWVC is able to compute high quality solutions on a large variety of instances. This includes many instances commonly used for the unweighted case, which have been given vertex weights drawn from a uniform distribution. Since there are not many publicly available weighted instances, this is a common approach that is also used in other works [35,77,172,115 SPP]

Wang et al. [165] also make use of reduction rules for vertices with degree at most 2 as a preprocessing step for a branch-and-bound solver. Furthermore they evaluate different degree-based heuristics for selecting branching vertices and use pruning based on the best solution found so far.

Lamm et al. [115 SPP] proposed a practically efficient branch-and-reduce algorithm for the WMIS problem that is able to solve a large number of real-world instances. For this purpose they develop a comprehensive set of practically efficient reduction rules. These include both generalizations of previous weighted and unweighted reduction rules, as well as two "meta reductions" which serve as a general framework for the other rules. They use these rules to build a branch-and-reduce algorithm that uses many of the approaches that worked well in the unweighted case. In particular, they use local searches to compute initial solution which can be used for pruning, treat connected components individually and make use of dependency checking. Finally, they show that their reduction rules can be used to improve the performance of other state-of-the-art algorithms

Zheng et al. [172] propose an exact and heuristic approach that both make use of reduction rules for vertices of degree at most 2. Their exact approach is a branch-and-reduce algorithm that applies these reduction rules recursively. However, the authors do not provide any details on the bounds or branching strategies used during the algorithm. Their heuristic approach is inspired by the reducing-peeling framework of Chang et al. [42]. Thus, it exhaustively applies their reduction rules and subsequently removes high-degree vertices to extend the space of possible reductions.

Gellner et al. [77] proposed new practically efficient variants of the struction rule by Ebenegger et al. [59]. The struction is a reduction that is able to be applied to arbitrary vertices in a graph, but comes at the cost of potentially increasing the overall number of vertices. Thus the authors propose three new variants of the struction that aim to limit the number of newly created vertices. Furthermore, they derive practically efficient special cases of their reduction rules and use them as a preprocessing step in the branch-and-reduce solver of Lamm et al. [115 SPP]. The algorithm is able to produce the smallest-known reduced graphs, solves more instances than previous exact approaches and has a running time that is comparable to heuristic algorithms.

Open Problem 4. *Can other problems also benefit from reductions that may temporarily increase the graph size? If so, how much of an increase should be allowed to remain practical?*

2.2 Finding and Enumerating Maximum Cliques

Given an undirected graph $G = (V,E)$, the goal of the *maximum clique* (MC) problem is to compute a set of vertices $C \subseteq V$ such that (1) all vertices in C are adjacent to one another, (2) the set C has maximum cardinality among all such sets. As mentioned in the previous section, MC solvers are often used in the context of independent sets. This is due to the fact that a clique of G is an independent set in the complement graph $\bar{G} = (V,\bar{E})$ with $\bar{E} = \{\{u,v\} \mid u,v \in V \wedge \{u,v\} \notin E\}$. Thus, one can leverage maximum clique algorithms for finding independent sets by computing the complement graph. Since many algorithms for finding maximum independent sets aim to perform well on sparse graphs, the resulting complement graphs that need to be handled by clique algorithms will often be dense. Fortunately MC has been more extensively studied for dense instances than for sparse instances. Like MIS and MVC, finding a maximum clique is also an NP-hard optimization problem [75]. Furthermore, unless P=NP, there is no polynomial time algorithm that approximates the MC size within a factor $\mathcal{O}(n^{1-\varepsilon})$ for any constant $\varepsilon > 0$ [173]. Finally, MC is $W[1]$-hard [56] parameterized by solution size k, making it unlikely that the problem is fixed-parameter tractable in k. However, it is fixed-parameter tractable under different parameterizations, e.g., when parameterized with the degeneracy of the graph [64]. All previous observations also hold for the *maximum clique enumeration* (MCE) problem of enumerating all maximum cliques in a graph.

Eblen et al. [60] presents a maximum clique solver (MCF) that adapts some of the reduction rules that have already been shown to work well for MVC and MIS. In particular, their algorithm begins by greedily computing a large clique C which is then used as a lower bound in order to remove vertices of degree less than $|C|-1$ [1]. Next, they use an adaptation of the degree-0 reduction rule previously used in MVC algorithms, as well as a rule based on heuristic colorings [160] to remove additional vertices. The authors also investigate the use of other reduction rules including an adaptation of the degree-1 reduction rule used in MVC algorithms. Finally, they compare applying reduction rules as a preprocessing method for a branch-and-bound solver against running them in a branch-and-reduce solver. Their experiments indicate that the branch-and-reduce approach performs better on real-world genome data.

Eblen et al. [60] then use the previous MCF solver to develop several approaches for the maximum clique enumeration (MCE) problem based on the algorithm by Bron and Kerbosch [30]. In particular, they develop two reduction rules based on MCF: First, they propose a reduction rule that uses MCF to compute a maximum clique cover and removes vertices not adjacent to this cover. Second, they propose a second data-driven preprocessing rule that computes so-called essential vertices, i.e., vertices that are present in every maximum clique. Vertices that are not adjacent to these vertices are subsequently removed from the graphs. Their experiments indicate that this rule works particularly well on large transcriptomic graphs, that often have a small set of essential vertices. However, its performance degrades for networks that do not have a small set of essential vertices, e.g., for uniform random graphs.

Open Problem 5. *Can one give similar data-driven reduction rules for other types of networks, e.g., social networks or road networks?*

Verma et al. [164] propose another type of reduction rule based on k-communities. A k-community is defined as a subgraph $G' = (V', E')$ where each edge $\{u, v\} \in E'$ connects vertices that have at least k common neighbors in G'. Subsequently, a subset of vertices $V' \subseteq V$ is called a k-community if there is a k-community with vertex set V' in G. Note, that a clique of size k is a $(k-t)$-community for any $t \in \{2, \ldots, k\}$. They then derive a reduction rule which computes a lower bound on the clique size based on maximum $(k-2)$-communities and prune vertices with a smaller degree. They then combine this reduction rule with the k-core based approach of Pardalos and Resende [1] and show that the resulting algorithm works well for handling large low-density graphs.

Chang [40,41] notes that even though many real-world networks are usually sparse, MC has been more extensively studied for dense instances. Thus, the authors propose a branch-and-reduce algorithm that leverages the existing work on MC for dense instances by transforming an instance of MC over a sparse graph to instances of k-clique finding (KCF) over dense subgraphs. For this purpose, the authors iteratively compute small and dense subgraphs (so-called ego networks) that are then handled by a KCF solver. In order to reduce the size of the subgraphs that are handled by this solver, their algorithm uses a combination of well-known upper bounds and lightweight reduction rules. In particular, they use five reduction rules for KCF, most of which are targeted toward removing vertices of high degree. The authors also present a heuristic algorithm for MC, as well as a two stage approach for MCE that makes use of their exact algorithm to compute the size of the largest clique. Furthermore, they show that the reduction rules used for MC can also be adapted for MCE.

Weighted MC. Recently, Cai and Lin [36] proposed the first (and only) practical algorithm for the *(vertex-)weighted maximum clique* (WMC) problem that uses reduction rules. The WMC problem is a generalization of MC where one is given an additional real-valued vertex weighting function $w : V \to \mathbb{R}^+$. Subsequently, one is tasked with finding a clique, such that the sum of the weights of its vertices is maximal among all possible cliques. In order to solve WMC on large sparse graphs, Cai and Lin [36] interleave clique construction with reduction rules. To be more specific, they gradually add "beneficial" vertices to a clique using an approximation of the benefit of a vertex. This is done by computing the mean of a cost-efficient upper and lower bound for each vertex and then selecting vertices using a dynamic best from multiple selection [34]. Finally, if a new best clique is found, the graph is reduced using two reduction rules. Both rules make use of the fact that one is able to remove vertices where an upper bound on any maximum clique containing this vertex is smaller than the weight of the current best clique. For their rules, the authors then propose two different upper bounds that make use of the neighborhood of a vertex.

***k*-plexes.** A k-plex is a generalization of a clique where each vertex is allowed to have several missing connections, i.e., not every vertex has to be connected to all other vertices in the k-plex [151]. In particular, a k-plex is a subset $S \subseteq V$ such that the degree of every vertex in the induced subgraph $G[S]$ is at least $|S| - k$. Furthermore, $|S|$ is called the size of the k-plex and the *maximum k-plex problem* (MK) is that of finding a k-plex of maximum size.

Gao et al. [72] present multiple theoretical properties that allow the removal of vertices based on a lower bound on the maximum k-plex size. Based on these properties they propose four reduction procedures which are then used in a branch-and-reduce algorithm. In particular, they then use an extension of the algorithm by Jiang et al. [100] to compute an initial lower bound and then use this bound to exhaustively apply their linear-time vertex reduction and the more costly subgraph reduction rules for preprocessing. Afterwards they use different sets of reduction rules depending on the type of branch (selecting or discarding a vertex). The authors also present a type of targeted branching that aims to select vertices which lead to a larger reduction in size. The resulting algorithm is able to solve multiple previously infeasible real-world instances and is considerably faster than previous state-of-the-art solvers (e.g., [168]).

Open Problem 6. *Can targeted branching be used for other problems? For example, the most commonly used branching strategy for independent sets is degree-based and does not take any reduction rules into account.*

Conte et al. [46] investigated reduction rules for the problem of enumerating all maximum k-plexes. For this purpose, they introduce the concepts of coreness and cliqueness. Coreness states that vertices of a k-plex of size at least m must have a degree not smaller than $m-k$. Thus, vertices with a smaller degree can iteratively be removed, resulting in the computation of $(m-k)$-cores. Cliqueness states that every vertex of a k-plex of size at least m is part of a clique not smaller than $\lceil m/k \rceil$. Therefore, vertices with a degree less than $\lceil m/k \rceil$ can be removed from the graph. Furthermore, if one knows the size of the maximum clique ω the search space for the size of the maximum k-plex can be limited to $[\omega, \omega \cdot k]$. Based on these observations the authors then present an algorithm that begins by computing the size of a maximum clique. Afterwards a lower bound for the size of the maximum k-plex $p \in [\omega, \omega \cdot k]$ is guessed. If this guess turned out to be wrong (i.e., all k-plexes found are smaller than p), the interval bounds are updated and a new lower bound is guessed. Otherwise, all k-plexes with maximum size are returned. Their algorithm is able to reduce a large set of instances by up to 99% and achieves running times that are multiple orders of magnitude faster than previous approaches [14].

2.3 Maximum Cuts

The *max-cut* problem originates from important applications in physics and operations research [10]; therefore, it has long been the subject of engineering more and more sophisticated algorithms which solve large-scale instances arising in practice. In particular, max-cut is one of the few problems where engineers and practitioners alike are interested in finding optimal solutions (rather than just approximate ones). Formally, the max-cut problem takes as input an edge-weighted graph G and seeks a bipartition of the vertex set V of G into two disjoint parts, V_1 and V_2, which maximizes the weight of the edges which *cross* the bipartition, that is, edges whose one endpoint is in V_1 and the other endpoint is in V_2. The state of the art for max-cut though is that even after much effort, optimal solutions are still unknown for several benchmark instances. Those reasons are the key motivations for engineering effective, and efficient, kernelization rules. The objective is to reduce the given graph G to a new instance G' of smaller size, such that a maximum cut in G can be recovered efficiently from any maximum cut in G'. To the best of our knowledge, preprocessing rules with theoretical guarantees have been studied so far mainly for the unit-weight max-cut. That special case of max-cut, where all edges have the same (unit) weight, is still NP-hard. The goal is thus to find a bipartition (V_1, V_2) which maximizes the size of the cut, which is the number of edges with one endpoint in V_1 and the other endpoint in V_2. To measure the effectiveness of preprocessing rules for unit-weight max-cut, one introduces an integer parameter k. This parameter measures the difference between the size of the maximum cut, and the value

$m/2-(n-1)/4$, which is the well-known lower bound on the size of the maximum cut in any m-edge n-vertex graph, due to Edwards and Erdős [61,62]. There is a set of preprocessing rules, devised by Etscheid and Mnich [66 SPP] which compresses any m-edge n-vertex graph G in linear time to a graph G' on just $\mathcal{O}(k)$ vertices, while allowing to recover the maximum cut of G. This set of rules strengthened earlier work by Crowston et al. [47 SPP], and is moreover the asymptotically best possible. To understand the practical relevance of those rules, Ferizovic et al. [68 SPP] expanded and engineered them. They demonstrated their significant impact on benchmark data sets, including synthetic instances, and data sets from the VLSI and image segmentation application domains. Their experiments revealed that current state-of-the-art solvers can be sped up by up to multiple orders of magnitude when combined with their data reduction rules. On social and biological networks in particular, the preprocessing enabled them to solve four instances that were previously unsolved in a ten-hour time limit with state-of-the-art solvers; three of these instances are now solved in less than two seconds. It is possible to expand the work on preprocessing for unit-weight max-cut to instances with all positive weights. However, designing practically-efficient preprocessing rules for the general max-cut problem, which also provides theoretical guarantees on the kernel size, remains a challenge. Recent work in this direction was done by Lange et al. [116], who designed reduction rules for general max-cut. They showed the efficacy of their rules on instances from computer vision, biomedical image analysis and statistical physics, and for those instances managed to obtain substantial size reductions.

Open Problem 7. *Is it possible to engineer efficient reduction techniques for max-cut with general edge weights?*

2.4 Treewidth and Treedepth

Many NP-hard graph problems can be efficiently solved when the input graph is a tree. A tree decomposition maps vertices of a graph to vertices in a tree, which allows techniques for trees, especially dynamic programming, to be adapted to arbitrary graphs. However, the quality of the tree decomposition impacts the efficiency of such algorithms. *Treewidth* [146] is one measure of this quality, which has been extensively studied in parameterized algorithms literature, which we now describe.

Formally, a *tree decomposition* of a graph $G=(V,E)$ is a family of subsets $\mathscr{X} \subseteq 2^V \setminus \{\varnothing\}$ of V called bags, together with a tree $T=(\mathscr{X},F)$, such that

- $V=\cup_{X\in\mathscr{X}} X$,
- for all $\{u,v\}\in E$ there exists a bag $X\in\mathscr{X}$ such that $u,v\in X$, and
- for all $v\in V$, the bags $\mathscr{X}_v=\{X\in\mathscr{X} \mid v\in X\}$ containing v induce a connected subgraph $T[\mathscr{X}_v]$ (which is necessarily a subtree of T).

The *width* of a tree decomposition of G is one less than the cardinality of its largest bag, that is, $\max_{X\in\mathscr{X}}\{|X|\}-1$. The treewidth of G, denoted $\mathrm{tw}(G)$, is the minimum width over all tree decompositions of G.

Unsurprisingly, computing $\mathrm{tw}(G)$ is NP-hard and deciding if $\mathrm{tw}(G)\le k$ for some positive integer k is NP-complete. This *treewidth* problem is a canonical problem with many theoretical and practical results in the literature. It is fixed-parameter tractable with running time $2^{\mathcal{O}(k^3)}n$ [21], implying it has a kernel exponential in k^3 [32]. The problem does not have a kernel size subexponential in k unless NP $\subseteq$ coNP/poly [22]. Hence, most work focuses on constructing tree decompositions of small width, either approximately [23], or exactly using methods such as positive-instance driven dynamic programming [156]. Both the first and second PACE Challenges had a treewidth track [55]. However, polynomial kernels exist for other parameters. Bodlaender et al. [25] give polynomial kernels of size $\mathcal{O}(\mathsf{fvs}(G)^4)$ and $\mathcal{O}(\mathsf{vc}(G)^3)$, where $\mathsf{fvs}(G)$ is the size of a minimum feedback vertex set and $\mathsf{vc}(G)$ the size of a minimum vertex cover of G, respectively. Their work is inspired by data reduction rules that are known to work well in practice (discussed below), and also includes new rules based on the notion of "clique seeing" paths. Jansen [98] improved the latter kernel to size $\mathcal{O}(\mathsf{vc}(G)^2)$ by introducing a new reduction rule to efficiently find independent sets whose elimination has a predictable effect on the treewidth. To the best of our knowledge, no experiments have been done with clique seeing paths or Jansen's reduction.

Open Problem 8. *Is the rule of Jansen [98] effective in practice?*

Much work has been done in making practical data reductions for the treewidth problem. In early work, Arnborg and Proskurowski [8] introduced reduction rules for recognizing and characterizing partial 3-trees. Bodlaender et al. [27] categorized these reductions into six types (islet, twig, series, triangle, buddy, and cube) and extended these rules, showing them to be highly effective at reducing graph size in practice [27]. Of note here are two variations of well-known reductions from other problems: simplicial vertices and twins of degree 3. They further give a reduction for *almost* simplicial vertices (vertices with all but one neighbor inducing a clique). On graphs with up to 3 032 vertices, the reductions quickly remove 77% of vertices on average, whereas the simplicial vertex reduction alone remove 51% of vertices on average. The worst performing instances had 30% of their vertices removed. Den van Eijkhof et al. [63] generalized many of these reduction rules. They not only introduce new weighted variants, but generalize most previous reductions with a "contraction" reduction rule, and further introduce a reduction for twins of higher degree.

Later, Bodlaender et al. [26] introduced the concept of a safe separator, which decomposes the graph into subgraphs that can be solved independently. It was already known that clique separators were safe [136]; however, they generalize the concept and introduce other easy-to-find separators. They further show that previous reduction rules are subsumed by their safe separator technique. In experiments, their reductions decomposed 33 out of 40 instances. When run as a preprocessing step, their technique speeds up an existing triangulation heuristic, sometimes by multiple orders of magnitude. However, it only gives modest speedups over preprocessing using existing reductions.

Open Problem 9. *How effective are existing treewidth reductions on large sparse graphs (e.g.,with millions of vertices) in practice?*

Open Problem 10. *Can heuristic methods be used to efficiently find safe separators in practice?*

A related concept exists for decompositions into rooted trees. A *treedepth decomposition* of a graph $G = (V,E)$ is a rooted forest F, together with an injective mapping

$\phi : V(G) \to V(F)$ such that, for each edge $(u,v) \in E$, one of $\phi(v)$ or $\phi(u)$ is an ancestor of the other. The treedepth of G, denoted by $\mathsf{td}(G)$, is the minimum height of any treedepth decomposition of G. The *treedepth* problem, deciding if $\mathsf{td}(G) \leq k$ for some positive integer k, is NP-complete [142].

Many similar results exist for the treewidth and treedepth problems. Reidl et al. [145] give a fixed-parameter tractable algorithm for treedepth k, with running time $2^{\mathcal{O}(k^2)}n$, implying the existence of a kernel of size exponential in k^2, and no subexponential kernel exists unless NP $\subseteq$ coNP/poly [22]. However, when parameterized on the vertex cover number $\mathsf{vc}(G)$, the problem has a kernel of size $\mathcal{O}(\mathsf{vc}(G)^3)$ [109], which is achieved through two simple reduction rules that also apply to treewidth: removing simplicial vertices and adding edges between vertices with at least k common neighbors.

However, as far as we are aware, there are significantly fewer experimental works with data reduction rules for treedepth. The 5th PACE Challenge in 2020 was dedicated to exact and heuristic solutions for treedepth. The winning solver by Trimble [161] did not employ any data reduction rules (instead using symmetry breaking together with a variety of lower bounding techniques); however, the second place solver by Korhonen [112] applies the simplicial vertex rule by Kobayashi and Tamaki [109] and a generalization of their common neighbor rule. Korhonen further introduces a new reduction rule based on the Schäffer's linear-time algorithm [149] for computing the treedepth of trees. This rule replaces a tree subgraph $G[T]$ having $|N(V \setminus T)| = 1$ with a subgraph of size $\mathsf{td}(G[T]^2)$. As far as we know there are no published results on the efficacy of these reduction rules. Of further interest is that this algorithm uses minimal separator enumeration. We conclude with the following open problems.

Open Problem 11. *How effective are the reductions of Kobayashi and Tamaki [109] and Korhonen [112] in practice?*

Open Problem 12. *Does the notion of a safe separator extend to the treedepth problem?*

2.5 Hitting Set

Given a set S along with a collection C of its subsets, the *hitting set* problem asks for a subset of S, of minimum cardinality, that has a non-empty intersection with each and every member of C. Hitting set is the dual of *set cover*, which seeks a minimum-cardinality subset of C whose union is S. If the elements of S and C are treated as red and blue vertices, respectively, of a bipartite graph, the equivalent graph theoretic problem is known as *red-blue dominating set* (RBDS).

Hitting set is NP-hard, and $W[2]$-hard when parameterized by the solution size [56]. It becomes fixed-parameter tractable when each member of C is of size bounded by a constant d. In this case the problem is often referred to as d-Hitting Set and it corresponds to RBDS restricted to (red-blue) graphs where each red vertex has at most d neighbors. The problem is also known to be fixed-parameter tractable when parameterized by $|C|$, but this particular parameter is expected to be large in practice. The most popular reductions for Hitting Set are due to Weihe [166]. They are simply based on removing any possible redundant elements from S and C. In this context, an element of S is redundant if all members of C that contain it contain another element; while a member of C is redundant if it is a superset of another member of C. The application of these two rules alone proved to be highly effective on large public transportation networks resulting in a huge reduction in size as pointed out recently by Bläsius et al. [15].

More sophisticated reduction algorithms appeared in the context of kernelization for *d-hitting set* [2, 129, 134]. The kernelization approach of Abu-Khzam [2] was adopted by Mellor et al. [125] and proved to be effective in the context of multiple drug selection for cancer therapy. Moreover, linear-time algorithms that can obtain a kernel of size $\mathcal{O}(k^d)$ were presented by van Bevern [162] and Fafianie and Kratsch [67]. Practical implementations of these algorithms have been addressed recently by van Bevern and Smirnov [163] where they were shown to be more efficient than the reduction procedure of Weihe [166] for small d (up to 5), but can result in more effective data reduction when combined with the reduction rules of Weihe [166].

2.6 Steiner Trees

Given an undirected graph with non-negative edge weights as well as a subset of the vertices (terminals), the *Steiner tree* problem is to find the lightest tree spanning the terminals. There has been a wide range of implementations tackling the Steiner tree problem. Data reductions have long been used for the problem, see, e.g., Polzin [141] or Daneshmand [52]. Daneshmand [52] in particular has shown already in 2004 that many Steiner tree problem instances can be solved by reduction- and heuristic-based approaches.

Recently there have been two implementation challenges, the 11th DIMACS Implementation Challenge in 2014 and the 3rd PACE Challenge [28] in 2018. Here, we focus on the most successful implementations of the 3rd PACE Challenge and the approaches that have been published afterwards. The PACE Challenge had three tracks overall – two exact tracks with one focusing on algorithms for problems with few terminals and one focusing on problems with low treewidth, as well as one heuristic track.

The implementation of Iwata and Shigemura [96] won the track with problems that have few terminals. Their algorithm is based on the dynamic programming formulation of Erickson-Monma-Veinott [65] which has a theoretical running time of $\mathcal{O}(3^t n + 2^t(n \log n + m))$ with t being the number of terminals. Iwata and Shigemura use a novel separator-based pruning technique to speed up their implementation (while keeping the worst-case bound of Erickson-Monma-Veinott). This technique allows them to prune a large number of entries in the dynamic programming table.

The track with problems that have low treewidth was won by SCIP-Jack [143, 144] due to Koch and Rehfeldt. This approach is based on the branch-and-cut principle and was already very successful during the 11th DIMACS Implementation Challenge. For the PACE Challenge, the authors use data reductions that typically reduce the number of edges in the problems by more than 90%. Many instances can already be completely solved by presolving. Moreover, on the remaining instances that can not be presolved, the authors use heuristics to find strong upper and lower bounds quickly. The authors find that in more than 90% of cases that the heuristic already finds the optimum solution on the instances that have not been presolved. Lastly, the branch-and-cut procedure is used to compute lower bounds and prove optimality. Later, the approach was improved [152] to run in distributed memory and thus, by using up to 43 000 cores, managed to solve additional previously unsolved instances or improved on the previously best known solution.

Open Problem 13. *Are there new reductions that have not yet been tried in practice that could help to solve more instances to optimality in practice?*

Open Problem 14. *Can existing reductions for the standard Steiner tree problem be transferred to the more general multi-level Steiner tree problem?*

2.7 Minimum Fill-In

The *minimum fill-in* problem is a critical problem that accelerates Gaussian elimination when solving sparse linear systems [147]. Given a matrix A representing the sparse linear system $Ax = b$, the goal is to find a permutation matrix P that minimizes the number of non-zeros introduced when factorizing $A = PAP^T$. Equivalently, treating A as the adjacency matrix of a graph $G = (V, E)$, we wish to minimize the number of edges introduced in an *elimination ordering*, defined as follows. An *elimination step* removes a vertex $v \in V$ and its incident edges, and adds edges between non-adjacent vertices in $N_G(v)$, producing an elimination graph G_v. An *elimination ordering* of G is a permutation $v_1 v_2 .. v_n$ of all the vertices in G, and the *fill-in* of the ordering is the number of edges introduced by eliminating vertices $v_1, v_2, \ldots, v_n$ in this order. The minimum fill-in is the smallest fill-in given by any elimination ordering. We are often interested in not just computing the minimum fill-in, but an elimination ordering that has minimum fill-in.

Not only is the minimum fill-in NP-hard to compute [171], no polynomial time approximation scheme exists for the problem unless P = NP [39]. However, the problem is fixed-parameter tractable [103], when the input parameter k is the minimum fill-in. The fastest known fixed-parameter algorithm for the problem is due to Fomin and Villanger [71], with running time $2^{\mathcal{O}(\sqrt{k}\log k)} + \mathcal{O}(k^2nm)$, where the additive $\mathcal{O}(k^2nm)$ term is the time to compute a kernel of $\mathcal{O}(k^3)$ vertices [102]. Note that this algorithm is subexponential in the minimum fill-in k and, moreover, is nearly optimal: Cao and Sandeep [39] showed that no algorithm with running time $2^{\mathcal{O}(k^{1/2-\delta})} \cdot n^{\mathcal{O}(1)}$ exists for any positive constant δ, assuming the exponential time hypothesis holds. The smallest known kernel for the problem is due to Natanzon et al. [132] has $2k^2 + 4k$ vertices. The reductions all have the same flavor and are derived for the equivalent problem of *chordal completion*: finding the minimum number of edges to add to the graph so that it is chordal. Kernelization is done by partitioning the vertices into two sets A and B where B induces a chordal graph and A contains vertices from every chordless cycle in G. The set A is formed by repeatedly finding chordless cycles in $G[B]$ via the MCS algorithm [157, 158] and moving a subset of their vertices to A until $G[B]$ is chordal. Then *essential edges* are added to the chordless cycles induced by A, which is the kernel.

In practice, the minimum fill-in problem is extremely hard to solve exactly. Indeed, in the 2nd PACE Challenge in 2017, the winning solver for the minimum fill-in problem only solved 54 out of 100 instances [55], when each instance is given a 30-min time limit. The top three submissions all used kernelization [102] together with dynamic programming over potential maximal cliques [29,156]. The first place submission by Kobayashi and Tamaki used generalized variants of the data reduction rules of Bodlaender et al. [24], and the third place submission performed preprocessing adapted from the safe separator technique for treewidth [26] in addition to kernelization [102].

However, heuristics, including nested dissection [78] and minimum-degree ordering [159], work quite well in practice for real-world (typically sparse) graphs. Early researchers noted that indistinguishable vertices may be eliminated together, and therefore may be collapsed into a representative vertex while ordering [9,57]. This reduction speeds up the minimum degree algorithm by more than a factor two in experiments [79]. Ost et al. [137 SPP] recently introduced new data reduction rules based on twins, simplicial vertices, and path compression, and experiments show that they are highly effective in practice when applied before running nested dissection. For road networks, when used as a preprocessing step with other inexact reductions, their techniques give speedups of between 1.79 and 6.37 over nested dissection while simultaneously reducing the fill-in. On social networks, their reductions yield speedups of between 1.72 and 3.92 on 19 out of 21 social networks tested, and the fill-in was reduced on all but one instance.

Open Problem 15. *How effective are the reductions by Ost et al. [137 SPP] when combined with other reductions [132]?*

Open Problem 16. *Is branch-and-reduce feasible for the minimum fill-in problem?*

2.8 Vertex Coloring

Given an unweighted, undirected simple graph $G = (V, E)$, the *q-coloring* problem asks if there exists an assignment of at most q colors to all vertices in V such that no two adjacent vertices have the same color (i.e., a *proper coloring*). The problem of finding the minimum number $\chi(G)$ of colors for which a proper coloring of G exists is known as the *chromatic number* problem.

These problems have received considerable attention by the parameterized algorithms community; however, somewhat surprisingly, there is a wide divide between theory and practice. In theory, a kernel parameterized on only the number of colors is unlikely: since graph coloring is NP-hard for $q = 3$ colors [74], this would give a constant-sized kernel, implying P = NP. Therefore, research has focused on other parameters.

When considering the treewidth $\mathsf{tw}(G)$ of the graph G, if G is given together with a tree decomposition of width $k \geq \mathsf{tw}(G)$, dynamic programming over the tree decomposition gives an algorithm solving q-coloring in time $q^k k^{\mathcal{O}(1)} n$ [49, Theorem 7.9]. Assuming the Strong Exponential Time Hypothesis (SETH) no algorithm of running time $\mathcal{O}(q-\varepsilon)^{\mathsf{tw}(G)}$ exists [122] for any $\varepsilon > 0$. Using the same technique, the chromatic number can be computed in time $k^{\mathcal{O}(k)} n$ [49, Theorem 7.10]. Since these algorithms are fixed-parameter algorithms, the result due to Cai et al. [32] implies kernels of size $q^k k^{\mathcal{O}(1)}$ and $k^{\mathcal{O}(k)}$ exist for q-coloring and chromatic number, respectively. Treewidth is often small for sparse graphs in practice; however, as far as we know, these techniques have not been tried in practice, leading to the following open problem.

Open Problem 17. *How effective is dynamic programming over a tree decomposition for q-coloring (or chromatic number) on sparse graphs in practice?*

Another parameter of interest is size of a minimum vertex cover. Recently, Jansen and Pieterse [99] gave a kernel parameterized on the number $q \geq 3$ of colors and the size k of a minimum vertex cover, having size $\mathcal{O}(k^{q-1} \log k)$ bits, which is optimal up to a factor of $k^{\mathcal{O}(1)}$ [97]. Their result also applies for a tighter parameter, when k is the size of the twin cover. Their technique uses constraint satisfaction with low-degree polynomials. However, in practice, sparse graphs often have a minimum vertex cover size that is linear in the number of vertices. Thus, to be useful in practice, the actual kernel would need to have significantly smaller size. However, to date no one has tested their method in practice, leading to our next open problem for q-coloring.

Open Problem 18. *How effective are the reductions of Jansen and Pieterse [99] in practice?*

The data reductions that have been implemented in practice are simple and without theoretical guarantees on the size of the reduced graph; however, they are also very effective on large sparse graphs. In particular, in experiments for a branch-and-cut algorithm, Mendéz-Díaz and Zabala [126] first preprocess the input graph by computing a large maximal clique K of k vertices, which is a lower bound on the chromatic number. They then iteratively remove each vertex v of degree at most $k-1$ (resulting in a k-core), which is possible since $\chi(G) = \chi(G - \{v\})$. They further give a rule to remove certain vertices with non-neighbors in K. In experiments on 63 graphs of up to 5 231 vertices from the second DIMACS Implementation Challenge[1], their data reductions reduced all graphs between 1–93%, working best on sparse instances. 36 of the 63 instances were reduced by at least 25%, and 21 instances were reduced by at least 50%. The largest percentage reduction was 93% for the `homer` instance, reducing from 561 to 38 vertices.

Verma et al. [164] extend this technique. They first compute lower and upper bounds for the chromatic number, and then iteratively apply the k-core reduction to heuristically color graphs for decreasing values of k. Their key contribution is beginning with an exact coloring of the k-core, which gives a better bound than an initial clique. With this technique they are able to exactly find the chromatic number for very large sparse graphs with up to millions of vertices, with running time varying from seconds to hours. In total they solve 33 of 53 instances from SNAP[2] and the tenth DIMACS Implementation Challenge[3]. Lin et al. [121] extended the low degree reduction to remove entire independent sets of vertices with low degree, which in some cases is orders of magnitude faster than the algorithm of Verma et al. [164]. However, they are not able to solve any additional instances.

We finally note that a crown reduction exists for the *dual coloring* problem, which asks if the graph has an $(n-k)$-coloring [70]. Crown reductions are particularly effective in practice for other problems, specifically the minimum vertex cover problem. In theory, for dual coloring, the crown reduction produces a kernel of size at

[1] http://archive.dimacs.rutgers.edu/pub/challenge/graph/benchmarks/color/.
[2] http://snap.stanford.edu/data.
[3] http://www.cc.gatech.edu/dimacs10/.

most $3k-3$ [70, Theorem 4.9]. As far as we are aware, no one has performed experiments with this reduction, leading to our final open problem for graph coloring.

Open Problem 19. *How effective is the crown reduction [70, Theorem 4.9] for graph coloring in practice?*

2.9 Cluster Editing

The *cluster editing* problem is as follows: given a graph $G=(V,E)$, transform it into a vertex-disjoint union of cliques by inserting and deleting a minimum number of edges,i.e., by making a minimum number of editions in the graph. The problem is also known as correlation clustering and has many applications, especially in computational biology [17]. The parameterized complexity of the cluster editing problem using the number of edits k as a parameter is well-studied. The currently best known algorithm in theory is due to Böcker [16] and has running time $\mathcal{O}(1.62^k+n+m)$, where m is the number of edges.

There has been a wide range of methods applying fixed-parameter techniques in the area. Dehne et al. [53] presented the first practical implementation of a fixed-parameter based method for cluster editing. Their algorithm is exact and implements the kernelization routines of [82] and adds ideas to bound the search space for the parameter k via linear programming. Gramm et al. contributed three reduction rules. For example, if two vertices u and v have more than k common neighbors then the edge $\{u,v\}$ has to be in the solution and is added if it is not present. Moreover, if u and v have more than k non-common neighbors, i.e., vertices that are either neighbors of u but not v or vice versa, then the edge $\{u,v\}$ does not belong to the solution. Lastly, if u and v have more than k common and more than k non-common neighbors, then the given instance has no solution. Overall, their method performs best using a refined branching method with re-kernelization. Interestingly, the experimental analysis of their algorithm shows that binary search may not be the best way to implement a fixed-parameter based approach for cluster editing.

Guo [83] later gave parameter-independent data reductions based on critical cliques, obtaining a linear kernel of $4k$ vertices, which was improved by Chen and Meng [45] to $2k$. Böcker et al. [20] introduced additional parameter-independent data reductions and find that preprocessing is possible if the number of edge modifications is significantly smaller than the number of vertices in the graph. In addition to the parameter-independent rules they combine their technique with the parameter-dependent reductions from above with lower and upper bounds. Böcker et al. find that they can effectively reduce graphs that satisfy $k \leq 25|V|$, whereas the reductions due to Guo [83] are only effective for $k \leq |V|/2$. Their experiments show that computing exact solutions for cluster editing is no longer limited to small or almost transitive graphs. Afterwards, Böcker et al. [18, 19] extended their results to the weighted version of the problem in which the weight of an edge yields the cost of deleting or inserting it, and the goal is to apply a set of edge modifications with minimum total weight. To this end, they include non-trivial extensions of the data reduction rules of the unweighted case. Additionally, they present a technique to merge vertices which drastically improves the running time of their algorithm. Recently, Bastos et al. [135] combine exact methods with local search heuristics. More precisely, the authors propose a GRASP and an ILS metaheuristic with different neighborhoods as well as a new reduction rule for the problem. They show that the used data reduction rules can speed up linear programming for some instances up to 95% decreased runtime after using reduction rules and 41% decreased runtime on average on the instances that the solver could solve to optimality.

Open Problem 20. *Is it possible to compute small kernels in practice if the parameter k is larger than $25|V|$? Are there any specific data reduction rules for that case? If an instance in practice does not reduce well, does that help to obtain bounds on the parameter k?*

Since the parameter k is often large compared to the number of vertices, fixed-parameter algorithms may not always be practical. There has been several attempts to use other parameters such as the number of missing edges per cluster as well as the number of edges between clusters [85], the total number of edge modifications per vertex [3, 110]. Abu-Khzam [3], using local parameters that bound the amount of (either or both) edge addition and deletion per vertex resulted in a number of reduction rules, showed how to solve much larger problem instances and apply the problem effectively in data analysis [11, 12].

2.10 Multiterminal Cut

The *multiterminal cut* problem with k terminals is defined as follows: Its input is an undirected edge-weighted graph $G = (V, E, w)$ with edge weights $w : E \mapsto \mathbb{N}_{>0}$ and its goal is to divide its set of vertices into b blocks such that each block contains exactly one terminal and the weight sum of the edges running between the blocks is minimized. It is a fundamental combinatorial optimization problem that was first formulated by Dahlhaus et al. [50] and Cunningham [48]. It is NP-hard for all $b \geq 3$ [50], even on planar graphs, and reduces to the minimum s-t-cut problem, which is in P, for $b = 2$. The minimum s-t-cut problem aims to find the minimum cut in which the vertices s and t are in different blocks. Most algorithms for the multiterminal cut problem use minimum s-t-cuts as a subroutine. Dahlhaus et al. [50] give a $2(1 - 1/b)$ approximation algorithm with polynomial running time. Their approximation algorithm uses the notion of *isolating cuts*, i.e., a minimum cut separating a terminal from all other terminals. They prove that the union of the $b - 1$ smallest isolating cuts yields a valid multiterminal cut with the desired approximation ratio. The currently best known approximation algorithm by Buchbinder et al. [31] uses linear program relaxation to achieve an approximation ratio of 1.323.

Marx [123] proves that the multiterminal cut problem is fixed-parameter tractable when parameterized by multiterminal cut weight $\mathscr{W}(G)$. Chen et al. [44] give the first fixed-parameter tractable algorithm with running time of $4^{\mathscr{W}(G)} \cdot n^{\mathscr{O}(1)}$, later improved by Xiao [167] to $2^{\mathscr{W}(G)} \cdot n^{\mathscr{O}(1)}$ and by Cao et al. [38] to $1.84^{\mathscr{W}(G)} \cdot n^{\mathscr{O}(1)}$.

Recently, Henzinger et al. [88] engineer an algorithm that combines the branch-and-bound formulation of Xiao [167] with existing and new data reduction rules for the problem and present a shared-memory parallel branch-and-reduce algorithm for the multiterminal cut problem. Experiments indicate that this is orders of magnitude faster than previous ILP formulations for the problem that have been employed by practitioners. Later, reduction rules were combined with local search algorithms for the problem [87 SPP]. The algorithm uses a wide variety of reduction rules with varying computational complexity; using vertex neighborhoods, edge connectivities, articulation points, maximum flows and more criteria to reduce the problem size; Henzinger et al. [87 SPP] report size reductions of up to multiple orders of magnitude in some instances, which make large instances solvable in practice. Additionally, they give an inexact algorithm that aggressively prunes subproblems which likely do not yield an improved solution.

Open Problem 21. *Is there an efficient way to find semi-isolated small clusters that can be contracted (either exact or inexact contraction)?*

Open Problem 22. *The algorithm by Henzinger et al. [88] uses only reductions that guarantee that the optimal solution remains in the graph. Are there reductions that do not guarantee optimality but give good performance in practice?*

3 Recent Advances for Problems in P

3.1 Minimum Cut

Given an undirected graph with non-negative edge weights, the *minimum cut* problem is to partition the vertices into two sets so that the sum of edge weights between the two sets is minimized. The size of a minimum cut is often also referred to as the *edge connectivity* of a graph [91, 130]. Gomory and Hu [81] observed that a (global) minimum cut can be computed with $n-1$ minimum s-t-cut computations. For the following decades, this result by Gomory and Hu was used to find better algorithms for global minimum cut using improved maximum flow algorithms [105]. Hao and Orlin [84] adapt the push-relabel algorithm to pass information to future flow computations. When a push-relabel iteration is finished, they implicitly merge the source and sink to form a new sink and find a new source. Vertex heights are maintained over multiple iterations of push-relabel. With these techniques, they achieve a total running time of $\mathcal{O}(mn\log\frac{n^2}{m})$ for a graph with n vertices and m edges, which is asymptotically equal to a single run of the push-relabel algorithm.

However, for minimum cut algorithms to be viable for applications they must be fast on small data sets and scale to large data sets. Thus, an algorithm should have either linear or near-linear running time, or have an efficient parallelization. *All* existing exact algorithms have non-linear running time [84, 91, 105], the fastest of which is the deterministic algorithm of Henzinger et al. [91] with running time $\mathcal{O}(m\log^2 n\log\log^2 n)$. Although this is arguably near-linear theoretical running time, it is not known how the algorithm performs in practice. Even the randomized algorithm of Karger and Stein [105], which finds a minimum cut only with high probability, has $\mathcal{O}(n^2\log^3 n)$ running time, although this was later improved by Karger [104] to $\mathcal{O}(m\log^3 n)$ and recently improved further by Gawrychowski et al. [76] to $\mathcal{O}(m\log^2 n)$. The algorithm of Karger and Stein can be seen as probabilistic data reduction algorithms as they contract random edges to reduce the problem size, and give the correct answer with a certain probability.

Padberg and Rinaldi [138] give a set of heuristics for edge contraction. Chekuri et al. [43] give an implementation of these heuristics that can be performed in time linear in the graph size. Using these heuristics it is possible to sparsify a graph while preserving at least one minimum cut in the graph. If their algorithm does not find an edge to contract, it performs a maximum flow computation, giving the algorithm worst case running time $\mathcal{O}(n^4)$. However, the heuristics can also be used to improve the expected running time of other algorithms by applying them on interim graphs [43].

Open Problem 23. *Some reductions of Padberg and Rinaldi [138] potentially check each triangle in a graph. Can pruning be used to efficiently identify which subset needs to be checked?*

Nagamochi et al. [130, 131] give a minimum cut algorithm that does not use any flow computations. Instead, their algorithm uses maximum spanning forests to find a non-empty set of contractible edges. The intuition behind the algorithm is as follows: suppose you have an unweighted graph with minimum cut value exactly one. Then any spanning tree must contain at least one edge of each of the minimum cuts. Hence, after computing a spanning tree, every remaining edge can be contracted without losing the minimum cut. Nagamochi, Ono and Ibaraki extend this idea to the case where the graph can have edges with positive weight as well as the case in which the minimum cut is bounded by $\hat{\lambda}$ and show how edges are identified using one modified breadth first search. This contraction algorithm is run until the graph is contracted into a single vertex. The algorithm has a running time of $\mathcal{O}(mn + n^2 \log n)$. Stoer and Wagner [154] give a simpler variant of the algorithm of Nagamochi, Ono and Ibaraki [131], which has a the same asymptotic time complexity. The performance of this algorithm on real-world instances, however, is significantly worse than the performance of the algorithms of Nagamochi, Ono and Ibaraki or Hao and Orlin, as shown in experiments conducted by Jünger et al. [101]. Both the algorithms of Hao and Orlin, and Nagamochi, Ono and Ibaraki achieve close to linear running time on most benchmark instances [43, 101].

Based on the algorithm of Nagamochi, Ono and Ibaraki, Matula [124] gives a $(2+\varepsilon)$-approximation algorithm for the minimum cut problem. The algorithm contracts more edges than the algorithm of Nagamochi, Ono and Ibaraki to guarantee a linear time complexity while still guaranteeing a $(2+\varepsilon)$-approximation factor. Inspired by random contractions, Henzinger et al. [89, 150 SPP] first gave an shared-memory parallel algorithm without guarantees on the cut size. The algorithm is randomized, and has running time $\mathcal{O}(n+m)$ when run sequentially. It repeatedly reduces of the input graph size with both heuristic and exact techniques, and then solve the smallest remaining problem with exact methods. The core idea of the inexact algorithm is that edges in densely connected regions (i.e., inside a cluster of a clustering) are unlikely to be in a minimum cut. The algorithm further uses exact reduction rules from Padberg and Rinaldi [138]. For example, given a bound $\hat{\lambda}$ on the minimum cut, one can obviously contract each edge having weight larger than $\hat{\lambda}$, without losing optimality. Experimental results indicate that the algorithm finds optimal cuts on almost all instances. At the same time, even when run sequentially, the algorithm is significantly faster (up to a factor of 4.85) than other state-of-the-art algorithms.

Later, Henzinger et al. [86, 150 SPP] engineered the fastest known *exact* minimum cut algorithm for the problem. To do so, the authors incorporate the proposed inexact method, use better-suited data structures and other optimizations as well as parallelization of exact methods. More precisely, the exact algorithm uses the *inexact* minimum cut algorithm from above [89, 150 SPP] to obtain a better approximate bound $\hat{\lambda}$ for the problem (recall that the algorithm almost always gave the correct result). As known reduction techniques depend on this bound, the better bound enables us to apply more reductions and to reduce the size of the graph much faster. For example, edges whose incident vertices have a connectivity of at least $\hat{\lambda}$, can be contracted without the contraction affecting the minimum cut. The new exact algorithm outperforms the state-of-the-art by a factor of up to 2.5 already sequentially, and when run in parallel by a factor of up to 12.9. Similar reduction rules were later used by Henzinger et al. [90 SPP, 150 SPP] to find all minimum cuts in graphs.

3.2 Matching

A matching M of a graph $G = (V,E)$ is a subset of edges such that no two elements of M have a common endpoint. Many applications require the computation of matchings with certain properties, like being maximal (no edge can be added to M without violating the matching property), having maximum cardinality, or having maximum total weight $\sum_{e \in M} w(e)$, where w is a positive weight function that assigns weights to edges. Although these problems can be solved optimally in polynomial time, optimal algorithms are not fast enough for many applications involving large graphs where we need near linear time algorithms. For example, the most efficient algorithms for graph partitioning rely on repeatedly contracting maximal matchings, often trying to maximize some edge rating function w. We refer to Holtgrewe et al. [94] for details and examples. For the *maximum cardinality matching* problem, already in the 1980s data reduction rules were proposed by Karp and Sipser [107]. The rules are able to deal with vertices that have degree smaller than two. For example, it is quite easy to see that a vertex having degree zero can be removed from the graph, or if a vertex has degree one, then there is always a maximum matching that has this edge matched.

Möhring and Müller-Hannemann [128] were among the first to use the rules to speed up heuristic algorithms for the general maximum cardinality problem. As exact algorithms for the matching problems typically search for augmenting paths, they can be sped up by using a good initial matching. Hence, later Langguth et al. [117] analyzed the effects of various initializations on the total running time of several exact algorithms for the bipartite maximum cardinality problem and are able to achieve significant speed-ups.

Korenwein et al. [111] implement (near-)linear time data reduction rules for the unweighted case as well as the positive-integer-weight case. Applied reductions include Karp-Sipser rules, as well as rules due to Mertzios et al. [127] who have also shown that the maximum cardinality matching problem admits a kernel with at most $12k$ vertices and $13k$ edges where k is the feedback edge number. Moreover, Koana et al. [111] transfer results from vertex cover to the matching problem, e.g.,crown and LP-based data reductions. Experiments indicate that using data reduction rules can speed up state-of-the-art solvers by a factor of 4.7 for the unweighted case and 12.72 on average in the weighted case.

Open Problem 24. *Can the reduction rules due to Koana et al. [111] be exhaustively applied in linear time? Are there more rules that can be transferred from vertex cover to the matching problem that can be applied in near-linear time?*

Kaya et al. [108] also use Karp-Sipser-based kernels for bipartite graph matching. In particular, the authors describe an efficient implementation as well as modifications to reduce time complexity on worst-case instances. Their implementation is about a factor 2 faster then the general purpose implementation of Koana et al. [111]. Recently, Panagiotas and Uçar [139] engineer fast almost optimal algorithms for bipartite graph matching. To this end, the authors investigate two randomized algorithms by Karp et al. [106] and Goel et al. [80] and convert them to efficient heuristics for bipartite graphs. In particular, the algorithm by Karp [106] incorporates Karp-Sipser rules. Both of their heuristics run in near linear time and obtain matchings whose cardinality is more than 99% of the maximum.

Open Problem 25. *Is it possible to implement the degree-2 vertex Karp-Sipser rule in linear time?*

4 Engineering Techniques

Engineering techniques are necessary to make data reduction algorithms scale in practice. We give a short overview of techniques that are currently used in practice. The techniques we reference here include dependency checking, reduction tracking, plateau/increasing data transformations, limiting to simple and fast reductions, reduce and peeling, limited reductions, on-the-fly reductions and lastly parallelization.

Dependency checking allows pruning of reductions when they will provably not succeed, therefore significantly reducing the number of failed reductions. To compute a kernel, algorithms typically apply their reductions $r_1, \ldots, r_j$ by iterating over all reductions and trying to apply the current reduction r_i to all vertices. If r_i reduces at least one vertex, they restart with reduction r_1. When reduction r_j is executed, but does not reduce any vertex, all reductions have been applied exhaustively, and a kernel is found. Trying to apply every reduction to all vertices can be expensive in later stages of the algorithm where few reductions succeed. The algorithm may repeatedly attempt to apply the same reduction to a vertex even though the graph has not changed sufficiently to allow the reduction to succeed. Checking dependencies between reductions [93], allows to avoid applying certain local reductions when they will provably not succeed, e.g.,if their relevant neighborhood did not change since the reduction was last checked. Therefore dependency checking keeps a set D of *viable* candidate vertices: vertices whose relevant neighborhood has changed and vertices that have never been considered for reductions. Then reductions are only applied to candidates that are in the set D. This avoids a lot of work and can speed up data reduction significantly.

Reduction Tracking. The algorithm by Hespe et al. [93] stops local reductions when they are not effectively reducing the global graph sizes. It is not *always* ideal to apply reductions exhaustively—for example, if only few reductions will succeed and they are costly. During later stages of a data reduction algorithm, local reductions may lead to very few graph changes. Therefore, it may be better to stop local reductions early instead of performing them exhaustively and switch to global, more expensive reductions that may change the graph more significantly. Although the resulting graph is kernel-like, it may be possible to reduce it further. Such a graph is called a *quasi kernel*. Note, however, that this is a trade-off between size of the reduced graph and data reduction speed.

Plateau/Increasing Transformations. The general scheme in data reduction is to apply reductions exhaustively until non of the available reductions can be applied anymore. Gellner et al. [77] engineer new generalized data reduction and transformation rules for the weighted independent set problem. A key feature of this work are some transformation rules that can *increase* the size of the input. Surprisingly, these so-called *increasing transformations* can simplify the problem and also open up the reduction space to yield even smaller irreducible graphs later throughout the algorithm. Overall, for the weighted independent set problem, this yields significant speed ups and enables the authors to solve more instances to optimality than previously possible.

Simple Reductions. Often the smallest kernels (or seemingly equivalently, the most varied reductions) give the best chance at finding solutions. For instance, the reductions used by Akiba and Iwata [5] for the maximum independent set problem are the *only* ones known to compute an exact solution on certain large-scale graphs, and these reductions are further successful in computing exact solutions in an evolutionary approach [114]. However it is not always beneficial to compute the smallest kernel possible. Fast and simple reductions can compute kernels that are "small enough" for local search to quickly find high-quality, and even exact, solutions much faster than the reductions used to find the smallest kernels [42,51]. Fast and simple reductions can even be used to solve many large-scale instances exactly [155] just as quickly as the algorithm by Akiba and Iwata [5].

Reduce and Peel. Lamm et al. [114] showed that including reductions in a branch-and-reduce inspired evolutionary algorithm for the independent set problem enables finding exact solutions much faster than provably exact algorithms. To this end, reductions are applied exhaustively. Once a reduced graph is computed, vertices that are unlikely to be in the solution, e.g.,vertices having a very large degree, are removed from the graph and hence excluded from the solution. The algorithm then proceeds recursively. Chang et al. [42] improved on this result by implementing reduction rules to reduce the lead time for kernelization for local search. They introduce "reducing–peeling" to find a large initial solution for local search. This technique can be viewed as computing one path through the search space of a branch-and-reduce algorithm: they repeatedly exclude high-degree vertices and reduce the graph until it is empty, then they take the solution found as an initial solution for local search.

Limited Reductions. Sometimes reductions can be very expensive, for example if their running time depends on the number of edges in the neighborhood of a certain vertex. However, as mentioned above it is often not necessary to compute the smallest possible kernel in practice. Hence, a common technique in practice is to exclude such reductions, for example, if the degree of a vertex is too large. An application of this technique is due to Ost et al. [137 SPP] for the vertex ordering problem, where the simplicial vertex reduction rule is limited to vertices of degree at most 18.

On-the-Fly Reductions. Data reduction can be used as a preprocessing step to exact algorithms. However, reductions are also used to reduce the size of the search space of local search algorithms without losing solution quality. Dahlum et al. [51] apply a set of simple reductions on the fly for the independent set problem. For this algorithm, they use simple reductions that do not require changing the neighborhoods of vertices. Instead, vertices are marked as removed, e.g.,simplicial vertices. This speeds up local search significantly.

Parallelization. A general technique to speed up algorithms is parallelization. Also in data reduction parallelization is used to speed up preprocessing times. For example, "local" reduction rules have been parallelized by using graph partitioning techniques, i.e., each process works on a subgraph and applied reductions only in his subgraph [93]. At the same time, there are also attempts [93] to parallelize more expensive "global" reductions, e.g., reductions that need to access the whole input instance.

Targeted Branching. Branch-and-reduce algorithms often make use of vertex selection strategies that are carried over from existing branch-and-bound approaches. However, these selection strategies often do not take into account that removing certain vertices from the graph might result in an increase of the reduction space, which in turn might lead to smaller search trees. Gao et al. [72] thus present a dynamic vertex selection strategy that also takes into account one of their reduction rules and uses a degree-based selection as a fallback. Their experiments indicate that this strategy is able to provide better results when compared to a purely degree-based selection rule.

Data-Driven Reductions. Eblen et al. [60] show the benefits of using application-specific reduction rules that exploit prior knowledge of the input space. In particular, they use a reduction rule that is based on the empirical evaluation of large transcriptomic graphs and is able to drastically reduce the running time of their algorithm on similar instances. However, this comes at the drawback of a decrease in performance for random graphs.

5 Open Problems and Future Work

We already discussed problem-specific open problems throughout this article. Here, we list some general open questions that apply to a range of problems touched in this survey. For example, in a branch-and-reduce algorithm can we branch to specifically get graphs that reduce better using the available portfolio of reductions? As a concrete example, as stated above, it may be helpful to end up with a lot of independent connected components and to achieve this one may be able to branch on a small vertex separator first. For most problems, what makes an instance hard to reduce is currently unknown, e.g., when does which data reduction rule work well in practice and why?

From the theory perspective of a practitioner, it would be better to have an analysis of the expected kernel size, rather than the worst case so as to get more realistic results in practice. One does not always need a single optimal solution, but a diverse set of high-quality solutions. Theoretical approaches for this have been proposed [13], however, they remain untested in practice. Probabilistic reductions have not yet been tried in practice. On the other hand, most of the dynamic techniques that maintain a problem kernel have also not yet been implemented. A problem that needs careful investigation is the order in which reduction rules are applied, e.g., when is it good to apply which reduction rule first? Lastly, consider an instance for a problem on which you already applied all data reduction rules at hand exhaustively. Moreover, assume that you already have an optimal solution on the reduced instance. Is it possible to discover new rules by applying machine learning techniques on such instances?

Acknowledgement. Partially supported by DFG grants MN 59/1-1, SCHU 2567/1-2 and SCHU 2567/3-1.

References

1. Abello, J., Pardalos, P.M., Resende, M.: On maximum clique problems in very large graphs. In: External Memory Algorithms, pp. 119–130 (1999). https://doi.org/10.1090/dimacs/050/06
2. Abu-Khzam, F.N.: A kernelization algorithm for d-hitting set. J. Comput. Syst. Sci. **76**(7), 524–531 (2010). https://doi.org/10.1016/j.jcss.2009.09.002
3. Abu-Khzam, F.N.: On the complexity of multi-parameterized cluster editing. J. Discrete Algor. **45**, 26–34 (2017). https://doi.org/10.1016/j.jda.2017.07.003
4. Abu-Khzam, F.N., Collins, R.L., Fellows, M.R., Langston, M.A., Suters, W.H., Symons, C.T.: Kernelization algorithms for the vertex cover problem: theory and experiments. In: Proceedings of ALENEX/ANALCO, pp. 62–69 (2004)
5. Akiba, T., Iwata, Y.: Branch-and-reduce exponential/FPT algorithms in practice: a case study of vertex cover. Theore. Comput. Sci. 609, Part **1**, 211–225 (2016). https://doi.org/10.1016/j.tcs.2015.09.023
6. Alsahafy, M., Chang, L.: Computing maximum independent sets over large sparse graphs. In: Cheng, R., Mamoulis, N., Sun, Y., Huang, X. (eds.) WISE 2020. LNCS, vol. 11881, pp. 711–727. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34223-4_45
7. Andrade, D.V., Resende, M.G., Werneck, R.F.: Fast local search for the maximum independent set problem. J. Heuristics **18**(4), 525–547 (2012). https://doi.org/10.1007/s10732-012-9196-4
8. Arnborg, S., Proskurowski, A.: Characterization and recognition of partial 3-trees. SIAM J. Algeb. Discrete Methods **7**(2), 305–314 (1986). https://doi.org/10.1137/0607033
9. Ashcraft, C.: Compressed graphs and the minimum degree algorithm. SIAM J. Scient. Comput. **16**(6), 1404–1411 (1995). https://doi.org/10.1137/0916081
10. Barahona, F., Grötschel, M., Jünger, M., Reinelt, G.: An application of combinatorial optimization to statistical physics and circuit layout design. Oper. Res. **36**(3), 493–513 (1988). https://doi.org/10.1287/opre.36.3.493
11. Barr, J.R., Shaw, P., Abu-Khzam, F.N., Yu, S., Yin, H., Thatcher, T.: Combinatorial code classification vulnerability rating. In: 2020 Second TransAI, pp. 80–83 (2020). https://doi.org/10.1109/TransAI49837.2020.00017

12. Barr, J.R., Shaw, P., Abu-Khzam, F.N., Chen, J.: Combinatorial text classification: the effect of multi-parameterized correlation clustering. In: Proceedings of GC 2019, pp. 29–36 (2019). https://doi.org/10.1109/GC46384.2019.00013
13. Baste, J., et al.: Diversity of solutions: an exploration through the lens of fixed-parameter tractability theory. In: Proceedings of IJCAI 2020, pp. 1119–1125 (2020). https://doi.org/10.24963/ijcai.2020/156
14. Berlowitz, D., Cohen, S., Kimelfeld, B.: Efficient enumeration of maximal k-plexes. In: Proceedings of SIGMOD 2015, pp. 431–444 (2015). https://doi.org/10.1145/2723372.2746478
15. Bläsius, T., Fischbeck, P., Friedrich, T., Schirneck, M.: Understanding the effectiveness of data reduction in public transportation networks. In: Proceedings of WAW 2019, pp. 87–101 (2019). https://doi.org/10.1007/978-3-030-25070-6_7
16. Böcker, S.: A golden ratio parameterized algorithm for cluster editing. J. Discrete Algor. **16**, 79–89 (2012). https://doi.org/10.1016/j.jda.2012.04.005
17. Böcker, S., Baumbach, J.: Cluster editing. In: Bonizzoni, P., Brattka, V., Löwe, B. (eds.) CiE 2013. LNCS, vol. 7921, pp. 33–44. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39053-1_5
18. Böcker, S., Briesemeister, S., Bui, Q.B.A., Truss, A.: Going weighted: parameterized algorithms for cluster editing. Theor. Comput. Sci. **410**, 5467–5480 (2009). https://doi.org/10.1016/j.tcs.2009.05.006
19. Böcker, S., Briesemeister, S., Bui, Q.B.A., Truß, A.: A fixed-parameter approach for weighted cluster editing. In: Proceedings of APBC 2008, pp. 211–220 (2008). https://doi.org/10.1142/9781848161092_0023
20. Böcker, S., Briesemeister, S., Klau, G.W.: Exact algorithms for cluster editing: evaluation and experiments. Algorithmica **60**(2), 316–334 (2011). https://doi.org/10.1007/s00453-009-9339-7
21. Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM J. Comput. **25**(6), 1305–1317 (1996). https://doi.org/10.1137/S0097539793251219
22. Bodlaender, H.L., Downey, R.G., Fellows, M.R., Hermelin, D.: On problems without polynomial kernels. J. Comput. Syst. Sci. **75**(8), 423–434 (2009). https://doi.org/10.1016/j.jcss.2009.04.001
23. Bodlaender, H.L., Drange, P.G., Dregi, M.S., Fomin, F.V., Lokshtanov, D., Pilipczuk, M.: A $c^k n$ 5-approximation algorithm for treewidth. SIAM J. Comput. **45**(2), 317–378 (2016). https://doi.org/10.1137/130947374
24. Bodlaender, H.L., Heggernes, P., Villanger, Y.: Faster parameterized algorithms for MINIMUM FILL-IN. Algorithmica **61**(4), 817–838 (2010). https://doi.org/10.1007/s00453-010-9421-1
25. Bodlaender, H.L., Jansen, B.M.P., Kratsch, S.: Preprocessing for treewidth: a combinatorial analysis through kernelization. SIAM J. Discrete Math. **27**(4), 2108–2142 (2013). https://doi.org/10.1137/120903518
26. Bodlaender, H.L., Koster, A.M.: Safe separators for treewidth. Discrete Math. **306**(3), 337–350 (2006). https://doi.org/10.1016/j.disc.2005.12.017
27. Bodlaender, H.L., Koster, A.M., Eijkhof, F.V.d.: Preprocessing rules for triangulation of probabilistic networks. Comput. Intell. **21**(3), 286–305 (2005). https://doi.org/10.1111/j.1467-8640.2005.00274.x
28. Bonnet, É., Sikora, F.: The PACE 2018 parameterized algorithms and computational experiments challenge: the third iteration. In: Proceedings of IPEC 2018, pp. 26:1–26:15 (2018). https://doi.org/10.4230/LIPIcs.IPEC.2018.26

29. Bouchitté, V., Todinca, I.: Treewidth and minimum fill-in: grouping the minimal separators. SIAM J. Comput. **31**(1), 212–232 (2001). https://doi.org/10.1137/S0097539799359683
30. Bron, C., Kerbosch, J.: Algorithm 457: finding all cliques of an undirected graph. Comm. ACM **16**(9), 575–577 (1973). https://doi.org/10.1145/362342.362367
31. Buchbinder, N., Naor, J., Schwartz, R.: Simplex partitioning via exponential clocks and the multiway cut problem. SIAM J. Comput. **47**, 1463–1482 (2018). https://doi.org/10.1137/15M1045521
32. Cai, L., Chen, J., Downey, R.G., Fellows, M.R.: Advice classes of parameterized tractability. Ann. Pure Appl. Logic **84**(1), 119–138 (1997). https://doi.org/10.1016/S0168-0072(95)00020-8
33. Cai, S., Su, K., Luo, C., Sattar, A.: NuMVC: an efficient local search algorithm for minimum vertex cover. J. Artif. Intell. Res. **46**, 687–716 (2013). https://doi.org/10.1613/jair.3907
34. Cai, S.: Balance between complexity and quality: local search for minimum vertex cover in massive graphs. In: Proceedings of IJCAI 2015, pp. 747–753 (2015)
35. Cai, S., Hou, W., Lin, J., Li, Y.: Improving local search for minimum weight vertex cover by dynamic strategies. In: Proceedings of IJCAI 2018, pp. 1412–1418 (2018). https://doi.org/10.24963/ijcai.2018/196
36. Cai, S., Lin, J.: Fast solving maximum weight clique problem in massive graphs. In: Proceedings of IJCAI 2016, pp. 568–574 (2016)
37. Cai, S., Lin, J., Luo, C.: Finding a small vertex cover in massive sparse graphs: construct, local search, and preprocess. J. Artif. Intell. Res. **59**, 463–494 (2017). https://doi.org/10.1613/jair.5443
38. Cao, Y., Chen, J., Fan, J.H.: An $\mathscr{O}(1.84^k)$ parameterized algorithm for the multiterminal cut problem. Inf. Proc. Lett. **114**(4), 167–173 (2014). https://doi.org/10.1016/j.ipl.2013.12.001
39. Cao, Y., Sandeep, R.: Minimum fill-in: inapproximability and almost tight lower bounds. Inf. Comput. **271**, 104514 (2020). https://doi.org/10.1016/j.ic.2020.104514
40. Chang, L.: Efficient maximum clique computation over large sparse graphs. In: Proceedings of KDD 2019, pp. 529–538 (2019). https://doi.org/10.1145/3292500.3330986
41. Chang, L.: Efficient maximum clique computation and enumeration over large sparse graphs. VLDB J. **29**(5), 999–1022 (2020). https://doi.org/10.1007/s00778-020-00602-z
42. Chang, L., Li, W., Zhang, W.: Computing a near-maximum independent set in linear time by reducing-peeling. Proc. SIGMOD **2017**, 1181–1196 (2017). https://doi.org/10.1145/3035918.3035939
43. Chekuri, C., Goldberg, A.V., Karger, D.R., Levine, M.S., Stein, C.: Experimental study of minimum cut algorithms. In: Proceedings of SODA 1997, pp. 324–333 (1997)
44. Chen, J., Liu, Y., Lu, S.: An improved parameterized algorithm for the minimum node multiway cut problem. Algorithmica **55**(1), 1–13 (2009). https://doi.org/10.1007/s00453-007-9130-6
45. Chen, J., Meng, J.: A $2k$ kernel for the cluster editing problem. J. Comput. Syst. Sci. **78**(1), 211–220 (2012). https://doi.org/10.1016/j.jcss.2011.04.001
46. Conte, A., Firmani, D., Mordente, C., Patrignani, M., Torlone, R.: Fast enumeration of large k-plexes. In: Proceedings of KDD 2017, pp. 115–124 (2017). https://doi.org/10.1145/3097983.3098031

47 SPP. Crowston, R., Jones, M., Mnich, M.: Max-cut parameterized above the Edwards-Erdős bound. Algorithmica **72**(3), 734–757 (2014). https://doi.org/10.1007/s00453-014-9870-z

48. Cunningham, W.H.: The optimal multiterminal cut problem. In: Reliability of Computer and Communication Networks, pp. 105–120 (1989). https://doi.org/10.1090/dimacs/005/07
49. Cygan, M., et al.: Parameterized Algorithms. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21275-3
50. Dahlhaus, E., Johnson, D.S., Papadimitriou, C.H., Seymour, P.D., Yannakakis, M.: The complexity of multiterminal cuts. SIAM J. Comput. **23**(4), 864–894 (1994). https://doi.org/10.1137/S0097539792225297
51. Dahlum, J., Lamm, S., Sanders, P., Schulz, C., Strash, D., Werneck, R.F.: Accelerating local search for the maximum independent set problem. In: Goldberg, A.V., Kulikov, A.S. (eds.) SEA 2016. LNCS, vol. 9685, pp. 118–133. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-38851-9_9
52. Daneshmand, S.V.: Algorithmic approaches to the Steiner problem in networks. Ph.D. thesis, Universität Mannheim, Germany (2004). http://bibserv7.bib.uni-mannheim.de/madoc/volltexte/2004/176/index.html
53. Dehne, F., Langston, M.A., Luo, X., Pitre, S., Shaw, P., Zhang, Y.: The cluster editing problem: implementations and experiments. In: Bodlaender, H.L., Langston, M.A. (eds.) IWPEC 2006. LNCS, vol. 4169, pp. 13–24. Springer, Heidelberg (2006). https://doi.org/10.1007/11847250_2
54. Dell, H., Husfeldt, T., Jansen, B.M.P., Kaski, P., Komusiewicz, C., Rosamond, F.A.: The first parameterized algorithms and computational experiments challenge. In: Proceedings of IPEC 2016, LIPI, vol. 63, pp. 30:1–30:9 (2016). https://doi.org/10.4230/LIPIcs.IPEC.2016.30
55. Dell, H., Komusiewicz, C., Talmon, N., Weller, M.: The PACE 2017 parameterized algorithms and computational experiments challenge: the second iteration. In: Proceedings of IPEC 2017, LIPI, vol. 89, pp. 30:1–30:12 (2017)
56. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1999). https://doi.org/10.1007/978-1-4612-0515-9
57. Duff, I.S., Reid, J.K.: Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems. ACM Trans. Math. Softw. **22**(2), 227–257 (1996). https://doi.org/10.1145/229473.229480
58. Dzulfikar, M.A., Fichte, J.K., Hecher, M.: The PACE 2019 parameterized algorithms and computational experiments challenge: the fourth iteration (invited paper). In: Proceedings of IPEC 2019, LIPI, vol. 148, pp. 25:1–25:23 (2019). https://doi.org/10.4230/LIPIcs.IPEC.2019.25
59. Ebenegger, C., Hammer, P., De Werra, D.: Pseudo-boolean functions and stability of graphs. In: North-Holland mathematics studies, vol. 95, pp. 83–97 (1984). https://doi.org/10.1016/S0304-0208(08)72955-4
60. Eblen, J.D., Phillips, C.A., Rogers, G.L., Langston, M.A.: The maximum clique enumeration problem: algorithms, applications, and implementations. In: BMC Bioinformatics, p. S5 (2012). https://doi.org/10.1186/1471-2105-13-S10-S5
61. Edwards, C.S.: Some extremal properties of bipartite subgraphs. Can. J. Math. **25**(3), 475–485 (1973). https://doi.org/10.4153/CJM-1973-048-x
62. Edwards, C.: An improved lower bound for the number of edges in a largest bipartite subgraph. In: Recent Advances in Graph Theory, pp. 167–181 (1975)
63. van den Eijkhof, F., Bodlaender, H.L., Koster, A.M.C.A.: Safe reduction rules for weighted treewidth. Algorithmica **47**(2), 139–158 (2007). https://doi.org/10.1007/s00453-006-1226-x
64. Eppstein, D., Löffler, M., Strash, D.: Listing all maximal cliques in large sparse real-world graphs in near-optimal time. J. Exp. Algorithmics **18**, 3–1 (2013). https://doi.org/10.1145/2543629

65. Erickson, R.E., Monma, C.L., Jr., A.F.V.: Send-and-split method for minimum-concave-cost network flows. Math. Oper. Res. **12**(4), 634–664 (1987). https://doi.org/10.1287/moor.12.4.634

66 SPP. Etscheid, M., Mnich, M.: Linear kernels and linear-time algorithms for finding large cuts. Algorithmica **80**(9), 2574–2615 (2017). https://doi.org/10.1007/s00453-017-0388-z

67. Fafianie, S., Kratsch, S.: A shortcut to (sun)flowers: kernels in logarithmic space or linear time. In: Italiano, G.F., Pighizzini, G., Sannella, D.T. (eds.) MFCS 2015. LNCS, vol. 9235, pp. 299–310. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48054-0_25

68 SPP. Ferizovic, D., Hespe, D., Lamm, S., Mnich, M., Schulz, C., Strash, D.: Engineering kernelization for maximum cut. In: Proceedings of ALENEX 2020, pp. 27–41 (2020). https://doi.org/10.1137/1.9781611976007.3

69. Fomin, F.V., Grandoni, F., Kratsch, D.: A measure & conquer approach for the analysis of exact algorithms. J. ACM **56**(5), 25:1–25:32 (2009). https://doi.org/10.1145/1552285.1552286

70. Fomin, F.V., Lokshtanov, D., Saurabh, S., Zehavi, M.: Kernelization: Theory of Parameterized Preprocessing. Cambridge University Press, Cambridge (2019). https://doi.org/10.1017/9781107415157

71. Fomin, F.V., Villanger, Y.: Subexponential parameterized algorithm for minimum fill-in. SIAM J. Comput. **42**(6), 2197–2216 (2013). https://doi.org/10.1137/11085390X

72. Gao, J., Chen, J., Yin, M., Chen, R., Wang, Y.: An exact algorithm for maximum k-plexes in massive graphs. In: Proceedings of IJCAI 2018, pp. 1449–1455 (2018). https://doi.org/10.24963/ijcai.2018/201

73. Gao, W., Friedrich, T., Kötzing, T., Neumann, F.: Scaling up local search for minimum vertex cover in large graphs by parallel kernelization. In: Proceedings of ACAI 2017, pp. 131–143 (2017). https://doi.org/10.1007/978-3-319-63004-5_11

74. Garey, M.R., Johnson, D.S., Stockmeyer, L.: Some simplified NP-complete problems. In: Proceedings of STOC 1974, pp. 47–63 (1974). https://doi.org/10.1145/800119.803884

75. Garey, M.R., Johnson, D.S.: Computers and Intractability. W. H. Freeman and Co., San Francisco, Calif. (1979). A Guide to the Theory of NP-Completeness

76. Gawrychowski, P., Mozes, S., Weimann, O.: Minimum cut in $\mathcal{O}(m \log^2 n)$ time. In: Proceedings of ICALP 2020, LIPI, vol. 168, pp. 57:1–57:15 (2020). https://doi.org/10.4230/LIPIcs.ICALP.2020.57

77. Gellner, A., Lamm, S., Schulz, C., Strash, D., Zaválnij, B.: Boosting data reduction for the maximum weight independent set problem using increasing transformations. In: Proceedings of ALENEX 2021, pp. 128–142. https://doi.org/10.1137/1.9781611976472.10

78. George, A.: Nested dissection of a regular finite element mesh. SIAM J. Numer. Anal. **10**(2), 345–363 (1973). https://doi.org/10.1137/0710032

79. George, A., Liu, J.W.: The evolution of the minimum degree ordering algorithm. SIAM Rev. **31**(1), 1–19 (1989). https://doi.org/10.1137/1031001

80. Goel, A., Kapralov, M., Khanna, S.: Perfect matchings in $\mathcal{O}(n \log n)$ time in regular bipartite graphs. SIAM J. Comput. **42**(3), 1392–1404 (2013). https://doi.org/10.1137/100812513

81. Gomory, R.E., Hu, T.C.: Multi-terminal network flows. J. Soc. Ind. Appl. Math. **9**(4), 551–570 (1961). https://doi.org/10.1137/0109047

82. Gramm, J., Guo, J., Hüffner, F., Niedermeier, R.: Graph-modeled data clustering: fixed-parameter algorithms for clique generation. In: Petreschi, R., Persiano, G., Silvestri, R. (eds.) CIAC 2003. LNCS, vol. 2653, pp. 108–119. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44849-7_17
83. Guo, J.: A more effective linear kernelization for cluster editing. Theor. Comput. Sci. **410**(8), 718–726 (2009). https://doi.org/10.1016/j.tcs.2008.10.021
84. Hao, J., Orlin, J.B.: A faster algorithm for finding the minimum cut in a graph. In: Proceedings of SODA 1992, pp. 165–174 (1992)
85. Heggernes, P., Lokshtanov, D., Nederlof, J., Paul, C., Telle, J.A.: Generalized graph clustering: recognizing (p,q)-cluster graphs. In: Thilikos, D.M. (ed.) WG 2010. LNCS, vol. 6410, pp. 171–183. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16926-7_17
86. Henzinger, M., Noe, A., Schulz, C.: Shared-memory exact minimum cuts. In: Proceedings of IPDPS 2019, pp. 13–22 (2019). https://doi.org/10.1109/IPDPS.2019.00013
87 SPP. Henzinger, M., Noe, A., Schulz, C.: Faster parallel multiterminal cuts. Technical report (2020). https://arxiv.org/abs/2004.11666
88. Henzinger, M., Noe, A., Schulz, C.: Shared-memory branch-and-reduce for multiterminal cuts. In: Proceedings of ALENEX 2020, pp. 42–55 (2020). https://doi.org/10.1137/1.9781611976007.4
89. Henzinger, M., Noe, A., Schulz, C., Strash, D.: Practical minimum cut algorithms. ACM J. Exp. Algorithmics **23** (2018). https://doi.org/10.1145/3274662
90 SPP. Henzinger, M., Noe, A., Schulz, C., Strash, D.: Finding all global minimum cuts in practice. In: Proceedings of ESA 2020, pp. 59:1–59:20 (2020). https://doi.org/10.4230/LIPIcs.ESA.2020.59
91. Henzinger, M., Rao, S., Wang, D.: Local flow partitioning for faster edge connectivity. SIAM J. Comput. **49**(1), 1–36 (2020). https://doi.org/10.1137/18M1180335
92. Hespe, D., Lamm, S., Schulz, C., Strash, D.: WeGotYouCovered: the winning solver from the PACE 2019 challenge, vertex cover track. In: Proceedings of CSC 2020, pp. 1–11 (2020). https://doi.org/10.1137/1.9781611976229.1
93. Hespe, D., Schulz, C., Strash, D.: Scalable kernelization for maximum independent sets. J. Exp. Algor. **24**(1), 1–22 (2019). https://doi.org/10.1145/3355502
94. Holtgrewe, M., Sanders, P., Schulz, C.: Engineering a scalable high quality graph partitioner. In: Proceedings of IPDPS 2010, pp. 1–12 (2010). https://doi.org/10.1109/IPDPS.2010.5470485
95. Iwata, Y., Oka, K., Yoshida, Y.: Linear-time FPT algorithms via network flow. In: Proceedings of SODA 2014, pp. 1749–1761 (2014). https://doi.org/10.1137/1.9781611973402.127
96. Iwata, Y., Shigemura, T.: Separator-based pruned dynamic programming for Steiner tree. In: Proceedings of AAAI 2019, pp. 1520–1527 (2019). https://doi.org/10.1609/aaai.v33i01.33011520
97. Jaffke, L., Jansen, B.M.P.: Fine-grained parameterized complexity analysis of graph coloring problems. In: Fotakis, D., Pagourtzis, A., Paschos, V.T. (eds.) CIAC 2017. LNCS, vol. 10236, pp. 345–356. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57586-5_29
98. Jansen, B.M.P.: On sparsification for computing treewidth. Algorithmica **71**(3), 605–635 (2014). https://doi.org/10.1007/s00453-014-9924-2
99. Jansen, B.M.P., Pieterse, A.: Optimal data reduction for graph coloring using low-degree polynomials. Algorithmica **81**(10), 3865–3889 (2019). https://doi.org/10.1007/s00453-019-00578-5
100. Jiang, H., Li, C., Manyà, F.: An exact algorithm for the maximum weight clique problem in large graphs. In: Proceedings of AAAI 2017, pp. 830–838 (2017)

101. Jünger, M., Rinaldi, G., Thienel, S.: Practical performance of efficient minimum cut algorithms. Algorithmica **26**(1), 172–195 (2000). https://doi.org/10.1007/s004539910009
102. Kaplan, H., Shamir, R., Tarjan, R.E.: Tractability of parameterized completion problems on chordal, strongly chordal, and proper interval graphs. SIAM J. Comput. **28**(5), 1906–1922 (1999). https://doi.org/10.1137/S0097539796303044
103. Kaplan, H., Shamir, R., Tarjan, R.E.: Tractability of parameterized completion problems on chordal and interval graphs: minimum fill-in and physical mapping. In: Proceedings of FOCS 1994, pp. 780–791 (1994). https://doi.org/10.1109/SFCS.1994.365715
104. Karger, D.R.: Minimum cuts in near-linear time. J. ACM **47**(1), 46–76 (2000). https://doi.org/10.1145/331605.331608
105. Karger, D.R., Stein, C.: A new approach to the minimum cut problem. J. ACM **43**(4), 601–640 (1996). https://doi.org/10.1145/234533.234534
106. Karp, R.M., Kan, A.H.G.R., Vohra, R.V.: Average case analysis of a heuristic for the assignment problem. Math. Oper. Res. **19**(3), 513–522 (1994). https://doi.org/10.1287/moor.19.3.513
107. Karp, R.M., Sipser, M.: Maximum matchings in sparse random graphs. In: Proceedings of FOCS 1981, pp. 364–375 (1981). https://doi.org/10.1109/SFCS.1981.21
108. Kaya, K., Langguth, J., Panagiotas, I., Uçar, B.: Karp-Sipser based kernels for bipartite graph matching. In: Proceedings of ALENEX 2020, pp. 134–145 (2020). https://doi.org/10.1137/1.9781611976007.11
109. Kobayashi, Y., Tamaki, H.: Treedepth parameterized by vertex cover number. In: Proceedings of IPEC 2016, LIPI, vol. 63, pp. 18:1–18:11 (2016). https://doi.org/10.4230/LIPIcs.IPEC.2016.18
110. Komusiewicz, C., Uhlmann, J.: Cluster editing with locally bounded modifications. Discrete Appl. Math. **160**(15), 2259–2270 (2012). https://doi.org/10.1016/j.dam.2012.05.019
111. Korenwein, V., Nichterlein, A., Niedermeier, R., Zschoche, P.: Data reduction for maximum matching on real-world graphs: theory and experiments. In: Proceedings of ESA 2018, LIPI, vol. 112, pp. 53:1–53:13 (2018). https://doi.org/10.4230/LIPIcs.ESA.2018.53
112. Korhonen, T.: SMS in PACE 2020. Technical report (2020). https://arxiv.org/abs/2006.07302
113 SPP. Lamm, S., Sanders, P., Schulz, C.: Graph partitioning for independent sets. In: Bampis, E. (ed.) SEA 2015. LNCS, vol. 9125, pp. 68–81. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20086-6_6
114. Lamm, S., Sanders, P., Schulz, C., Strash, D., Werneck, R.F.: Finding near-optimal independent sets at scale. J. Heurist. **23**(4), 207–229 (2017). https://doi.org/10.1007/s10732-017-9337-x
115 SPP. Lamm, S., Schulz, C., Strash, D., Williger, R., Zhang, H.: Exactly solving the maximum weight independent set problem on large real-world graphs. In: Proceedings of ALENEX 2019, pp. 144–158 (2019). https://doi.org/10.1137/1.9781611975499.12
116. Lange, J.H., Andres, B., Swoboda, P.: Combinatorial persistency criteria for multicut and max-cut. In: Proceedings of IEEE Conference Computer Vision Pattern Recognition, pp. 6093–6102 (2019). https://doi.org/10.1109/CVPR.2019.00625
117. Langguth, J., Manne, F., Sanders, P.: Heuristic initialization for bipartite matching problems. ACM J. Exp. Algorithmics **15** (2010). https://doi.org/10.1145/1671970.1712656

118. Lavallee, B., Russell, H., Sullivan, B.D., van der Poel, A.: Approximating vertex cover using structural rounding. In: Proceedings of ALENEX 2020, pp. 70–80 (2020). https://doi.org/10.1137/1.9781611976007.6
119. Li, C.M., Jiang, H., Manyà, F.: On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem. Comput. Oper. Res. **84**, 1–15 (2017). https://doi.org/10.1016/j.cor.2017.02.017
120. Li, R., Hu, S., Cai, S., Gao, J., Wang, Y., Yin, M.: NuMWVC: a novel local search for minimum weighted vertex cover problem. J. Oper. Res. Soc., 1–12 (2019). https://doi.org/10.1080/01605682.2019.1621218
121. Lin, J., Cai, S., Luo, C., Su, K.: A reduction based method for coloring very large graphs. In: Proceedings of IJCAI 2017, pp. 517–523 (2017). https://doi.org/10.24963/ijcai.2017/73
122. Lokshtanov, D., Marx, D., Saurabh, S.: Known algorithms on graphs of bounded treewidth are probably optimal. ACM Trans. Algor. **14**(2) (2018). https://doi.org/10.1145/3170442
123. Marx, D.: Parameterized graph separation problems. Theor. Comput. Sci. **351**(3), 394–406 (2006). https://doi.org/10.1016/j.tcs.2005.10.007
124. Matula, D.W.: A linear time $2+\varepsilon$ approximation algorithm for edge connectivity. In: Proceedings of SODA 1993, pp. 500–504 (1993)
125. Mellor, D., Prieto-Rodríguez, E., Mathieson, L., Moscato, P.A.: A kernelisation approach for multiple d-hitting set and its application in optimal multi-drug therapeutic combinations. PLoS ONE **5**, 1–13 (2010)
126. Méndez-Díaz, I., Zabala, P.: A branch-and-cut algorithm for graph coloring. Discrete Appl. Math. **154**(5), 826–847 (2006). https://doi.org/10.1016/j.dam.2005.05.022
127. Mertzios, G.B., Nichterlein, A., Niedermeier, R.: The power of linear-time data reduction for maximum matching. Algorithmica **82**(12), 3521–3565 (2020). https://doi.org/10.1007/s00453-020-00736-0
128. Möhring, R., Müller-Hannemann, M.: Cardinality matching: heuristic search for augmenting paths. Technical Report 439, Technische Universität Berlin, Fachbereich 3 (1995)
129. Moser, H.: Finding optimal solutions for covering and matching problems. Ph.D. thesis, Friedrich-Schiller-Universität Jena (2010). http://d-nb.info/999819399
130. Nagamochi, H., Ibaraki, T.: Computing edge-connectivity in multigraphs and capacitated graphs. SIAM J. Discrete Math. **5**(1), 54–66 (1992). https://doi.org/10.1137/0405004
131. Nagamochi, H., Ono, T., Ibaraki, T.: Implementing an efficient minimum capacity cut algorithm. Math. Prog. **67**(1), 325–341 (1994). https://doi.org/10.1007/BF01582226
132. Natanzon, A., Shamir, R., Sharan, R.: A polynomial approximation algorithm for the minimum fill-in problem. SIAM J. Comput. **30**(4), 1067–1079 (2000). https://doi.org/10.1137/S0097539798336073
133. Nemhauser, G., Trotter, L.E., J.: Vertex packings: structural properties and algorithms. Math. Prog. **8**(1), 232–248 (1975). https://doi.org/10.1007/BF01580444
134. Niedermeier, R., Rossmanith, P.: An efficient fixed-parameter algorithm for 3-hitting set. J. Discrete Algor. **1**(1), 89–102 (2003). https://doi.org/10.1016/S1570-8667(03)00009-1
135. Bastos, L., Ochi, L.S., Protti, F., Subramanian, A., Martins, I.C., Pinheiro, R.G.S.: Efficient algorithms for cluster editing. J. Comb. Optim. **31**(1), 347–371 (2014). https://doi.org/10.1007/s10878-014-9756-7
136. Olesen, K.G., Madsen, A.L.: Maximal prime subgraph decomposition of Bayesian networks. IEEE Trans. Syst. Man Cybern. Part B (Cybern.) **32**(1), 21–31 (2002). https://doi.org/10.1109/3477.979956

137 SPP. 1 Ost, W., Schulz, C., Strash, D.: Engineering data reduction for nested dissection. In: Proceedings of ALENEX 2021, pp. 113–127 (2021). https://doi.org/10.1137/1.9781611976472.9
138. Padberg, M., Rinaldi, G.: An efficient algorithm for the minimum capacity cut problem. Math. Prog. **47**(1), 19–36 (1990). https://doi.org/10.1007/BF01580850
139. Panagiotas, I., Uçar, B.: Engineering fast almost optimal algorithms for bipartite graph matching: Extended version. Research Report RR-9321, Inria Research Centre Grenoble, Rhône-Alpes (2020). https://hal.inria.fr/hal-02463717
140. Pelofske, E., Hahn, G., Djidjev, H.: Solving large minimum vertex cover problems on a quantum annealer. In: Proceedings of CF 2019, pp. 76–84 (2019). https://doi.org/10.1145/3310273.3321562
141. Polzin, T.: Algorithms for the Steiner problem in networks. Ph.D. thesis, Universität des Saarlandes, Saarbrücken, Germany (2003). http://scidok.sulb.uni-saarland.de/volltexte/2004/218/index.html
142. Pothen, A.: The complexity of optimal elimination trees. Technical report, Pennsylvania State University, Department of Computer Science (1988). https://www.cs.purdue.edu/homes/apothen/Papers/shortest-etree1988.pdf
143. Rehfeldt, D., Koch, T.: SCIP-Jack - a solver for STP and variants with parallelization extensions: an update. In: Proceedings of OR 2017, pp. 191–196 (2017). https://doi.org/10.1007/978-3-319-89920-6_27
144. Rehfeldt, D., Koch, T., Maher, S.J.: Reduction techniques for the prize collecting Steiner tree problem and the maximum-weight connected subgraph problem. Networks **73**(2), 206–233 (2019). https://doi.org/10.1002/net.21857
145. Reidl, F., Rossmanith, P., Villaamil, F.S., Sikdar, S.: A faster parameterized algorithm for treedepth. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) ICALP 2014. LNCS, vol. 8572, pp. 931–942. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43948-7_77
146. Robertson, N., Seymour, P.: Graph minors. II. Algorithmic aspects of tree-width. J. Algor. **7**(3), 309–322 (1986). https://doi.org/10.1016/0196-6774(86)90023-4
147. Rose, D.J.: Triangulated graphs and the elimination process. J. Math. Anal. Appl. **32**(3), 597–609 (1970). https://doi.org/10.1016/0022-247X(70)90282-9
148. Sanders, P., Schulz, C.: KaHIP v3.00 - Karlsruhe High Quality Partitioning - User Guide. Technical report (2013). https://arxiv.org/abs/1311.1714
149. Schäffer, A.A.: Optimal node ranking of trees in linear time. Inf. Proc. Lett. **33**(2), 91–96 (1989). https://doi.org/10.1016/0020-0190(89)90161-0
150 SPP. Schulz, C.: Scalable Graph Algorithms. Habilitation (2019). http://arxiv.org/abs/1912.00245
151. Seidman, S.B., Foster, B.L.: A graph-theoretic generalization of the clique concept. J. Math. Sociol. **6**(1), 139–154 (1978). https://doi.org/10.1080/0022250X.1978.9989883
152. Shinano, Y., Rehfeldt, D., Koch, T.: Building optimal steiner trees on supercomputers by using up to 43,000 cores. In: Rousseau, L.-M., Stergiou, K. (eds.) CPAIOR 2019. LNCS, vol. 11494, pp. 529–539. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-19212-9_35
153. Stallmann, M.F., Ho, Y., Goodrich, T.D.: Graph profiling for vertex cover: targeted reductions in a branch and reduce solver. Technical report (2020). https://arxiv.org/abs/2003.06639
154. Stoer, M., Wagner, F.: A simple min-cut algorithm. J. ACM **44**(4), 585–591 (1997). https://doi.org/10.1145/263867.263872
155. Strash, D.: On the power of simple reductions for the maximum independent set problem. In: Dinh, T.N., Thai, M.T. (eds.) Proccedings of COCOON 2016. LNCS, vol. 9797, pp. 345–356. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42634-1_28

156. Tamaki, H.: Positive-instance driven dynamic programming for treewidth. In: Proceedings of ESA 2017, LIPI, vol. 87, pp. 68:1–68:13 (2017). https://doi.org/10.4230/LIPIcs.ESA.2017.68
157. Tarjan, R.E., Yannakakis, M.: Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. SIAM J. Comput. **13**(3), 566–579 (1984). https://doi.org/10.1137/0213035
158. Tarjan, R.E., Yannakakis, M.: Addendum: simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. SIAM J. Comput. **14**(1), 254–255 (1985). https://doi.org/10.1137/0214020
159. Tinney, W.F., Walker, J.W.: Direct solutions of sparse network equations by optimally ordered triangular factorization. Proc. IEEE **55**(11), 1801–1809 (1967). https://doi.org/10.1109/PROC.1967.6011
160. Tomita, E., Kameda, T.: An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. J. Glob. Optim. **37**(1), 95–111 (2007). https://doi.org/10.1007/s10898-006-9039-7
161. Trimble, J.: An algorithm for the exact treedepth problem. In: Proceedings of SEA 2020, LIPI, vol. 160, pp. 19:1–19:14 (2020). https://doi.org/10.4230/LIPIcs.SEA.2020.19
162. Bevern, R.: Towards optimal and expressive kernelization for d-hitting set. Algorithmica **70**(1), 129–147 (2013). https://doi.org/10.1007/s00453-013-9774-3
163. van Bevern, R., Smirnov, P.V.: Optimal-size problem kernels for d-hitting set in linear time and space. Inf. Process. Lett. **163**, 105998 (2020). https://doi.org/10.1016/j.ipl.2020.105998
164. Verma, A., Buchanan, A., Butenko, S.: Solving the maximum clique and vertex coloring problems on very large sparse networks. INFORMS J. Comput. **27**(1), 164–177 (2015). https://doi.org/10.1287/ijoc.2014.0618
165. Wang, L., Li, C.M., Zhou, J., Jin, B., Yin, M.: An exact algorithm for minimum weight vertex cover problem in large graphs. Technical report (2019). https://urldefense.com/v3/__https://www.mdpi.com/2227-7390/7/7/603__;!!NLFGqXoFfo8MMQ!ryv0VjrmlwLawl0j6PQDtgV3XzU7mM4U8uFD6oX3d4bPcT9yMMYD958fi7tNg1IaVc81OzW7E7AEb5NnCFGAplRjt2vxhvOs
166. Weihe, K.: Covering trains by stations or the power of data reduction. In: Proceedings of ALEX 1998, pp. 1–8 (1998)
167. Xiao, M.: Simple and improved parameterized algorithms for multiterminal cuts. Theory Comput. Syst. **46**(4), 723–736 (2010). https://doi.org/10.1007/s00224-009-9215-5
168. Xiao, M., Lin, W., Dai, Y., Zeng, Y.: A fast algorithm to compute maximum k-plexes in social network analysis. In: Proceedings of AAAI 2017, pp. 919–925 (2017)
169. Xiao, M., Nagamochi, H.: Confining sets and avoiding bottleneck cases: a simple maximum independent set algorithm in degree-3 graphs. Theor. Comput. Sci. **469**, 92–104 (2013). https://doi.org/10.1016/j.tcs.2012.09.022
170. Xiao, M., Nagamochi, H.: Exact algorithms for maximum independent set. Inf. Comput. **255**, 126–146 (2017). https://doi.org/10.1016/j.ic.2017.06.001
171. Yannakakis, M.: Computing the minimum fill-in is NP-complete. SIAM J. Algeb. Discrete Meth. **2**(1), 77–79 (1981). https://doi.org/10.1137/0602010

172. Zheng, W., Gu, J., Peng, P., Yu, J.X.: Efficient weighted independent set computation over large graphs. In: Proceedings of ICDE 2020, pp. 1970–1973 (2020). https://doi.org/10.1109/ICDE48307.2020.00216
173. Zuckerman, D.: Linear degree extractors and the inapproximability of max clique and chromatic number. Theory Comput. **3**(1), 103–128 (2007). https://doi.org/10.4086/toc.2007.v003a006

Skeleton-Based Clustering by Quasi-Threshold Editing

Ulrik Brandes[1], Michael Hamann[2(✉)], Luise Häuser[2], and Dorothea Wagner[2]

[1] ETH Zürich, Zürich, Switzerland
ubrandes@ethz.ch
[2] Karlsruhe Institute of Technology, Karlsruhe, Germany
michael@content-space.de, ufziw@student.kit.edu, dorothea.wagner@kit.edu

Abstract. We consider the problem of transforming a given graph into a quasi-threshold graph using a minimum number of edge additions and deletions. Building on the previously proposed heuristic Quasi-Threshold Mover (QTM), we present improvements both in terms of running time and quality. We propose a novel, linear-time algorithm that solves the inclusion-minimal variant of this problem, i.e., a set of edge edits such that no subset of them also transforms the given graph into a quasi-threshold graph. In an extensive experimental evaluation, we apply these algorithms to a large set of graphs from different applications and find that they lead QTM to find solutions with fewer edits. Although the inclusion-minimal algorithm needs significantly more edits on its own, it outperforms the initialization heuristic previously proposed for QTM.

Keywords: Quasi-threshold graph · Trivially perfect graph · Graph editing · Graph clustering · Community detection

1 Introduction

We consider the problem of clustering a graph by partitioning its nodes. Especially in the context of social networks, this problem is often referred to as community detection. The approach taken here is to view community detection as a graph modification problem. Specifically, Nastos and Gao [25] proposed to edit a given graph into a quasi-threshold graph and use its connected components to determine the clustering.

A quasi-threshold graph, also known as trivially perfect graph, is the transitive closure of a rooted forest [33], which can in turn be considered a skeleton of the graph. Figure 1 shows an example, and we provide a more detailed motivation for this particular approach in the next section.

As minimizing the number of edits is $\mathcal{NP}$-hard [25], the Quasi-Threshold Mover (QTM) heuristic [4 SPP] starts from some rooted forest on the nodes of the input graph and moves nodes within and between trees to reduce the edit distance between the input graph and the transitive closure of the forest.

Several improvements to QTM are proposed in this chapter. We reduce the running time of one round of node moves to linear and show that the edits incident to a single node can be minimized using an additional path sorting step. This ultimately

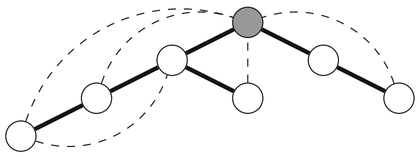


Fig. 1. Example quasi-threshold graph. The skeleton is denoted by thick edges, its transitive closure is dashed, the root is the gray node. (Color figure online)

leads to a linear-time algorithm for inclusion-minimal sets of edits. To also find smaller solutions, we propose a randomization of local moves. From an extensive experimental evaluation on empirical graphs we conclude that our modifications yields substantial improvements over the original QTM algorithm in terms of the size of the edit set.

2 Preliminaries

We consider simple undirected graphs $G = (V,E)$ consisting of $n := |V|$ nodes V that are connected by a set of $m := |E|$ edges $E \subset \binom{V}{2}$, i.e., without self-loops or multi-edges. By $N(u)$ we denote the set of neighbors of $u \in V$ and $\deg(u) := |N(u)|$ its degree. Further, let $N[u] := N(u) \cup \{u\}$ be the closed neighborhood of u. The subgraph induced by a set of nodes $X \subset V$ is denoted $G[X]$. With K_n, P_n, and C_n we denote the complete graph, path, and cycle on n nodes, respectively. These will be important as induced subgraphs, and we write, say, kK_n for k copies of K_n.

Quasi-threshold graphs are graphs that contain neither a P_4 nor a C_4 as node-induced subgraphs [34]. This is equivalent to an inductive construction in which the base case is a single node and there are two construction operators: either a universal node (adjacent to all previous nodes) is added, or the disjoint union of two quasi-threshold graphs is formed. The inductive construction of a quasi-threshold graph immediately gives rise to its skeleton forest referred to in the introduction.

2.1 Motivation

Many tasks in network analysis can be understood as first establishing an ideal, and then recovering that ideal from an empirical situation or at least determining a degree to which that ideal is met.

Take the most elementary notion, network density, as an example. The two idealized situations, polar opposites of one another, are graphs nK_1 of isolated nodes and cliques K_n[1]. The number of edges in a graph is a straightforward measure of distance from the ideal case of isolated nodes on an absolute scale of measurement. Since the number $\binom{n}{2}$ of edges in a clique varies with the number nodes n, *density* is often defined as the relative number $m/\binom{n}{2}$ of edges.

[1] It should not be lost on the reader that both names have social connotations [22].

A formulation of community detection using the same kind of reasoning can be developed as follows. Motivations for the vast majority of community detection methods generally state that the intention is to partition a graph into relatively dense subgraphs that are sparsely connected between them [17,28]. The idealized situation, with an undisputed partition into communities, is a *cluster graph* defined as disjoint unions of cliques or, equivalently, P_3-free graph. Each connected component is a clique, and these cliques are isolated from each other.

How far is a given graph from a cluster graph, and where are its communities? On an absolute scale, distance to the ideal situation is measured by counting the number of edges that need to be added or deleted to complete cliques and make those cliques independent. In *cluster editing*, a cluster graph of minimum edit distance is sought, and its cliques induce a clustering of the original graph. The normalized number of edges that do not have to be edited is known as *performance* [13], and cluster editing is a special case of *correlation clustering* [1]. Numerous other clustering approaches are based on objective functions that normalize the difference between a graph and the cluster graph ideal by taking additional factors such as the number of clusters, size of clusters, degree in clusters, etc. into account.

Like cluster graphs, quasi-threshold graphs represent an idealized situation, which we can think of as intersecting communities. To see this, we take two additional steps.

We start from *split graphs*, which are defined as those graphs that have a partition $V = C \uplus P$ into a clique $G[C]$ and an independent set $G[P]$, or, equivalently, $(2K_2, C_4, C_5)$-free graphs. They represent the ideal case of a core-periphery structure [3], and are characterized by their degrees: if $n > d_1 \geq \cdots \geq d_n \geq 0$ is the degree sequence of a graph, then it is a split graph if and only if the kth Erdős-Gallai inequality $\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{j=k+1}^{n} \min\{k, d_j\}$ is actually an equality for the *corrected Durfee number* $h = \max\{k : d_k \geq k-1\}$. In this case, h nodes of highest degree induce the clique and the others an independent set. The minimum number of edges that need to be edited to turn a graph into a split graph, known as its *splittance* [19], is half the difference between the two sides of the defining inequality, $\frac{1}{2}(\sum_{j=h+1}^{n} d_j - \sum_{i=1}^{h} d_i)$. These edits can be chosen so that h nodes of highest degree induce the clique and the remaining nodes the independent set. The computationally easy problem of split editing becomes intractable, for instance, if adapted for density [5] instead of edge numbers or for multiple cores [6].

By distinguishing a core from a periphery, split graphs also distinguish nodes that are central (as members of the core) from others that are not (as members of the periphery). Every node in the periphery is adjacent only to nodes in the core, and every node in the core is adjacent to all other core nodes. Hence, the neighborhood of any periphery node $u \in P$ is a subset of the closed neighborhood of any core node $v \in C$, $N(u) \subseteq N[v]$. This binary classification can be refined by comparing all pairs of nodes according to this neighborhood inclusion property, known as the vicinal preorder [16]. Schoch and Brandes [30] have shown retrospectively that this is actually the common ground of standard notions of centrality.

Graphs characterized by a total vicinal preorder are called *threshold* or *nested graphs* [23,24] and therefore the ideal structure of an undisputed ranking of nodes by centrality. Threshold editing is intractable, even if the input is a split graph [14], and for a number of reasons centrality has been defined via an abundance of indices rather than the node ranking in a closest threshold graph.

Threshold graphs are the $(2K_2, C_4, P_4)$-free graphs and therefore a subclass of quasi-threshold graphs. They can be constructed by adding one node at a time, either as a universal or isolated node, so that they have a skeleton that is a caterpillar. Each connected component of a quasi-threshold graph, in turn, can be seen as a group of nested graphs that intersect at their cores but may branch out into different peripheries.

We have thus motivated quasi-threshold graphs as idealized structures of (disjoint groups of) intersecting communities. Quasi-threshold editing yields a partition into communities and in addition for each of them a centralized nesting structure represented by their skeleton tree.

2.2 Related Work

Quasi-threshold graphs can be recognized in linear time [8, 34, 4 SPP]. While the first algorithm [34] computes a skeleton forest if G is a quasi-threshold graph, the other [8, 4 SPP] additionally computes a forbidden subgraph if G is not.

As mentioned, quasi-threshold editing is $\mathscr{NP}$-hard [25]. Due to its characterization via a finite set of finite forbidden subgraphs, it is fixed-parameter tractable in the number of edits k [7]. In combination with the certifying recognition in linear time, this yields a simple $\mathscr{O}(6^k \times (n+m))$ time algorithm. For the related problem of quasi-threshold deletion, where edges may be deleted but not added, improved branching rules have been proposed, reducing the running time from $\mathscr{O}(4^k \times (n+m))$ to $\mathscr{O}(2.42^k \times (n+m))$ [21]. Further, ordered enumeration of solutions is also possible with FPT delay [10]. A polynomial kernel of $\mathscr{O}(k^7)$ nodes has been introduced by Drange and Pilipczuk [15], who also show that the problem cannot be solved in time $2^{ok} \times n^{\mathscr{O}(1)}$ unless the Exponential Time Hypothesis fails.

The first editing heuristic has been proposed by Nastos and Gao [25]. With Quasi-Threshold Mover [4 SPP], the first editing heuristic with a running time close to linear has been proposed. Recently, a study on techniques for computing exact solutions has been published [18 SPP].

For the superclass of cographs, or P_4-free graphs [9], the problem of inclusion-minimal editing has recently been considered [11]. Instead of asking for a set of edge edits of minimum cardinality, it asks for a set of edge edits such that no proper subset yields a cograph. While cograph editing is also $\mathscr{NP}$-hard [20], inclusion-minimal cograph editing can be solved in linear time [11].

3 Quasi-Threshold Mover (QTM)

The Quasi-Threshold Mover algorithm, short QTM, iteratively improves the skeleton forest to heuristically minimize the number of induced edits. It starts with a given skeleton forest, this may be the trivial skeleton where every node is a root which implies that all edges are deleted. In each round, it iterates over all nodes u in a random order and possibly moves u to a new position in the forest if this decreases the number of induced edits. For this, it considers every node $v \in V \setminus \{u\}$ as a parent for u. Further, a subset of the children of the new parent v may be adopted, i.e., moved below u. In the induced quasi-threshold graph, u is then connected to v and all its ancestors as well as to all adopted children and their descendants. Every neighbor x of u in this set of nodes saves deleting the edges $\{u,x\}$ but every non-neighbor y of u implies inserting the edge $\{u,y\}$. Therefore, we select the parent v and the children to adopt such that the number of u-neighbors minus non-neighbors is maximized. Given a potential parent v, we always adopt children whose subtrees contain more neighbors than non-neighbors of u. We call those children *close children*. Using a DFS, we could determine for every node how many neighbors and non-neighbors are above/below that node and thus allowing the selection of the best parent and which children are close. However, this gives a quadratic running time per round. Instead, QTM starts limited local searches starting from the neighbors of u. They only visit one or two non-neighbors per neighbor. The idea is that whenever a subtree contains more neighbors than non-neighbors, it will be fully visited. Thus, the algorithm is able to determine all close children. Similarly, the best parent is determined by propagating information upwards in the skeleton. As QTM uses a priority queue to manage nodes during this bottom-up search, the running time per round is $\mathcal{O}(n + m \log \Delta)$, where Δ is the maximum degree.

In the following, we present several novel improvements for QTM. In Sect. 3.1, we show how to reduce the running time per round to linear in the number of nodes and edges. Further, in Sect. 3.2, we present an additional path sorting step that modifies the skeleton forest before every local move of a node u and yields a move that is optimal with respect to the edits incident to u. This local optimality directly gives us an inclusion-minimal algorithm as we show in Sect. 3.3. The last improvement is randomization, in Sect. 3.4, we show how we can select uniformly at random among all possible sets of edits incident to the moved node u.

3.1 Linear Running Time

To realize its bottom-up search, QTM needs to process nodes ordered by depth in the forest. While it is straightforward to use a bucket per level in the forest, this has the problem that this yields a running time linear in the depth of the deepest neighbor of u. It turns out, though, that we do not need to consider deep neighbors. More precisely, we show that we can ignore neighbors at a depth of more than $2 \times \deg(u)$. Consider a node $v \in N(u)$ with depth larger than $2 \times \deg(u)$. If u is connected to v in the edited graph, this implies that u is also connected to all at least $2 \times \deg(u)$ ancestors of u. Among these, there can be at most the remaining $\deg(u) - 1$ neighbors and thus at least $\deg(u) + 1$ non-neighbors. Thus, this implies $\deg(u) + 1$ edge insertions. Making u a root in the forest, i.e., deleting all edges incident to u, causes just $\deg(u)$ edits and is thus better. Therefore, we can ignore neighbors with depth larger than $2 \times \deg(u)$. We can thus use a bucket per depth of the remaining neighbors which eliminates the log-factor of the running time.

3.2 Sorting Simple Paths

QTM minimizes edits with respect to the choice of a parent and adopted children of that parent. Here we show that an additional sorting step minimizes the edits incident to u

in the edited graph independently of the chosen skeleton forest. For this, we consider *simple paths*, which we define as a maximal path in the skeleton forest in which each node has exactly one child except for the lowest node. Every node is thus part of exactly one simple path, which may only consist of the node itself. A crucial observation is that reordering nodes in simple paths is the only way the skeleton forest can be modified without affecting the induced quasi-threshold graph.

Lemma 1. *Let G be a graph and T a corresponding skeleton forest. It holds that $N[u] = N[v]$ if and only if u and v are on the same simple path.*

Proof. If $N[u] = N[v]$, then u and v are on the same simple path:

Assume otherwise, i.e., that u and v are not on the same simple path in T. Consider the path P_{uv} between u and v in T. As it is not simple, it contains a node that is not its lowest node and has at least a child x that is not on P_{uv}. As x is not on P_{uv}, either $\{u,x\} \in E$ and $\{v,x\} \notin E$ or vice-versa, depending on whether u is an ancestor of v or v an ancestor of u. This is a contradiction to $N[u] = N[v]$, thus u and v must be on the same simple path.

If u and v are on the same simple path, $N[u] = N[v]$:

An edge $\{u, v\}$ exists if and only if u and v are in an ancestor-descendant relationship in the skeleton T. Consider a node u. All ancestors/descendants of u apart from its simple path are also ancestors/descendants of all other nodes in its simple path. Further, the nodes in its simple path form a clique. Therefore, $N[u] = N[v]$. □

Lemma 2. *Let T, T' be two different skeletons that induce the same quasi-threshold graph G. Then the simple paths of u in T and T' consist of the same nodes.*

Proof. Assume otherwise, i.e., that the simple paths of u in T and T' differ. Then there is a node x that is on the simple path of u in T but not in T' (or vice-versa, but assume w.l.o.g. that it is in T). As x and u are on the same simple path in T, $N[u] = N[x]$ by Lemma 1. Lemma 1 also implies that u and x must be on the same simple path in T', which is a contradiction to the existence of x and thus our assumption. Thus, the simple paths of u must consist of the same nodes in T and T'. □

Lemma 3. *Let T, T' be two skeletons that induce the same quasi-threshold graph G. Then the only difference between T and T' is the reordering of simple paths.*

Proof. Assume otherwise, i.e., that there were two skeletons T, T' that imply the same quasi-threshold graph G but differ more than just reordering of simple paths. A forest is uniquely determined by specifying the set of ancestors of every node. Thus there must be a node u such that the ancestors of u in T are different from the ancestors in T'. As a consequence, there is a node v that is an ancestor of u in T or T', but not in both. Assume w.l.o.g. that v is an ancestor of u in T. Due to T, $\{u, v\} \in E$. As $\{u, v\} \in E$ if and only if v is an ancestor of u or v is a descendant of u, v must be a descendant of u in T'. As u is an ancestor of v in T, $N[u] \supseteq N[v]$. As v is an ancestor of u in T', $N[v] \supseteq N[u]$ and thus $N[u] = N[v]$. Due to Lemma 1, this implies that u and v are together on a simple path in both T and T'.

By Lemma 2, the simple path of u must consist of the same nodes in T and T'. Therefore, we can replace the simple path in T by the simple path in T' without altering the resulting graph, and then search for a new pair u, v as described above. This reordering of the simple path does not change any other simple path. Therefore, if we apply this procedure repeatedly, it cannot find the same nodes again. Thus, this procedure terminates after at most n steps. As every step just reorders a simple path, the only difference between T and T' was reordering of simple paths. □

The main idea of path sorting is, before moving a node u, to move all its neighbors to the top of their respective simple paths. Since it might unify simple paths and thus enable reordering, u is first removed from the graph. This reordering makes it possible to choose the lowest neighbor of a simple path as parent without needing to insert edges to other non-neighbors in it. Note that the order in simple paths does not play a role when adopting a node c as a child because all nodes in its path become neighbors of u anyway. We show that this minimizes the number of edits incident to u by considering an optimal set of edits and its skeleton forest and showing that our forest with reordered simple paths does not yield more edits.

Lemma 4. *Consider a graph $G = (V, E)$, a node $u \in V$ and a skeleton forest T. Applying QTM to u on T^- which is T with u removed and simple paths reordered such that neighbors of u are at the top of their simple paths minimizes the number of edits incident to u.*

Proof. Let Q be the quasi-threshold graph with minimum edits incident to u and T_Q a skeleton forest of Q. Let T_Q^- be T_Q without u, children of u attached to u's parent. This keeps all ancestor-descendant-relationships between all nodes except u and thus all remaining edges. The reverse of this operation is exactly what QTM does: choosing a parent and potentially adopting some of its children. Thus, QTM can find an optimal set of edits incident to u in T_Q^-. Since, by Lemma 3, T^- and T_Q^- differ only in the order of nodes on simple paths, we show that the orderings of T^- and T_Q^- are equally good.

Consider the parent p and children C of u in T_Q. If p is the lower end of its simple path in T_Q^-, we obtain the same ancestor from the lowest node of p's simple path in T^-. Similarly, for an adopted child $c \in C$, adoption of the highest node of c's simple path in T^- yields the same descendants. If p is not the lower end of its simple path in T_Q^-, we distinguish two cases: u adopted p's only child or u is a leaf node in T_Q. In the first case, neither the position in p's simple path nor its node order matters as any position and node order gives the same neighbors and thus edits. If u is a leaf node in T_Q and p is not the lower end of its simple path, the node order matters as u is only connected to p and p's ancestors but not the nodes below p on p's simple path P_p in T_Q^-. By Lemma 3, P_p also exists in T_Q^-. Every non-neighbor of u among p and its ancestors in P_p causes an edge insertion while every neighbor of u below p causes an edge deletion. By moving all neighbors of u to the top of P_p and choosing the lowest neighbor of u on P_p as parent, we do not get any edits incident to nodes of P_p and thus minimize the edits among all possible orderings of P_p. This shows that QTM finds a parent and children to adopt on T^- that minimize the number of edits incident to u. □

What remains to show is that maintaining and sorting all simple paths does not increase the asymptotic running time of QTM. Simple paths are maintained explicitly in a dynamic array, every node stores its simple path and position in it. This allows us to swap neighbors of the node to move u in constant time to the position of the first non-neighbor in its simple path, we also store this position. Moving nodes can cause simple paths to be split or joined. We store simple paths ordered from lowest to highest node. Whenever simple paths are split or merged, u is adjacent in the edited graph to the upper part of the path either before or after the move. In a split, we remove the upper part of the path from its end. In a merge, we add the nodes of the upper path to the lower path. Both operations are thus linear in the number of neighbors of u before or after the move. The running time analysis of QTM already accounts for running time linear in the number of neighbors of u in the edited graph both before and after the move. Thus, path sorting does not increase the asymptotic running time of QTM.

3.3 Inclusion-Minimal Editing

With the local moving routine of QTM, we can incrementally insert the nodes of a graph G into an initially empty graph. Due to Lemma 4, this minimizes the number of edits in each step. Overall, this yields an inclusion-minimal editing of G, as it has also been shown, e.g., for interval graphs [26]. The basic idea is that if there was a set of superfluous edits, these edits could have been omitted already at the steps where they were introduced, violating the local minimality guaranteed by Lemma 4.

This inclusion-minimal editing algorithm can also be considered a one-pass streaming algorithm. To add a node, we need the skeleton of the already seen nodes, which can be stored in $\mathcal{O}(\log(n))$ bits per node. We only consider the edges of every node once, the only constraint is that when we encounter a node u in the stream, we also need to get all incident edges that are incident to the already seen nodes.

3.4 Randomized Choices

To accelerate convergence, the original QTM algorithm moves a node u only if this reduces the number of edits and there is no rule for breaking ties between moves. The algorithm also never adopts children whose subtrees contain an equal number of neighbors and non-neighbors, as this only swaps edge deletions for insertions. We call such children *indifferent children*. We now propose to break ties by choosing uniformly at random from the best options for u, even if this does not lead to an improvement. The rationale is that on a plateau of equally good solutions only some may lead to better solutions in the next move. The same technique can also be applied to the inclusion-minimal editing, where a more diverse set of solutions can be obtained.

This poses two challenges: we need to find all options, and we may count each of them only once. In particular, different choices of a parent p and children C to adopt might actually yield the same quasi-threshold graph and thus only one of them should be considered. For instance, choosing a parent x without adopting any children is the same as choosing x's parent p as parent and adopting x. But we also cannot disregard p, because adopting a second child of p would yield a different quasi-threshold graph.

Since, according to Lemma 2, the set of simple paths is unique, we can resolve the ambiguity by ensuring that a node u that is moved is inserted at the bottom of its new simple path. The lowest node of a simple path does not have exactly one child, for otherwise the path would not end there. Accordingly, we ignore positions where u adopts exactly one child.

Thus, if a potential parent p has exactly one close child and no indifferent children, we disregard it. If p has one close child, we must choose at least one indifferent child and thus get $2^{c_i} - 1$ possibilities to choose among the c_i indifferent children. If p has at least two close children, we can choose an arbitrary subset of the indifferent children and get 2^{c_i} possibilities. If p has no close children and at most one indifferent child, we have only the single option of not adopting the child. If p has no close children and $c_i \geq 2$ indifferent children, we have $2^{c_i} - c_i$ possible choices among the indifferent children that do not lead to exactly one child.

In our algorithm, we propagate the number of choices upwards in our bottom-up search together with the minimum number of required edits and the best parent. When processing a node that is a suitable parent that achieves the same number of edits, we choose it with a probability that is proportional to its number of choices for adopting children divided by the total number of choices aggregated so far. As the number of choices is exponential in the number of indifferent children, we store the logarithm of the number of choices to avoid overflows or dealing with huge integers. While this introduces rounding errors when adding numbers that are of different orders of magnitude, in these cases the chances of choosing one parent instead of the other are vanishingly small anyway.

QTM already guarantees that we discover nodes whose subtree contains as many neighbors as non-neighbors, so it is easy to select them. There are some cases though, where QTM needs to be modified to propagate information about equally good parents. In particular, this is the case if the current candidate so far shows no benefit over isolating the node to move. In that case, QTM does not propagate any information as there is always an ancestor of the current node that is at least as good. We adapt QTM to also propagate information about equally good parents even if the number of saved edits is 0. We however do not insert the parent p into the priority queue unless there is an actual improvement over isolating the node to move. The reason for this is that if p is a non-neighbor, it causes an additional edit that leads to -1 saved edits. This cannot be compensated further up in the tree, as otherwise the path above p to the root contained more u-neighbors than non-neighbors and choosing the parent of p as parent of u and not adopting any children was better.

4 Experimental Evaluation

We added our extensions to the original QTM implementation in C++ as part of NetworKit [31 SPP][2]. All experiments were performed on an Intel Core i7-2600K CPU with 32GB RAM. Each algorithms was executed ten times with ten different seeds and randomly permuted node ids. By *instance*, we denote a combination of seed and (permuted) graph.

[2] Our implementation is available at https://github.com/michitux/networkit/tree/upstream/qtm-linear.

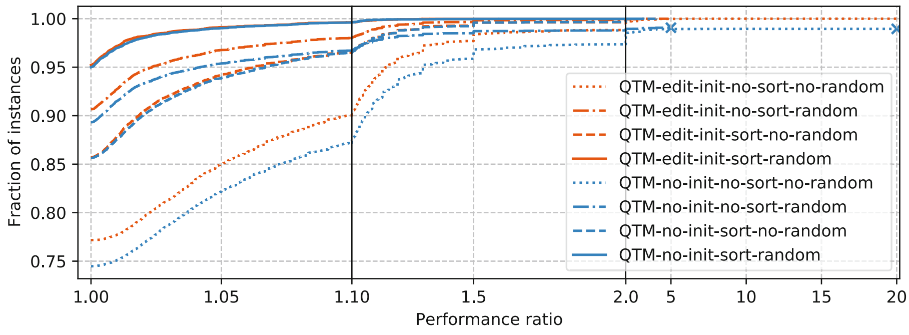


Fig. 2. Comparison of the different variants of QTM on the COQ protein similarity dataset with either no initialization or the initialization heuristic. Lines ending with a "x" are algorithms that need edits for instances that are quasi-threshold graphs and are thus infinitely worse than the best algorithm.

Our algorithms are evaluated on two datasets. The first consists of 3964 connected components of the COG protein similarity data [2, 27]. Each connected component consists of a symmetric matrix of similarities, and we construct an unweighted graph from its non-negative entries. Even though the dataset does not include fully connected components (i.e., cliques), 1666 components remain that are quasi-threshold graphs and do not require any edits. We restrict parts of our analysis to the 716 graphs that require at least 20 edits. As a second dataset we use 100 social networks of Facebook friendships at US universities and colleges [32].

Unless noted otherwise, QTM is run for a maximum of 400 iterations. We stop early if an iteration does not result in a node movement. With randomization enabled, however, we do continue for up to 50 iterations without improvements if nodes had more than one option.

We use so-called *performance profiles* [12] to compare the number of edits achieved by different algorithms. A performance profile indicates the fraction of instances on which an algorithm performed within a specified percentage of the best algorithm with the best seed on that graph. For readability, we sometimes divide the plots by vertical lines indicating intervals of the x-axis with different linear scales.

4.1 Sorting Paths and Randomization

We first examine the impact of sorting paths and randomization on the number of edits. In a $2 \times 2 \times 2$-design, we combine no initialization (a spanning forest of isolated nodes) and the previous initialization heuristic [4 SPP] with iterations that make or do not make use of path sorting and randomization.

Figure 2 shows the results for the full COG protein similarity dataset. Despite the many instances that are, or are almost, quasi-threshold graphs, clear differences arise, with the old variants performing the worst. As is to be expected, quasi-threshold graphs are not always recognized without initialization. The variants with just sorting follow with some margin. Here, the difference between the two initialization algorithms is

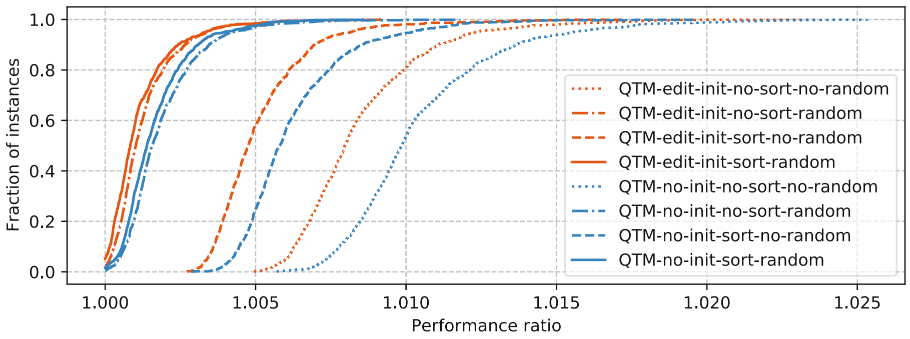


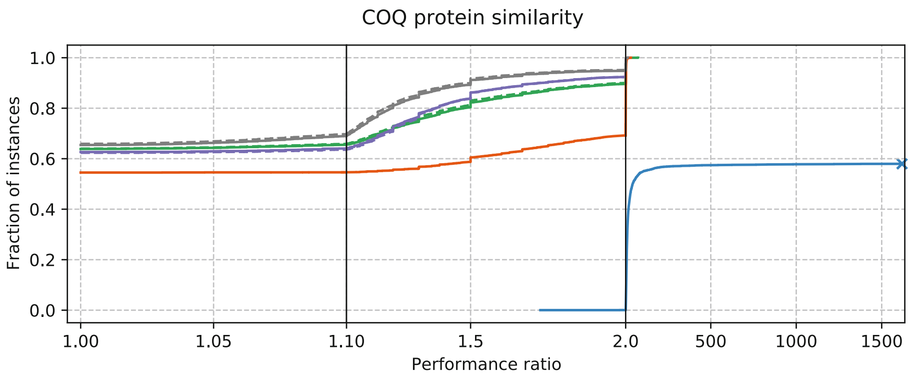
Fig. 3. Comparison of the different variants of QTM on the Facebook 100 dataset with either no initialization or the initialization heuristic.

almost gone and no quasi-threshold graph requires any edits. This is not the case right after the first iteration, though, so we can rule out that one iteration of this algorithm is an alternative inclusion-minimal algorithm.

The versions with just randomization performs even better than with just sorting paths. However, here, some graphs are not recognized as quasi-threshold graphs and a clear gap between the two initializations remains. With path sorting and randomization, the performance is even better, regardless of the initialization 95% of the instances are as good as the best algorithm and seed, and almost all instances are within 10% of the best solution.

For the Facebook 100 dataset, the results that are shown in Fig. 3 are slightly different. First, the instances are much more challenging with even the smallest requiring more than ten thousand edits. There are slight differences between the different solutions which mean that usually there is just one seed and algorithm that achieves the best result on a graph, explaining why no algorithm has the best solution for more than 10% of the instances. Also, we are no longer talking about 10% differences in the number of edits, but at most 2.5%. Still, there are clear differences between the algorithm variants. The original two variants need at least 0.5% more edits than the best solutions on almost all instances while the variants with path sorting and randomization need at most 0.5% more edits than the best solutions on almost all instances. The variants with path sorting give a good improvement, but unlike in the COG protein similarity dataset, the differences between the initializations remain. With randomization, the difference between the two initializations is even larger than the difference between using just randomization and using both path sorting and randomization.

Overall, we can conclude that using path sorting and randomization significantly improve the quality of the solutions. However, on the Facebook 100 dataset, initialization still seems to make a difference, indicating that even with these improvements we are not able to escape all local minima. Next, we consider the inclusion-minimal editing as initialization.



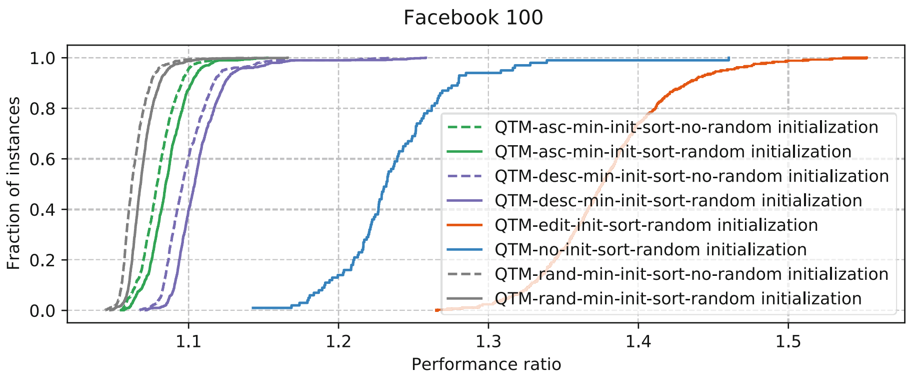


Fig. 4. Comparison of the different initializations of QTM on the COQ protein similarity dataset (top) and the Facebook 100 dataset (bottom).

4.2 Initialization and Convergence

Apart from the two original initialization methods, we consider three variants of the inclusion-minimal editing that differ based on the order in which nodes are inserted. We consider a random order and descending or ascending by degree. For the inclusion-minimal initialization, we also consider randomization of the chosen position in the skeleton.

First, we consider just the initialization itself in Fig. 4 for both datasets. Both plots use as "best algorithm" the algorithm runs with the up to 400 iterations. For the COQ protein similarity dataset, we can see that even just the initialization algorithms also match some of the best solutions, which is to be expected as some of them require no edits. No initialization corresponds to just deleting all edges, and we can see that for some graphs this is very far from an optimal solution. The inclusion-minimal variants clearly need less edits than the initialization heuristic, with the randomized order being best and a not so clear distinction between ascending and descending order. Interestingly, the variants without additional randomization seem to perform slightly better.

On the Facebook 100 dataset, a large fraction of the edges is edited, such that even just deleting all edges is less than 50% worse than what the best algorithm achieves.

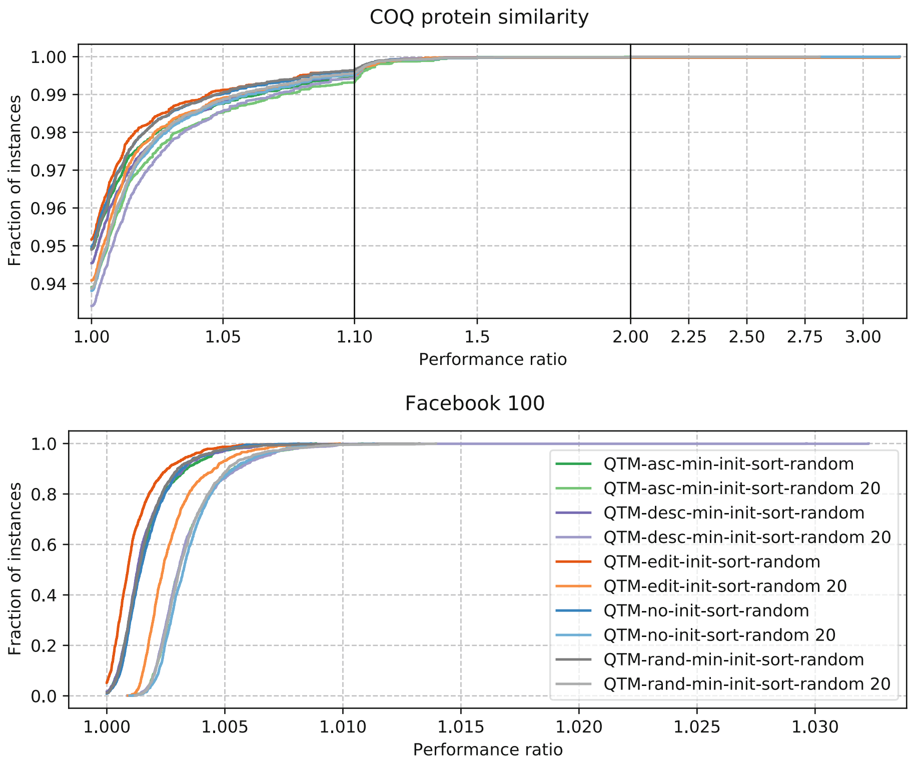


Fig. 5. Comparison of the different initialization variants of QTM on the COQ protein similarity dataset (top) and the Facebook 100 dataset (bottom) with both path sorting and randomization after up to 400 iterations and after 20 iterations.

The initialization heuristic actually needs more edits than just deleting all edges, an observation already made by Brandes et al. [4 SPP]. The inclusion-minimal initialization algorithms perform much better than that, even though they do not match any best results. Again, the randomized order is best, following by ascending and then descending degree order. We can also clearly see again that not randomizing the choices is slightly better. This indicates that there might be potential for further optimizing choices in the inclusion-minimal editing algorithm.

Next, we consider how the choice of the initialization algorithm influences the result after 20 iterations or up to 400 iterations with both path sorting and randomization enabled. Figure 5 compares the results for both datasets. For the COQ protein similarity dataset, the results are very close. The initialization heuristic wins both after 20 and 400 iterations, the inclusion-minimal editing with randomized order comes second. The remaining variants follow, with the descending degree ordering being last. The differences are small, though, and in some cases the initialization seems to be more important than the number of iterations.

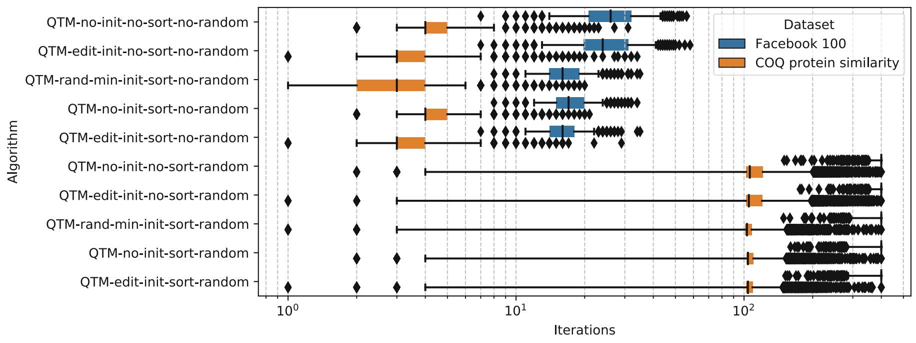


Fig. 6. Number of iterations used by QTM. The COQ protein similarity dataset only includes graphs that require at least 20 edits. Whiskers extend to the 5th and 95th percentile.

For the Facebook 100 dataset, the difference between 20 and 400 iterations is clearly visible. While the initialization heuristic clearly wins, the number of iterations seems more important than the initialization. This can be explained by the much larger and more difficult graphs that also require more iterations as shown in Fig. 6. Here, we exclude the ascending and descending degree ordered initialization to improve readability.

Without randomization, most instances of the COQ protein similarity dataset converge within 10 iterations. On the Facebook 100 dataset, those algorithms require up to 40 iterations for most instances to converge. Enabling path sorting decreases the number of required iterations. As the initialization is not counted as an iteration, it is natural that variants without initialization take an iteration longer in the median on the COQ protein similarity dataset. The difference between the initialization heuristic and the inclusion-minimal editing as initialization is small. With randomization enabled, most instances of the Facebook 100 dataset use all 400 iterations that we allowed. With path sorting enabled, some more instances converge earlier, i.e., either no move was possible – which is unlikely here – or no improvement has been found for 50 iterations. For the COQ protein similarity dataset, most instances finish in a bit more than 100 iterations. Again, this is less with path sorting.

We conclude that the initialization heuristic introduced by Brandes et al. [4 SPP] is still unmatched in results even though it is initially worse than the new inclusion-minimal variants. For the inclusion-minimal editing, a random node order seems to perform best. Path sorting leads not only to better results of QTM, but also faster convergence. Randomization leads to a much larger number of iterations that yield some improvements. Here, limiting the number of iterations is required to achieve reasonable running times but still even with 20 iterations, randomization improves results.

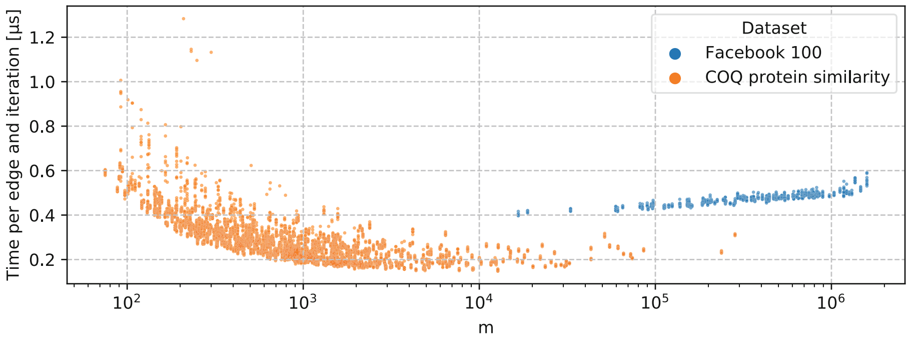


Fig. 7. Running time per edge and iteration vs. number of edges of QTM with initialization heuristic, path sorting and randomization on graphs of the two datasets requiring at least 20 edits.

4.3 Running Time

Figure 7 shows the running time per edge and iteration in microseconds for QTM with initialization heuristic, randomization and path sorting. Although this includes the time for initialization, we normalized by the number of subsequent iterations. Since initialization time is dominated by the iterations, which in turn are linear in the number of edges, this normalized running time should be roughly constant. For the COQ protein similarity dataset, it actually decreases with increasing graph size. Given that this happens in the range where these graphs have only hundreds of edges, initialization overheads might play a role. For the Facebook 100 dataset, running times actually increase slightly with graph size. Between the smallest and the largest graph, we see an increase from around 0.4µs to 0.6µs. We examined CPU statistics and found increased cache misses to be a likely explanation. The percentage of cache misses increases while the number of instructions per edge and iteration is almost constant across the Facebook 100 dataset.

5 Conclusion

We have extended the fast quasi-threshold editing heuristic QTM with new path sorting and randomization components. We have shown that path sorting both provides new local optimality guarantees in theory and better results in practice. Our experimental results indicate that randomization indeed helps escaping local optima, but convergence needs much longer, in particular for large graphs. Still, even with few iterations, results are improved in practice. We also modified QTM into a linear-time algorithm for inclusion-minimal edit sets, which serve well as initialization for QTM. While it reduces the number of edits compared to the previous initialization heuristic, the final result after convergence are slightly worse.

Therefore, it would be interesting to investigate further ways to escape local minima, e.g., by moving several nodes at once by some form of contraction. A recent master's thesis [29] extends QTM to the weighted quasi-threshold editing problem where every node pair has a cost and the goal is to find a set of edits with minimum total cost. It shows that with non-uniform edit costs, QTM seems to get stuck in local minima and investigates moving whole subtrees as a remedy. While moving subtrees helps, it also significantly increases the running time.

Acknowledgement. This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grants BR 2158/11-2 and WA 654/22-2 within the Priority Programme 1736 *Algorithms for Big Data*.

References

1. Bansal, N., Blum, A., Chawla, S.: Correlation clustering. Mach. Learn. **56**(1–3), 89–113 (2004). https://doi.org/10.1023/B:MACH.0000033116.57574.95
2. Böcker, S., Briesemeister, S., Bui, Q.B.A., Truß, A.: A fixed-parameter approach for weighted cluster editing. In: APBC, pp. 211–220. Imperial College Press (2008). http://www.comp.nus.edu.sg/%7Ewongls/psZ/apbc2008/apbc050a.pdf
3. Borgatti, S.P., Everett, M.G.: Models of core/periphery structures. Soc. Netw. **21**(4), 375–395 (2000). https://doi.org/10.1016/S0378-8733(99)00019-2

4 SPP. Brandes, U., Hamann, M., Strasser, B., Wagner, D.: Fast quasi-threshold editing. In: Bansal, N., Finocchi, I. (eds.) ESA 2015. LNCS, vol. 9294, pp. 251–262. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48350-3_22

5. Brandes, U., Holm, E., Karrenbauer, A.: Cliques in regular graphs and the core-periphery problem in social networks. In: Chan, T.-H.H., Li, M., Wang, L. (eds.) COCOA 2016. LNCS, vol. 10043, pp. 175–186. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48749-6_13
6. Bruckner, S., Hüffner, F., Komusiewicz, C.: A graph modification approach for finding core-periphery structures in protein interaction networks. Algorithms Mol. Biol. **10**, 16 (2015). https://doi.org/10.1186/s13015-015-0043-7
7. Cai, L.: Fixed-parameter tractability of graph modification problems for hereditary properties. Inf. Process. Lett. **58**(4), 171–176 (1996). https://doi.org/10.1016/0020-0190(96)00050-6
8. Chu, F.P.M.: A simple linear time certifying LBFS-based algorithm for recognizing trivially perfect graphs and their complements. Inf. Process. Lett. **107**(1), 7–12 (2008). https://doi.org/10.1016/j.ipl.2007.12.009
9. Corneil, D.G., Lerchs, H., Burlingham, L.S.: Complement reducible graphs. Discret. Appl. Math. **3**(3), 163–174 (1981). https://doi.org/10.1016/0166-218X(81)90013-5
10. Creignou, N., Ktari, R., Meier, A., Müller, J., Olive, F., Vollmer, H.: Parameterised enumeration for modification problems. Algorithms **12**(9), 189 (2019). https://doi.org/10.3390/a12090189
11. Crespelle, C.: Linear-time minimal cograph editing (2019). https://perso.ens-lyon.fr/christophe.crespelle/publications/SUB_minimal-cograph-editing.pdf
12. Dolan, E.D., Moré, J.J.: Benchmarking optimization software with performance profiles. Math. Program. **91**(2), 201–213 (2002). https://doi.org/10.1007/s101070100263
13. van Dongen, S.M.: Graph Clustering by Flow Simulation. Ph.D. thesis, University of Utrecht (2000)
14. Drange, P.G., Dregi, M.S., Lokshtanov, D., Sullivan, B.D.: On the threshold of intractability. In: Bansal, N., Finocchi, I. (eds.) ESA 2015. LNCS, vol. 9294, pp. 411–423. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48350-3_35
15. Drange, P.G., Pilipczuk, M.: A polynomial kernel for trivially perfect editing. Algorithmica **80**(12), 3481–3524 (2017). https://doi.org/10.1007/s00453-017-0401-6

16. Foldes, S., Hammer, P.L.: The Dilworth number of a graph. Ann. Discrete Math. **2**, 211–219 (1978). https://doi.org/10.1016/S0167-5060(08)70334-0
17. Fortunato, S.: Community detection in graphs. Phys. Rep. **486**(3–5), 75–174 (2010). https://doi.org/10.1016/j.physrep.2009.11.002
18 SPP. Gottesbüren, L., Hamann, M., Schoch, P., Strasser, B., Wagner, D., Zühlsdorf, S.: Engineering exact quasi-threshold editing. In: SEA, pp. 10:1–10:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.SEA.2020.10
19. Hammer, P.L., Simeone, B.: The splittance of a graph. Combinatorica **1**(3), 275–284 (1981). https://doi.org/10.1007/BF02579333
20. Liu, Y., Wang, J., Guo, J., Chen, J.: Complexity and parameterized algorithms for cograph editing. Theor. Comput. Sci. **461**, 45–54 (2012). https://doi.org/10.1016/j.tcs.2011.11.040
21. Liu, Y., Wang, J., You, J., Chen, J., Cao, Y.: Edge deletion problems: branching facilitated by modular decomposition. Theor. Comput. Sci. **573**, 63–70 (2015). https://doi.org/10.1016/j.tcs.2015.01.049
22. Luce, R.D., Perry, A.: A method of matrix analysis of group structure. Psychometrika **14**, 95–116 (1949). https://doi.org/10.1007/BF02289146
23. Mahadev, N.V., Peled, U.N.: Threshold Graphs and Related Topics. Ann. Discrete Math. **56**. Elsevier (1995)
24. Mariani, M.S., Ren, Z.M., Bascompte, J., Tessone, C.J.: Nestedness in complex networks: observation, emergence, and implications. Phys. Rep. **813**, 1–90 (2019). https://doi.org/10.1016/j.physrep.2019.04.001
25. Nastos, J., Gao, Y.: Familial groups in social networks. Soc. Netw. **35**(3), 439–450 (2013). https://doi.org/10.1016/j.socnet.2013.05.001
26. Ohtsuki, T., Mori, H., Kashiwabara, T., Fujisawa, T.: On minimal augmentation of a graph to obtain an interval graph. J. Comput. Syst. Sci. **22**(1), 60–97 (1981). https://doi.org/10.1016/0022-0000(81)90022-2
27. Rahmann, S., Wittkop, T., Baumbach, J., Martin, M., Truß, A., Böcker, S.: Exact and Heuristic Algorithms for Weighted Cluster Editing. In: CSB, pp. 391–401 (2007). https://doi.org/10.1142/9781860948732_0040
28. Schaeffer, S.E.: Graph clustering. Comput. Sci. Rev. **1**(1), 27–64 (2007). https://doi.org/10.1016/j.cosrev.2007.05.001
29. Schmitt, D.: Engineering Heuristic Quasi-Threshold Editing. Master's thesis, Karlsruhe Institute of Technology (2021). https://i11www.iti.kit.edu/_media/teaching/theses/ma-schmitt-21.pdf
30. Schoch, D., Brandes, U.: Re-conceptualizing centrality in social networks. Eur. J. Appl. Math. **27**(6), 971–985 (2016). https://doi.org/10.1017/S0956792516000401
31 SPP. Staudt, C.L., Sazonovs, A., Meyerhenke, H.: Networkit: a tool suite for large-scale complex network analysis. Netw. Sci. **4**(4), 508–530 (2016). https://doi.org/10.1017/nws.2016.20
32. Traud, A.L., Mucha, P.J., Porter, M.A.: Social structure of Facebook networks. Phys. A: Stat. Mech. Appl. **391**(16), 4165–4180 (2012). https://doi.org/10.1016/j.physa.2011.12.021
33. Wolk, E.S.: A note on "The comparability graph of a tree". Proc. AMS **16**(1), 17–20 (1965). https://doi.org/10.2307/2033992
34. Yan, J., Chen, J., Chang, G.J.: Quasi-threshold graphs. Discret. Appl. Math. **69**(3), 247–255 (1996). https://doi.org/10.1016/0166-218X(96)00094-7

The Space Complexity of Undirected Graph Exploration

Yann Disser[1(✉)] and Max Klimm[2]

[1] TU Darmstadt, Darmstadt, Germany
disser@mathematik.tu-darmstadt.de
[2] TU Berlin, Berlin, Germany
klimm@tu-berlin.de

Abstract. We review the space complexity of deterministically exploring undirected graphs. We assume that vertices are indistinguishable and that edges have a locally unique color that guides the traversal of a space-constrained agent. The graph is considered to be explored once the agent has visited all vertices. We visit results for this setting showing that $\Theta(\log n)$ bits of memory are necessary and sufficient for an agent to explore all n-vertex graphs. We then illustrate that, if agents only have sublogarithmic memory, the number of (distinguishable) agents needed for collaborative exploration is $\Theta(\log\log n)$.

Keywords: Graph exploration · Multi-agent exploration · Space complexity · Connectivity · Log-space

1 Introduction

When working with large data sets it is no longer justified to assume the entire input, or even a significant fraction of it, to be accessible at once. In particular, data may be spatially distributed along a dynamic network structure, such as the Internet or social networks. In this setting, the systematic navigation or crawling of the network becomes an integral component of any algorithmic processing of the data it holds. The theoretical framework of graph exploration is concerned precisely with the algorithmic problem of systematically traversing an initially unknown graph.

Generally, the main questions in graph exploration are regarding *feasibility*, i.e.,how much computational power is necessary for systematic exploration, and regarding *efficiency*, i.e.,how quickly a graph can be explored algorithmically. In the context of dealing with large data sets, the feasibility question is of particular importance. The necessary computational power can be captured theoretically by the space complexity of the exploration problem. Intuitively, the question is what portion of a graph we need to be able to memorize in order to avoid running in circles.

In this chapter, we review the most important results regarding the space complexity of undirected graph exploration. In Sect. 2, we introduce the graph exploration framework in more detail. In Sect. 3, we outline a general lower bound on the space complexity of graph exploration of $\Omega(\log n)$. Reingold's algorithm for undirected graph exploration is presented in Sect. 4. We then turn to collaborative graph exploration by a set of agents. In Sect. 5, we show that when all agents have sub-logarithmic memory $\mathcal{O}(\log^{1-\varepsilon} n)$ for some $\varepsilon > 0$, then $\Omega(\log\log n)$ agents are needed to explore any undirected graph with n vertices. Finally, in Sect. 6, we provide a matching upper bound showing that a team of $\mathcal{O}(\log\log n)$ agents can explore deterministically any undirected n-vertex graph, even if each agent has only constant memory.

The aim of this chapter is to survey the key ideas of these results, and we only sketch proofs on a high level. Whenever possible, intuition is prefered over formal statements, and many details are omitted to increase accessibility. For a more formal treatment, we refer to the original papers. Pointers to the relevant literature are given in Sect. 7.

2 Exploration and Feasibility

In the following, we consider an agent initially located at a vertex v_0 of an unknown, edge-colored, undirected graph $G = (V, E)$. We assume the edge-coloring to be locally unique in the sense that no two edges incident to a common vertex may share a color. The agent's perception of G is limited to observing the set of colors of the edges incident to its current location. In every step, the agent may choose one of these colors and move to the other endpoint of the corresponding edge. Importantly, vertices with the same set of colors adjacent to them are indistinguishable to the agent. The objective of the agent is to explore G, i.e.,to systematically visit all vertices of G in a finite number of steps. We are looking for a *deterministic* traversal algorithm that guarantees to explore every undirected graph. Regarding *randomized* traversal algorithms, it is known that a random walk of length $n^5 \log n$ visits all vertices of any graph with n vertices with high probability (Aleliunas et al. [1]). This yields a constant-space *perpetual* randomized graph exploration algorithm, i.e., an algorithm that runs forever and eventually visits all vertices. If n is known, combining this algorithm with a counter counting up to $n^5 \log n$ yields a log-space randomized graph exploration algorithm.

To illustrate the difficulty of deterministic exploration in this weak agent model, consider the exploration of a *fully regular* graph G, i.e.,a graph where all vertices are incident to edges of the exact same set of colors (cf. Fig. 1). Even if the agent knows that G is fully regular, after the first step where it learns the degree of the graph, its observations contain no information at all. In particular, every deterministic exploration algorithm must produce the same sequence of colors for any two fully regular graphs using the same colors. Intuitively, this is the most challenging setting for exploration. Then, the algorithmic problem reduces to asking for a *universal traversal sequence*, i.e.,a sequence of colors that we can follow to eventually visit all vertices, irrespective of G and v_0. Here and throughout, following a color sequence means to perform a sequence of movement decisions according to it, and we say that a color sequence explores G if the agent visits all vertices when following it.

The exploration problem is feasible in the sense that a universal traversal sequence always exists for fully regular graphs. To see this, follow any path in an edge-colored graph and then return to the starting location by backtracking along the same path to get a color sequence that is a palindrome. Conversely, following a color sequence that is a palindrome guarantees to yield a closed tour, irrespective of the graph and the starting location. This means that we can obtain a universal traversal sequence by chaining

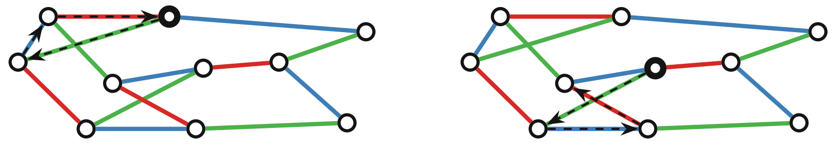


Fig. 1. A regular graph with two different starting locations. By following the color sequence “green, blue, red”, the agent either moves on a cycle (left) or not (right), but there is no way to distinguish between these two cases as vertices are indistinguishable. (Color figure online)

together all color sequences that are palindromes in order of increasing lengths. The resulting sequence is guaranteed to follow every path from the starting location, irrespective of the graph, and thus to eventually visit all vertices.

For non-regular graphs, a universal traversal sequence seems unattainable since not every color needs to be available at every vertex. However, the exploration of an arbitrary non-regular graph $G = (V,E)$ can be reduced to the exploration of a fully regular graph $G_{\text{freg}} = (V_{\text{freg}}, E_{\text{freg}})$ via the construction shown in Fig. 2. To this end, we first construct a regular graph $G_{\text{reg}} = (V_{\text{reg}}, E_{\text{reg}})$ with bi-colored edges. For every vertex $v \in V$ and each color c of its adjacent edges, we introduce a color copy $(v,c) \in V_{\text{reg}}$, connect the color copies of v in a cycle and add the original edges between the respective color copies. The resulting graph has only three colors. The edges in the cycles are bi-colored with one color pointing to the next color copy, and one color pointing to the previous color copy. Edges between color copies of different vertices have a third color. We proceed to eliminate the bi-colored edges in G_{reg} and obtain a fully regular graph G_{freg}. This can be done by first adding an intermediate vertex for each bi-colored edge, and then mirroring (i.e.,copying) the entire construction and connecting each vertex of degree 2 with its reflection with the third color.

As explained above, there is a universal traversal sequence for 3-regular graphs and, thus, the sequence also explores G_{freg}. Given a universal traversal sequence for G_{freg}, we can explore G with an additional memory overhead that is logarithmic in the maximum degree of the original graph and, thus, in $\mathcal{O}(\log n)$. The idea is to perform a virtual traversal of G_{freg} and only actually move in G whenever the virtual traversal transitions between color copies of different vertices of G. The memory is used to store which color copy of its location in G the agent is (virtually) located at in G_{freg}, as well as whether it is at a vertex or its reflection and whether it is located on the intermediate vertex of a bi-colored edge.

While we have now established the general feasibility of the exploration problem, the constructed exploration algorithm is not very satisfactory in the sense that it enumerates an exponential number of sequences before all vertices are guaranteed to have been visited. This means that the algorithm requires an exponential number of moves and a linear memory size to keep track of its current state. Note that as long as the color sequence remains aperiodic, linear memory is needed to perform an exponential number of steps and, conversely, making use of a linear number of memory bits means visiting an exponential number of memory states and thus an exponential running time. In that sense, there is a direct correspondence between exponential time and linear memory. From now on, we focus on memory usage only. The natural question in this context becomes: Can we solve the exploration problem in sub-linear memory?

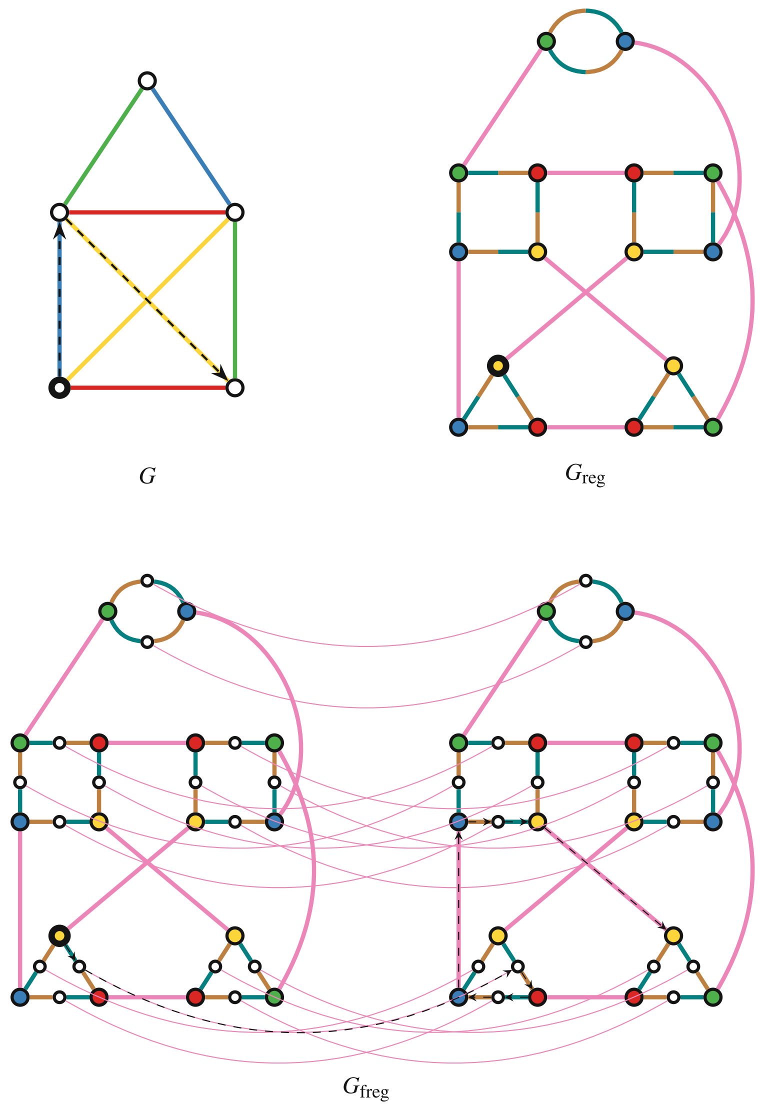


Fig. 2. Turning an arbitrary graph G into a regular graph G_{reg} with bi-colored edges and further into a fully regular graph G_{freg}. In the construction, we order the four colors of G cyclically as yellow-red-green-blue. In G_{reg}, brown edges point to the next color available at the corresponding vertex in G, teal edges point to the previous color, and purple edges move to a color copy of another vertex. To construct G_{freg} from G_{reg}, an intermediate vertex is added to the center of each bi-colored edge, the graph is copied, and two corresponding intermediate vertices are connected by a purple edge. Starting with the color yellow on the left copy in G_{freg}, the color sequence "teal-purple-brown-teal-brown-purple-brown-teal-purple" for G_{freg} leads to the movement along a blue edge and a yellow edge as indicated in G. (Color figure online)

3 Trapping a Single Agent

To approach the question of how much memory is necessary in general to deterministically explore a graph G of size n, we first need to realize how insufficient memory can manifest itself in terms of the inability of the agent to explore: Essentially, the only way that the agent may fail to explore G in finite time is by getting "trapped" in periodic behavior that forces it to move on a closed tour eternally, without having visited all vertices. With this in mind, we make the following definition.

Definition 1. *A* trap *for an exploration algorithm is given by an edge-colored graph G together with an initial location v_0, such that the algorithm never visits all vertices of G when starting at v_0.*

We fix a deterministic exploration algorithm $\mathscr{A}$ with a finite number $b \in \mathbb{N}$ of memory bits and construct a trap of some size n for this algorithm. The size of our trap then bounds the largest size of graphs that the algorithm can explore. Conversely, since the construction can be carried out for any deterministic algorithm, we obtain a lower bound on the required number of memory bits necessary to explore graphs of size (up to) n.

To construct a trap G for $\mathscr{A}$, first observe that $\mathscr{A}$ has at most 2^b different memory states at its disposal. Our construction ensures that G is a fully regular graph of degree 3, using a fixed set of three colors C. As observed in the previous section, $\mathscr{A}$ is sure to yield the same sequence S of colors for all fully regular graphs using colors C and irrespective of the initial location v_0. Since $\mathscr{A}$ has at most 2^b different states, it must enter at least one state for the second time within the first 2^b steps. Assume the same state is entered in steps $1 \leq i < j \leq 2^b$. Then the behavior of $\mathscr{A}$ and, consequently, S must become periodic after step i, i.e.,$S = (c_1, \dots, c_{i-1}) \oplus S_\mathrm{p}^\infty$, where '$\oplus$' denotes concatenation of sequences, and S_p is a finite subsequence of S of length $j - i$.

Consider the infinite walk $W = (v_0, v_1, v_2, \dots)$ induced by S in the infinite 3-regular tree where the set of colors of the edges incident to each vertex is C; cf. Fig. 3 (top). By definition, $\mathscr{A}$ is in the same memory state after steps i and j, implying that it follows the same infinite color sequence starting at v_i in steps $i+1, i+2, \dots$ as it does starting at v_j in steps $j+1, j+2, \dots$. Assume that $v_i = v_j$. Then, the algorithm moves on a closed tour of length $j - i$ after step i while having visited at most $i + j - i = j \leq 2^b$ different vertices. We can now take the subgraph G of the infinite tree induced by all edges incident to vertices in W as our trap. Note that this graph need not be fully regular, but we can add missing edges by mirroring G as before (cf. Sect. 2) and connecting corresponding vertex pairs of degree smaller three by an edge of a color they are missing. This decreases the number of missing colors at all vertices of degree smaller three and needs to be repeated at most once to make the graph fully regular.

In the case $v_i \neq v_j$ the algorithm may visit an infinite number of different vertices. The intuitive idea now is to "close a loop" by ensuring that both the edges of color $c_{i+1} = c_{j+1}$ at v_i and at v_j lead to the same vertex. Of course, we cannot simply replace the edge of color c_{j+1} at v_j by the edge $\{v_j, v_{i+1}\}$ of the same color, since we also need to keep the edge $\{v_i, v_{i+1}\}$ of this color. However, we can achieve the same result by "folding" v_i onto v_j, i.e.,by identifying $v_i = v_j$ and identifying the predecessors of v_i along W and their neighborhoods accordingly. More precisely, we identify each vertex

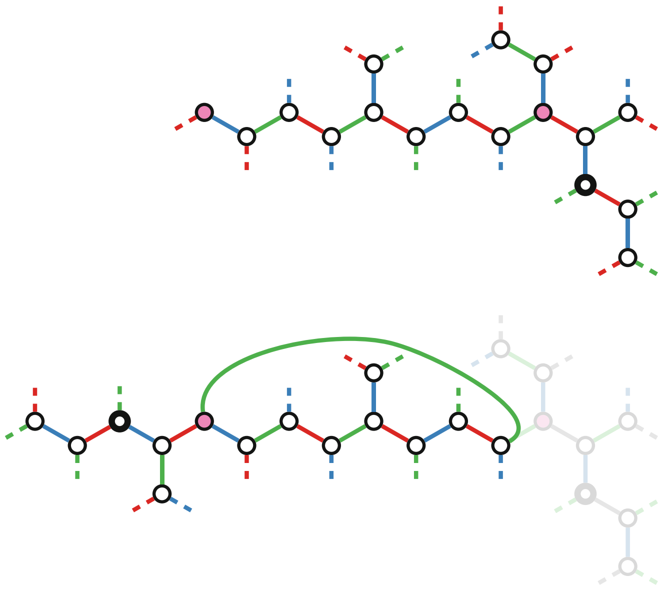


Fig. 3. Construction of a trap for a single agent with b bits of memory. Top: After at most 2^b steps in a fully regular graph, the same memory state must repeat (purple vertices). Bottom: Closing a loop to trap the agent on a closed walk. (Color figure online)

v adjacent to v_i with the unique vertex v' adjacent to v_j such that the colors of the edges $\{v_i, v\}$ and $\{v_j, v'\}$ coincide. We repeat this process for all vertices $v_{i-1}, \dots, v_0$ along W; cf Fig. 3 (bottom). Afterwards, we again take the subgraph induced by $\{v_0, \dots, v_j\}$ together with their neighbors as our trap, making it fully regular as before.

In either case, we have constructed a trap of size $n = \mathcal{O}(2^b)$. Since we can perform this construction for any deterministic algorithm with b memory bits, this implies a lower bound of $\Omega(\log n)$ on the required number of memory bits to explore every graph of size up to $n \in \mathbb{N}$. We have shown the following.

Theorem 1 (Fraigniaud et al. [12]). *The number of memory bits needed for undirected, deterministic graph exploration is $\Omega(\log n)$.*

4 Reingold's Algorithm

We will see that the lower bound shown in Sect. 3 on the memory needed to explore an undirected graph deterministically is tight, i.e.,undirected graphs with n vertices can be explored deterministically with $\mathcal{O}(\log n)$ memory. This algorithmic result follows from a famous result of Reingold [16] in which he established that USTCON $\in \mathscr{L}$. Here, $\mathscr{L}$ is the class of problems solvable with logarithmic memory and USTCON is the problem of deciding, for a given undirected graph $G = (V, E)$ and two designated vertices $s, t \in V$, whether s and t are connected in G. The algorithm devised by Reingold for his proof can be turned into a log-space exploration algorithm, which we outline in the following.

We first argue that fully regular graphs with constant degree and good vertex expansion can be explored with logarithmic memory. Suppose the graph G is fully regular with constant degree d and enjoys the property that there is a constant $\varepsilon > 0$ such that for all vertex sets $S \subset V$ with $|S| \leq n/2$ there are at least $(1+\varepsilon)|S|$ vertices that are connected by an edge to a vertex in S. An upshot of this vertex expansion property is that the graph has at most logarithmic diameter. Indeed, for an arbitrary vertex $u \in V$ there are more than $n/2$ vertices within a distance of $k = \frac{\log(n/2)}{\log(1+\varepsilon)} + 1$ of u, so that every pair of vertices has a common vertex within distance k and, thus, the diameter is at most $2k \in \mathcal{O}(\log n)$. Similar to the argument in Sect. 2, it suffices to enumerate all returning color sequences of length $2k$ which can be done with $\mathcal{O}(\log n)$ space.

Regularity can be achieved with the transformation from G to G_{reg} explained in Sect. 2. Here, we stick to G_{reg} with its bi-colored edges instead of transforming G_{reg} further into G_{freg} since the bi-colored edges of G_{reg} do not harm our further arguments. We proceed to describe further transformations that turn G_{reg} into another regular graph G_{exp} with good vertex expansion. Let G be a fully d-regular graph with n vertices and let H be a c-regular graph with d vertices where c and d are constants. Then the *replacement product* $G \textcircled{r} H$ is the graph where each vertex v in G is replaced by a copy of H that we call the *cloud* of v. The edges within a cloud keep the colors that they have in H. For each edge of G, we introduce an edge with a new inter-cloud color between the respective vertices in the corresponding clouds; cf. Fig. 4. The resulting graph $G \textcircled{r} H$ is fully regular with degree $c+1$. Based on the replacement product $G \textcircled{r} H$, we introduce another graph product, the *zig-zag product* $G \textcircled{z} H$. The zig-zag-product $G \textcircled{z} H$ has the same set of vertices as the replacement product $G \textcircled{r} H$, but only edges between vertex clouds of different vertices. Specifically, let (u,i) be a vertex belonging to the cloud of u, and (v,j) be a vertex belonging to the cloud of v. Then, the edge $\{(u,i),(v,j)\}$ is contained in the replacement product if and only if there is path of length three from (u,i) to (v,j) in $G \textcircled{r} H$ where the middle edge is an edge between different clouds. For a vertex (u,i) there are exactly c^2 such paths starting in (u,i): the first degree of freedom is to choose one of c colors (within the current cloud), then change the cloud with an inter-cloud edge, and then choose one of c colors for the second cloud. Associating each of these c^2 color combinations with a new color in $G \textcircled{z} H$, we obtain that $G \textcircled{z} H$ is fully regular with degree c^2. We note that this construction also works if G has bi-colored edges by allowing inter-cloud edges also between different copies of vertices of H. In any case, we may end up with a graph $G \textcircled{z} H$ having bi-colored edges. Suppose that H is of constant size and that we have a traversal sequence for $G \textcircled{z} H$. Then, every edge traversal in $G \textcircled{z} H$ corresponds to three edge traversals in $G \textcircled{r} H$. We maintain a stack of future edge traversals in $G \textcircled{r} H$. Since H has constant degree, so has $G \textcircled{r} H$, and we can store this stack with up to three colors in constant memory. In this way, we obtain a traversal sequence for $G \textcircled{r} H$ with constant memory overhead. From a traversal sequence for $G \textcircled{r} H$, we further obtain a traversal sequence for G by memorizing the current copy of the vertex of H within the current cloud similar to the virtual traversal of G_{freg} in Sect. 2. As H has constant size, this requires only constant memory overhead. We conclude that a traversal sequence for $G \textcircled{z} H$ can be used to traverse G with constant memory overhead.

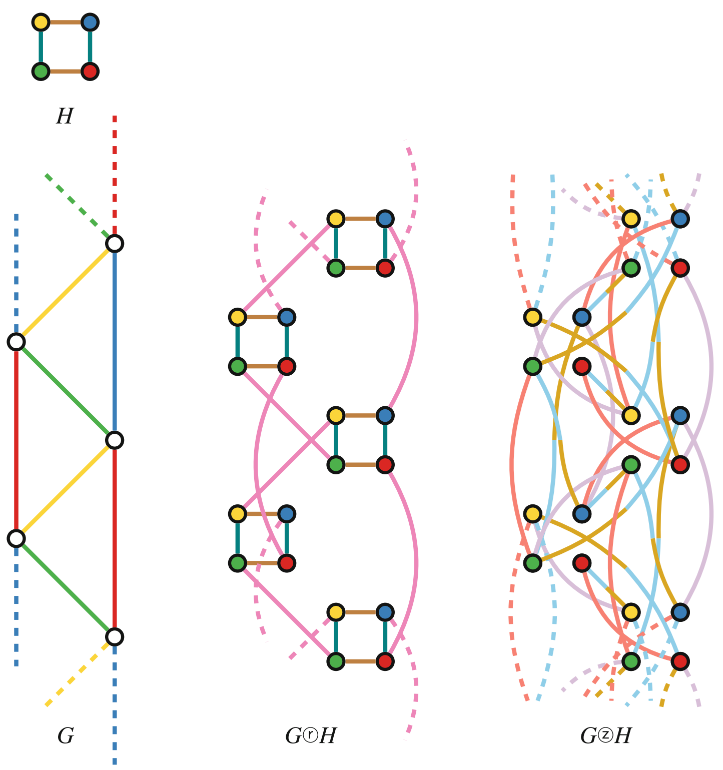


Fig. 4. The replacement product $G \circledr H$ and the zig-zag-product $G \circledz H$ for two graphs G and H. In the replacement product $G \circledr H$, edges within a cloud keep the colors they had in H, here brown or teal. The edges between clouds get a new inter-cloud color, here purple. Every edge in $G \circledz H$ corresponds to a path of length three in $G \circledr H$ where the middle edge is purple, e.g.,an edge color gold in $G \circledz H$ corresponds to a path in $G \circledr H$ that is teal-pink-brown. (Color figure online)

It is left to show that we can transform G_{reg} into a graph with good vertex expansion. In order to show that a d-regular graph has good vertex expansion, it suffices to show that the second largest eigenvalue λ of the normalized adjacency matrix is bounded from above by a constant strictly smaller than 1; cf. Tanner [21], Alon and Milnan [3], and Alon [2]. For the normalized adjacency matrix $M = (m_{u,v})_{u,v \in V}$, the entry $m_{u,v}$ is defined as $1/d$ times the number of edges from u to v. For ease of notation, we call a d-regular graph on n vertices an (n,d,α)-graph if $\lambda \leq \alpha$. We use the following properties of the second largest eigenvalues of regular graphs:

1. Alon and Sudakov [5]:
   A d-regular, connected, non-bipartite n-vertex graph is a $\left(n,d,1-\frac{1}{dn^2}\right)$-graph.
2. Basic linear algebra:
   Taking the k-th power of a graph means introducing an edge for each k-edge path in the original graph. If G is an (n,d,λ)-graph, then its k-th power is an (n,d^k,λ^k)-graph.

3. Alon and Roichman [4] (cf.discussion in Reingold et al. [17, § 5]):
   There exists a $(c^{16}, c, 1/2)$-graph for some constant c.
4. Reingold et al. [17]:
   Let G be an (n,d,λ)-graph and let H be a $(d,c,1/2)$-graph. Then $G \textcircled{z} H$ is an $\big(nd, c^2, \frac{1}{8}(3\lambda + \sqrt{9\lambda^2+16})\big)$-graph.

Let H be a $(c^{16}, c, 1/2)$-graph with c constant as in Property 3.. For an arbitrary graph G on n vertices, first construct G_{reg}. Let G_0 be equal to G_{reg} except that $c^{16}-3$ self loops are added to each vertex. Let $\ell = 2\lceil \log(c^{16}n^4)\rceil$. For $i = 1,\dots,\ell$, define $G_i = (G_{i-1} \textcircled{z} H)^8$, i.e.,to obtain the next graph in the sequence, we first apply the zig-zag product with H and then take the 8-th power of the resulting graph. Note that this is well-defined since $G_{i-1} \textcircled{z} H$ has degree c^2, so that $G_i = (G_{i-1} \textcircled{z} H)^8$ has degree c^{16}, and $G_i \textcircled{z} H$ is defined. Any traversal sequence for G_i can be transformed with constant memory overhead to a traversal sequence for G_{i-1}, since it involves taking the zig-zag product with a graph of constant size and power 8 (which requires to memorize up to 7 additional steps). Thus, a traversal sequence for G_ℓ can be transformed to a traversal sequence for G_0 and, hence, an exploration sequence for G with memory overhead of $\mathcal{O}(\ell) = \mathcal{O}(\log n)$. It remains to show that G_ℓ has good vertex expansion. We claim that $\lambda(G_i) \le \max\{\lambda(G_{i-1})^2, 1/2\}$ for all $i = 1,\dots,\ell$. To prove the claim, let $\lambda = \lambda(G_{i-1})$ and note that, by Property 4.,

$$\lambda(G_{i-1} \textcircled{z} H) \le \frac{1}{8}\Big(3\lambda + \sqrt{9\lambda^2+16}\Big) \le \frac{1}{8}\Big(3\lambda + 5\Big) = 1 - \frac{3}{8}\Big(1-\lambda\Big) < 1 - \frac{1}{3}\Big(1-\lambda\Big),$$

implying $\lambda(G_i) < \big(1-\frac{1}{3}(1-\lambda)\big)^8$ by Property 2. If $\lambda < \frac{1}{2}$, then $\lambda(G_i) < \left(\frac{5}{6}\right)^8 < \frac{1}{2}$. Otherwise, it is straightforward to verify that the function $f(x) = (1-\frac{1}{3}(1-x))^4$ is convex on $[0,1]$ and $1 \ge f(1)$ as well as $1/2 \ge f(1/2)$. We conclude that $f(x) \le x$ for all $x \in [1/2, 1]$, in particular

$$\left(1 - \frac{1}{3}\Big(1-\lambda\Big)\right)^4 \le \lambda,$$

implying $\lambda(G_i) \le \lambda^2$. Finally, the graph G_0 is regular with degree c^{16} and has at most n^2 nodes. By Property 1. this implies that $\lambda(G_0) \le 1 - \frac{1}{c^{16}n^4}$. With the claim above, we obtain

$$\lambda(G_\ell) \le \max\left\{\left(1-\frac{1}{c^{16}n^4}\right)^{2^\ell}, \frac{1}{2}\right\} \le \left\{\left(\left(1-\frac{1}{c^{16}n^4}\right)^{c^{16}n^4}\right)^2, \frac{1}{2}\right\} \le \left\{\left(1-\frac{1}{e}\right)^2, \frac{1}{2}\right\} = \frac{1}{2}.$$

As we sketched above, the transformation from G to G_ℓ requires only logarithmic memory and can be conducted locally, i.e.,a traversal sequence for G_ℓ can be transformed into an exploration sequence for G with logarithmic space overhead. Finally, we eliminate the bi-colored edges of G_ℓ as in Sect. 2. Since this construction increases the diameter of the graph by at most a factor of 2, it has still a logarithmic diameter, so that a traversal sequence can be constructed with logarithmic space. This yields the following result.

Theorem 2 (Reingold [16]). *Undirected graphs can be deterministically explored with an agent that has $\mathcal{O}(\log n)$ bits of memory.*

5 Trapping Multiple Agents

After having established that $\Theta(\log n)$ memory bits are necessary and sufficient for deterministic exploration with a single agent, we now investigate whether this bound can substantially be lowered by allowing additional agents. More precisely, we consider a setting with $k \geq 2$ deterministic and distinguishable agents that behave as before individually, but move in a synchronized fashion and may exchange information while co-located at a vertex. To see that allowing collaboration makes a fundamental difference, even for $k = 2$, observe that, for example, two agents can distinguish closed tours simply by leaving one of them at the starting location; cf. Fig. 1. This additional power is also evidenced by a drastically increased difficulty of constructing traps: For a long time, the smallest known traps for k agents, with s memory states each, had a size of $\mathcal{O}\left(s^{s^{\cdot^{\cdot^{s}}}}\right)$ with $\mathcal{O}(k)$ levels in the exponent (Fraigniaud et al. [13], Rollik [18]), compared to the singly exponential bound of Theorem 1.

To see that a substantially different approach is needed to trap multiple agents, recall the construction in Sect. 3: The intuitive idea was to add vertices along a tree until the agent enters a memory state for the second time, at which point we close a loop. Since the number of memory states available to the agent is bounded by a constant, namely 2^b, this yielded a trap of singly exponential size in b. The key difference when allowing multiple agents is that the behavior of the agents no longer only depends on their collective memory state. It now might make a difference in the behavior of the algorithm at what points agents meet – which is exactly the reason, why they can distinguish cycles, as explained above. This means that the behavior of the algorithm may depend in a non-trivial way on the positions of the agents in the graph, relative to each other. As we increase the number of vertices n, the number of such configurations grows as n^k, and we can no longer hope for configurations to ever repeat.

The key idea to overcome this is to force the agents to stay "close" to each other, which ensures that the number of configurations stays bounded and allows us to use the same general approach as before. The following informal definition generalizes the notion of a trap to multiple agents.

Definition 2. *A k-*barrier *B_k in a graph G for an algorithm $\mathscr{A}$ is a subgraph of G whose removal disconnects the graph into two connected components, with the property that no agent ever traverses B_k from one component to the other without at least k other agents entering B_k during the traversal.*

In particular, a 1-barrier plays the role of a simultaneous trap for every individual agent. Note that agents may behave differently from one another, so we need to deal with each

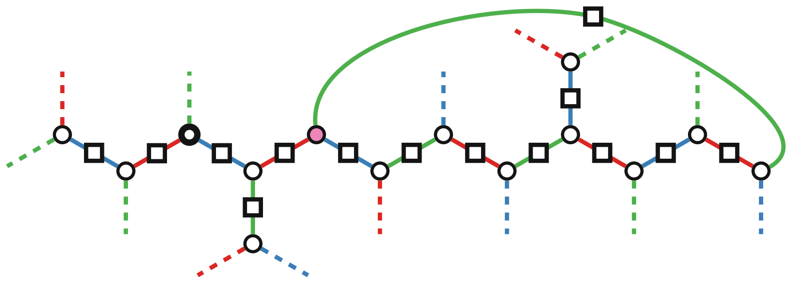


Fig. 5. Sketch of the construction of an i-barrier. Boxes indicate $(i-1)$-barriers.

one using a separate construction. We have seen in Sect. 3 how to construct a trap for a single agent, and we can essentially chain traps together for the individual agents in order to obtain a 1-barrier. We will now describe how to recursively construct i-barriers for $i \in \{2, \dots, k\}$. Once we have constructed a k-barrier, we have the desired trap for the set of all k-agents.

The idea of the recursive construction of an i-barrier is to use the same approach as in the trap for a single agent, but replacing every edge by an $(i-1)$-barrier; cf. Fig. 5. More precisely, we fix any set of i agents and assume that only these agents enter our construction. Since, on a meta-level, edges are now $(i-1)$-barriers, the agents can only traverse these "meta-edges" if they all enter the corresponding barrier, i.e.,if they stay somewhat close together. Essentially, throughout the traversal, all agents are guaranteed to be located in one of the three $(i-1)$-barriers surrounding some meta-vertex. Of course, the same is true recursively within every $(i-1)$-barrier containing at most $i-1$ of the agents. By a careful recursive inspection, the total number of configurations of the agents can be bounded independently of the number of meta vertices. This allows a similar approach as before: Add meta-vertices until a configuration repeats and close a loop to obtain a trap. To obtain an i-barrier, we again need to chain traps for every subset of i agents together.

With some refinement and a thorough analysis, it can be shown that this yields a k-barrier, and thus a trap, of size $\mathcal{O}(s^{2^{5k}})$ for k agents with s memory states each. In other words, the agents can explore graphs of size up to $n \leq s^{2^{5k}}$, i.e.,$\log n \leq 2^{5k} \cdot \log s$ has to hold. Assuming that each agent has $\mathcal{O}(\log^{1-\varepsilon} n)$ bits of memory for some $\varepsilon \in (0,1)$, i.e.,just shy of the number needed to explore the graph on its own, we obtain $\log s = \mathcal{O}(\log^{1-\varepsilon} n)$. Combining both bounds and taking logarithms yields $k = \Omega\big(\log\big(\frac{\log n}{\log^{1-\varepsilon} n}\big)\big) = \Omega(\log\log n)$. This means that we need at least $k = \Omega(\log\log n)$ agents to explore undirected graphs of size n, even if every agent has almost enough memory to explore on its own!

Theorem 3. (Disser et al. [10 SPP]). *Deterministic exploration of undirected graphs needs at least $\Omega(\log\log n)$ agents if we allow $\mathcal{O}(\log^{1-\varepsilon} n)$ bits of memory for every agent, where $\varepsilon > 0$.*

6 Multi-agent Exploration

We outline the design of a collaborative exploration algorithm that matches the lower bound of Theorem 3, i.e.,we show that $\mathcal{O}(\log\log n)$ agents with sub-logarithmic memory are sufficient to explore unknown graphs of size up to n. Observe that $\mathcal{O}(\log n)$ agents are trivially sufficient by Reingold's algorithm (Theorem 2), since we can let agents move together and make each one responsible for maintaining a constant number of memory bits.

We start with a single agent with a constant number $m_0 \in \mathbb{N}$ of memory bits and show how to iteratively boost its memory by using a small number of additional agents. First consider how much progress, in terms of visiting vertices, the agent is able to accomplish on its own. For a single agent, we already know Reingold's algorithm which needs logarithmic space. Expressed differently, executing Reingold's algorithm with m_0 bits of available memory guarantees that the agent visits a number of distinct vertices of order $\Omega(2^{m_0})$, or completes the exploration.

These vertices can be visited multiple times, and, in general, there is no way of knowing the order in which the vertices appear during the traversal T produced by Reingold's algorithm. However, using one additional agent to mark vertices and multiple repetitions of traversal with Reingold's algorithms for different positions of the additional agent, it can be shown that we can treat T as a simple cycle without self intersections. Assuming that the agent has this cycle T of length $\Omega(2^{m_0})$, for some constant $c \in \mathbb{N}$, that it can navigate systematically, it can position a constant number $a \in \mathbb{N}$ of additional agents along T. Since agents are distinguishable, there are $|T|^a$ configurations that can be established in this way. The key idea now is to use the configuration of the agents along T as a form of virtual memory state, in order to boost the amount of memory available to the agent.

The number of memory bits that can be encoded in this way is $m_1 = \log|T|^a$, which is of order am_0. This means that we have boosted the initial memory capacity roughly by a factor of a. Having more (virtual) memory at its disposal, the agent can now recursively repeat the procedure, again boosting the memory by another factor of a, and so on. After $\log\log n$ levels of recursion, the amount of virtual memory is of order $a^{\log\log n} \cdot m_0 = \Omega(\log n)$. But we already know that this is sufficient to complete the exploration, by Theroem 2.

For this approach to yield the claimed bound, it is crucial to argue that only a constant number of agents and memory bits are needed in each recursive level, not only to encode, but also to manipulate the virtual memory. In particular, in each move performed in some level of the recursion, the agents encoding the virtual memory on lower recursive levels need to be moved in the graph to stay in the same positions relative to the agent. It can be shown that this is indeed possible with a constant overhead in agents, and we obtain the following tight result.

Theorem 4. (Disser et al. [10 SPP]). *Undirected graphs can be deterministically explored with $\mathcal{O}(\log\log n)$ agents, even if we only allow constant memory for every agent.*

7 Bibliographic Notes

The first exploration algorithms were designed for mazes. A *maze* is a subgraph of the two-dimensional grid where the vertices are indistinguishable and each edge is labeled with its cardinal direction. To facilitative the exploration, the agent is sometimes equipped with a set of distinguishable pebbles that can be dropped and retrieved at nodes. After initial non-tight results (Blum and Sakoda [7], Budach [8], Shah [20]), it has been proven that an agent with finite memory needs two pebbles to explore any maze (Blum and Kozen [6]) and that one pebble does not suffice (Hoffmann [14]). Blum and Kozen [6] further showed that also two agents with finite memory can explore all mazes.

General undirected graphs are harder to explore. The lower bound of $\Theta(\log n)$ on the memory needed by a single agent to explore all undirected vertex graphs deterministically given in Sect. 3 is due to Fraigniaud et al. [12]. Aleliunas et al. [1] showed that a random walk of length $n^5 \log n$ explores an undirected n-vertex graph with high probability. The deterministic algorithm exploring undirected vertex graphs explained in Sect. 4 is due to Reingold [16]. We here follow the presentation of the algorithm and the analysis of Reingold's original paper. There are also alternative proofs for this result that avoid the use of the zig-zag-product; see Rozenman and Vadhan [19]. Reingold's algorithm constructs a universal exploration sequence. This concept was introduced by Koucky [15].

Regarding the exploration of a graph by a set of cooperating agents, Blum and Kozen [6] showed that three agents with finite memory cannot explore all finite undirected planar graphs. Rollik [18] strengthened this result showing that for any number $k \in \mathbb{N}$ of agents with $s \in \mathbb{N}$ states, there is a *trap* of size $\mathcal{O}\left(s^{s^{\cdot^{\cdot}}}\right)$ with $2k+1$ levels in the exponent, i.e.,a graph that the agents cannot explore. Fraigniaud et al. [13] improved this bound to $k+1$ levels in the exponent. The non-planar trap of size $\mathcal{O}\left(s^{2^{5k}}\right)$ given in Sect. 5 is due to Disser et al. [10 SPP]. This result implies that if each agent has a sublogarithmic memory of $\mathcal{O}(\log^{1-\varepsilon} n)$ with $\varepsilon > 0$, then $\mathcal{O}(\log\log n)$ agents are needed to explore all undirected n-vertex graphs. Another consequence from their construction is that a single agent with sublogarithmic memory needs $\mathcal{O}(\log\log n)$ pebbles to explore all undirected n-vertex graphs. The result that $\mathcal{O}(\log\log n)$ agents with constant memory can explore all undirected n-vertex graphs presented Sect. 6 is due to Disser et al. [10 SPP]. They actually showed that a single agent with constant memory and $\mathcal{O}(\log\log n)$ pebbles can explore the graph and provide a general reduction from agents to pebbles. They further proved that their algorithm runs in polynomial time. For results regarding the exploration time needed by an agent with unconstrained memory, see Dudek et al. [11] and Chalopin et al. [9].

References

1. Aleliunas, R., Karp, R.M., Lipton, R.J., Lovász, L., Rackoff, C.: Random walks, universal traversal sequences, and the complexity of maze problems. In: FOCS, pp. 218–223. IEEE Computer Society (1979). https://doi.org/10.1109/SFCS.1979.34
2. Alon, N.: Eigenvalues and expanders. Combinatorica **6**(2), 83–96 (1986). https://doi.org/10.1007/BF02579166
3. Alon, N., Milman, V.D.: λ_1, isoperimetric inequalities for graphs, and superconcentrators. J. Comb. Theory, Ser. B **38**(1), 73–88 (1985). https://doi.org/10.1016/0095-8956(85)90092-9

4. Alon, N., Roichman, Y.: Random Cayley graphs and expanders. Random Struct. Algorithms **5**(2), 271–285 (1994). https://doi.org/10.1002/rsa.3240050203
5. Alon, N., Sudakov, B.: Bipartite subgraphs and the smallest eigenvalue. Comb. Probab. Comput. **9**(1), 1–12 (2000). https://doi.org/10.1017/S0963548399004071
6. Blum, M., Kozen, D.: On the power of the compass (or, why mazes are easier to search than graphs). In: FOCS, pp. 132–142. IEEE Computer Society (1978). https://doi.org/10.1109/SFCS.1978.30
7. Blum, M., Sakoda, W.J.: On the capability of finite automata in 2 and 3 dimensional space. In: FOCS, pp. 147–161. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.20
8. Budach, L.: Automata and labyrinths. Math. Nachrichten **86**, 195–282 (1978). https://doi.org/10.1002/mana.19780860120
9. Chalopin, J., Das, S., Kosowski, A.: Constructing a map of an anonymous graph: applications of universal sequences. In: Lu, C., Masuzawa, T., Mosbah, M. (eds.) OPODIS 2010. LNCS, vol. 6490, pp. 119–134. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17653-1_10

10 SPP. Disser, Y., Hackfeld, J., Klimm, M.: Tight bounds for undirected graph exploration with pebbles and multiple agents. J. ACM **66**(6), 40:1–40:41 (2019). https://doi.org/10.1145/3356883

11. Dudek, G., Jenkin, M., Milios, E.E., Wilkes, D.: Robotic exploration as graph construction. IEEE Trans. Robot. Autom. **7**(6), 859–865 (1991). https://doi.org/10.1109/70.105395
12. Fraigniaud, P., Ilcinkas, D., Peer, G., Pelc, A., Peleg, D.: Graph exploration by a finite automaton. Theor. Comput. Sci. **345**(2–3), 331–344 (2005). https://doi.org/10.1016/j.tcs.2005.07.014
13. Fraigniaud, P., Ilcinkas, D., Rajsbaum, S., Tixeuil, S.: The reduced automata technique for graph exploration space lower bounds. In: Goldreich, O., Rosenberg, A.L., Selman, A.L. (eds.) Theoretical Computer Science. LNCS, vol. 3895, pp. 1–26. Springer, Heidelberg (2006). https://doi.org/10.1007/11685654_1
14. Hoffmann, F.: One pebble does not suffice to search plane labyrinths. In: Gécseg, F. (ed.) FCT 1981. LNCS, vol. 117, pp. 433–444. Springer, Heidelberg (1981). https://doi.org/10.1007/3-540-10854-8_47
15. Koucký, M.: Universal traversal sequences with backtracking. J. Comput. Syst. Sci. **65**(4), 717–726 (2002). https://doi.org/10.1016/S0022-0000(02)00023-5
16. Reingold, O.: Undirected connectivity in log-space. J. ACM **55**(4), 17:1–17:24 (2008). https://doi.org/10.1145/1391289.1391291
17. Reingold, O., Vadhan, S., Wigderson, A.: Entropy waves, the zig-zag graph product, and new constant-degree expanders. Ann. Math. **155**(1), 157–187 (2002). https://doi.org/10.2307/3062153
18. Rollik, H.: Automaten in planaren graphen. Acta Informatica **13**, 287–298 (1980). https://doi.org/10.1007/BF00288647
19. Rozenman, E., Vadhan, S.: Derandomized squaring of graphs. In: Chekuri, C., Jansen, K., Rolim, J.D.P., Trevisan, L. (eds.) APPROX/RANDOM -2005. LNCS, vol. 3624, pp. 436–447. Springer, Heidelberg (2005). https://doi.org/10.1007/11538462_37
20. Shah, A.N.: Pebble automata on arrays. Comput. Graph. Image Process. **3**(3), 236–246 (1974). https://doi.org/10.1016/0146-664X(74)90017-3
21. Tanner, R.M.: Explicit concentrators from generalized n-gons. SIAM J. Alg. Disc. Meth. **5**(3), 287–293 (1984). https://doi.org/10.1137/0605030

Algorithms for Big Data and Their Applications

Scalable Cryptography

Dennis Hofheinz[1(✉)] and Eike Kiltz[2]

[1] ETH Zürich, Zürich, Switzerland
hofheinz@inf.ethz.ch
[2] Ruhr-Universität Bochum, Bochum, Germany
eike.kiltz@rub.de

Abstract. In our modern digital society, cryptography is vital to protect the secrecy and integrity of transmitted and stored information. Settings like digital commerce, electronic banking, or simply private email communication already rely on encryption and signature schemes.

However, today's cryptographic schemes do not scale well, and thus are not suited for the increasingly large sets of data they are used on. For instance, the security guarantees currently known for RSA encryption—one of the most commonly used type of public-key encryption scheme—degrade linearly in the number of users and ciphertexts. Hence, larger settings (such as cloud computing, or simply the scenario of encrypting all existing email traffic) may enable new and more efficient attacks. To maintain a reasonable level of security in larger scenarios, RSA keylengths must be chosen significantly larger, and the scheme becomes very inefficient. Besides, a switch in RSA keylengths requires an update of the whole public key infrastructure, an impossibility in truly large scenarios. Even worse, when the scenario grows beyond an initially anticipated size, we may lose all security guarantees.

This problematic is the motivation for our project "Scalable Cryptography", which aims at offering a toolbox of cryptographic schemes that are suitable for huge sets of data. In this overview, we summarize the approach, and the main findings of our project. We give a number of settings in which it is possible to indeed provide scalable cryptographic building blocks. For instance, we survey our work on the construction of scalable public-key encryption schemes (a central cryptographic building block that helps secure communication), but also briefly mention other settings such as "reconfigurable cryptography". We also provide first results on scalable *quantum-resistant* cryptography, i.e., scalable cryptographic schemes that remain secure even in the presence of a quantum computer.

Keywords: Public-key cryptography · Security reductions

1 Introduction and Motivation

Motivation: Public-Key Cryptography... Public-key cryptography, introduced by Diffie and Hellman [13] in 1976, is at the heart of modern cryptography. A public-key encryption (PKE) scheme can be used to transmit messages securely by encrypting them. The main feature that distinguishes PKE schemes from earlier encryption schemes (and in particular from symmetric encryption schemes such as AES) is the existence of two separate keys: the encryption (or, public) key is used to encrypt messages, while the decryption (or, secret) key is used to decrypt ciphertexts.

Among the first suggested PKE schemes were the RSA scheme of Rivest, Shamir, and Adleman [35], and the scheme of Merkle and Hellman [31]. Later on, many more followed, e.g., [6,9,12,15,20,32]. Today, PKE schemes are crucially used to protect large-scale systems. For instance, PKE schemes secure Internet browsers [37] (including e-banking applications such as HBCI, the home banking computer interface standard), Internet auctions [10], or simply email [39]. We stress that such applications cannot be solved with more classical methods of encryption (like symmetric encryption) alone. However, symmetric encryption schemes like AES do play a role in making such applications more efficient.

It has become a standard requirement that a cryptographic scheme (and in particular a PKE scheme) should come with provable security guarantees. Indeed, the *in*security of a cryptographic scheme can have catastrophic consequences (think of an electronic voting scheme), and is usually not immediately detectable. Hence, security cannot be achieved using a trial-and-error method, and should be argued beforehand.

The dangers of a missing security proof are best demonstrated by the PKCS Internet browser encryption standard [36,37]. This de facto standard defines how browsers should encrypt their communication when accessing sensitive websites, e.g., for e-banking, or e-commerce. An older version of that standard [36] used a PKE scheme *without* security proof, and was subsequently broken by Bleichenbacher [8]. This caused massive media attention, and made expensive updates necessary. As a result, the updated standard [37] relies upon a variant of the RSA PKE scheme *with* (heuristic) security proof.

We stress that a security proof always refers to a formal security model which covers the possible attacks in practice. Goldwasser and Micali [20] gave the first formal security notion, and proved a simple (but comparatively inefficient) PKE scheme secure in this sense. Later on, more efficient provably secure PKE schemes were devised (e.g., [9,12]), and the considered security notions were refined (e.g., [14,32,33,38]).

... in a Big Data Scenario. Now consider the following simple but realistic example scenario. Namely, imagine that every owner of a smartphone encrypts all of his/her Internet communication (using a state-of-the art PKE scheme). Such an encryption already takes place for selected Internet connections, and usually for communication with email servers. However, for this example, we will assume that all communication is encrypted. This leads to a large-scale setting in which both the number of users and the number of encryptions is in the (large) millions. For simplicity, let us assume that there are $n_U = 2^{30}$ users, each performing $n_C = 2^{30}$ (i.e., about one billion) encryptions.[1]

We would like to derive provable security guarantees for the used encryption in this setting. This means that we would like to have a formal statement that the only way to break *any instance* of the used encryption scheme is to solve a (preferably well-

[1] Of course, many practical settings may actually involve fewer users or encryptions. To derive meaningful universal security guarantees, however, we are assuming what seems plausible in *some* realistic applications (like browser encryption or messaging apps).

understood) mathematical problem. Unfortunately, most existing PKE schemes do not scale well in this setting. For instance, the best known security guarantees for the PKCS encryption standard [37] degrade linearly in the number of users and ciphertexts. This means that while the scheme—implemented with current parameters and keylengths—is believed to be secure against attacks of complexity 2^{80}, the best guarantees we can currently derive for the same scheme in a 2^{30}-user, 2^{30}-ciphertext setting are almost trivial. (Namely, in that setting, we can only guarantee that any attack on the scheme must have complexity at least 2^{20}, i.e., the equivalent of less than a second of computing time on a modern desktop PC.)[2]

Goals of the "Scalable Encryption" Project. The central goal of the "Scalable Encryption" project is to provide security models and cryptographic schemes that do scale well to Big Data scenarios. In particular, we provide cryptographic constructions that feature a "tight security proof" (i.e., a security reduction which gives guarantees that do not degrade in the size of the application setting). In the following, we will present and highlight the main contributions of the project.

2 Tightly Secure Cryptography

Our first and central concrete goal was to construct cryptographic schemes (and in particular PKE and signature schemes) with security guarantees that do not degrade in larger settings. Technically, we have aimed at constructing cryptographic schemes with a tight security reduction to a standard computational assumption. Several of our works prepared in the course of the "Scalable Cryptography" project have dealt with this topic.

At the core of all of these techniques lies the observation that some computational problems (such as computing discrete logarithms in a cyclic group) are *re-randomizable*. That means that one problem instance I can be re-randomized to obtain many problem instances $I_1,\ldots,I_n$. The solution of any instance I_i will then also yield a solution for the original instance I. To show scalable security of, say, a PKE scheme, one would then start from a single instance I, and seek to embed many re-randomized problem instances I_i in different instances of the PKE scheme. (For instance, a problem instance I_i might correspond to the public key of a PKE instance, while the corresponding problem solution might correspond to the secret key.) If an adversary breaks any PKE instance, this leads to a solution for I_i, which in turn yields a solution for I. In other words, breaking *any* PKE scheme instance (from a selection of many PKE instances) is no easier than breaking a single given problem instance I.

There are a number of interesting computational problems (which are known to be cryptographically useful) with this re-randomizability property. However, the difficulty in executing the aforementioned strategy is to deal with *active* adversaries (that may, e.g., send maliciously formed ciphertexts to an honest user of the encryption scheme to see how this user reacts). Such adversaries may require a security reduction as above to also exhibit at least partial knowledge about the *secret key* of honest users. This makes

[2] We are also cautious when making assumptions about attacker complexity, and will typically assume liberal upper bounds. It should be noted, however, that current (publicly known) supercomputers are known to achieve almost 2^{60} floating-point operations *per second*.

embedding a given challenge (with an *unknown* solution) into PKE instances much harder (since the embedded problem instance might also be easier to solve given that partial knowledge about the secret key).

In our work, we have found various technical ways to embed problem instances into PKE and other cryptographic schemes. Namely, in our work [5] (published at the TCC 2015 conference), we have presented a general framework for constructing PKE, signature, and key exchange schemes with tight security proofs even in the face of *adaptive* corruptions. We note that the emphasis of this work does not lie in practical schemes. We merely describe a general paradigm to achieve an additional security property (security against adaptive corruptions) in large scenarios.

Our work [28] (published at the PKC 2015 conference) presents an identity-based encryption (IBE) scheme secure in large scenarios. While there are previous IBE schemes whose security does not degrade in the number of *users*, our scheme is the first IBE scheme whose security properties do not degrade in the number of *ciphertexts*. Hence, our scheme is the first IBE scheme suitable for the (very realistic) scenario of a large number of encryptions per user. The techniques developed in this work could furthermore be used in our next work, [16] (published at the EUROCRYPT 2016 conference) to develop a tightly secure PKE scheme. Our scheme is the first PKE scheme for large scenarios that does not require a mathematical pairing. As a consequence, our scheme is based upon a very standard computational assumption (the Decisional Diffie-Hellman assumption), and very efficient. This work has been awarded the "Best Paper" at the EUROCRYPT 2016 conference.

Most tightly secure encryption schemes (including the ones from [28] and [16]) share the disadvantage of a large public key. The work [25] (published at the TCC 2016 conference) presents a technique to obtain tightly secure encryption and signature schemes with small public keys (and ciphertexts, resp. signatures). Indeed, we could show that the concepts introduced in [25] lead not only to tightly secure public-key encryption schemes with short public keys (published in [17] at the CRYPTO 2017 conference), but also to tightly secure *structure-preserving* signature schemes (published in [1,18] at the CRYPTO 2017 and EUROCRYPT 2018 conferences), and identity-based encryption schemes [27] (published at ASIACRYPT 2018).

At this point, it might be interesting to explain the importance of *structure-preserving* cryptographic building blocks (like our signature schemes from [1,18]). Informally, a structure-preserving building block is one in which all public operations are algebraic (in a formally defined sense). As a consequence, it is possible to efficiently conduct non-interactive zero-knowledge proofs about their execution (e.g., using the highly efficient proof system of Groth and Sahai [21]). In other words, it is possible to efficiently and publicly prove, e.g., knowledge of a signature without releasing that signature. This enables applications like anonymous credentials (i.e., secure digital identities) which rely on *not* releasing all available information publicly. Our tightly secure structure-preserving signature schemes are the first of their kind, and form highly flexible and universal components for scalable such systems.

Our work [7] (published at the PKC 2015 conference) provides a new framework for obtaining digital signatures with a tight security reduction from standard hardness assumptions. Concretely, we show that any Chameleon Hash function can be transformed into a tree-based signature scheme with tight security. Our framework explains and generalizes most of the existing schemes as well as providing a generic means for constructing tight signature schemes based on arbitrary assumptions, which improves the standard Merkle tree transformation. Moreover, we obtain the first tightly secure signature scheme from the SIS assumption and several schemes based on Diffie-Hellman in the standard model.

Our paper [23] (also published at the PKC 2015 conference) considers security notions for public-key encryption in a slightly more realistic multi-challenge model. We show that two well-known and widely employed public-key encryption schemes—RSA Optimal Asymmetric Encryption Padding (RSA-OAEP) and Diffie-Hellman Integrated Encryption Standard (DHIES)—are secure in this model. Surprisingly, our reductions are optimal in terms of tightness in the sense that they are as tight as the ones for standard security. In the follow-up work [24] (to be published at the ASIACRYPT 2016 conference) we derive new and tight bounds for the composition of symmetric and asymmetric primitives. In particular, we consider the realistic cases where the symmetric part consists of popular modes of operation like CTR, CBC, CCM, and GCM.

We also investigate a similar generic encryption technique, the "Fujusaki-Okamoto" method to achieve secure encryption. Namely, in [26] (published at the TCC 2017 conference), we show that variants of this method achieve tight security or security against quantum computers. Similarly, and even more generically, the work [19] (published at the PKC 2018 conference), investigates the tightness of the generic "KEM-DEM" paradigm to achieve efficient public-key encryption schemes.

In the paper [29] (published at the CRYPTO 2016 conference), we perform a concrete security treatment of digital signature schemes obtained from canonical identification schemes via the Fiat-Shamir transform. If the identification scheme is random self-reducible and satisfies the weakest possible security notion (hardness of key-recoverability), then the signature scheme obtained via Fiat-Shamir is unforgeable against chosen-message attacks in the multi-user setting. Previous reductions incorporated an additional multiplicative loss of N, the number of users in the system. As an important application of our framework, we obtain a concrete security treatment for Schnorr signatures in the multi-user setting.

In the work [3] (published at the CRYPTO 2017 conference), we consider the "memory-tightness" of security reductions, as opposed to the "runtime-tightness" more commonly considered (in particular in most of the works from the previous subsection). Interestingly, this work finds that sometimes, security reductions have an inherent *intrinsic memory usage* (i.e., the reduction necessarily requires a significant amount of memory to perform its job), and that sometimes this memory usage grows with the size of the application setting. This yields another dimension of relations between different problems (and the security of certain cryptographic schemes), and shows that the scalability of cryptographic schemes can be a multi-faceted question.

The work [4] (published at the EUROCRYPT 2020 conference) which does not consider security guarantees (as given, e.g., by a security reduction), but instead investigates how the best concrete attacks on cryptographic schemes scale to larger scenarios. As a result, this work gives lower bounds (and thus also security guarantees) by more directly considering all possible attacks in a generalized setting, the generic group model.

The results we have surveyed so far are concerned with the quality of a security reduction as a measure of scalability. This is a very important factor when deriving concrete security guarantees, but not the only one. For instance, in our work [22] (published at the TCC 2016 conference), we have formalized the notion of reconfigurable cryptographic schemes. A reconfigurable scheme allows to adapt its security parameter (i.e., the quantitative level of given security guarantees) on the fly, without changing all registered user public keys (e.g., for encryption or signature schemes). Hence, reconfigurable cryptographic schemes avoid an expensive update of potentially huge public key databases.

This work also contains proof-of-concept PKE and signature schemes. In these schemes, every user has a long-term public key and secret key. The security of these long-term keys is based on very weak assumptions from the realm of secret-key cryptography: in our PKE scheme, for instance, the public key is the image of the secret key under a generic pseudo-random generator. These long-term keys are not directly used to encrypt or decrypt. Instead, they are used to derive short-term keys (e.g., for the RSA PKE scheme) of any desired bitlength that are then used for encryption or decryption.

3 Post-Quantum Cryptography

The security of all currently used asymmetric (public-key) cryptography relies on the intractability of only two number-theoretic intractability problems, namely the factoring problem and the discrete logarithm problem over elliptic curves and finite fields. This "monoculture" poses a dangerous security threat as, in the not too unlikely scenario of scalable quantum computers, Shor's algorithm will render all the asymmetric cryptosystems in current use immediately insecure: All data transmitted over encrypted channels - past and present - will immediately become public. This in particular also holds for the cryptography considered in the previous section. Leading international tech companies like Google and Microsoft are currently investing in building quantum computers. It can only be speculated whether large intelligence agencies are already in possession of a cryptologically useful quantum computer. For that reason, a number of standardization bodies (such as NIST) are currently selecting quantum-secure asymmetric cryptosystems. Promising candidates for building quantum-resistant asymmetric cryptosystems are, amongst others, based on finding solutions to certain difficult problems regarding codes and lattices. In this project we also worked on the foundations to find truly practical, and at the same time, provably secure encryption schemes, key exchange protocols, signature schemes, and more complex protocols based on well understood and meaningful hard mathematical problems over codes and lattices.

In the context of cryptography, a lattice is a (full-rank) discrete subgroup of R^n, commonly described by a basis. Basic lattice-based cryptosystems have already existed for almost two decades and are arguably among the most promising candidates for quantum-resilience. They are simple and efficient in that their algorithms consist mostly of matrix operations, and they currently resist sub-exponential and quantum attacks. Drawing on the seminal work of Ajtai in 1996 [2], we are able to connect the average-case complexity of lattice problems (upon which the security of our schemes is based) to their complexity in the worst case. The latter property is unique among all known hardness assumptions and is one of the many reasons why people believe in its intractability. In this context the "learning with errors" (LWE) problem emerged as a suitable abstraction for a hard problem on lattices since it was shown that solving this problem would imply breaking a few well-studied lattice-problems in the worst case, such as the approximate shortest vector problem.

In [11] (published at EuroS&P 2018) we proposed Kyber, a simple and fast encryption scheme. The design of Kyber has its roots in the seminal LWE-based encryption scheme of Regev [34]. Since Regev's original work, the practical efficiency of passively secure LWE encryption schemes has been improved by observing that the secret key can come from the same distribution as the noise and also noticing that"LWE-like" schemes can be built by using a square (rather than a rectangular) matrix as the public key. Kyber does some further efficiency improvements such as dropping several bits from the public-keys and ciphertexts to save bandwidth. At the core of its security analysis lies the security reduction of the Fujusaki-Okamoto transformation [26] already mentioned in Sect. 2, which transforms any passively secure encryption scheme into one withstanding active adversaries. The key feature here is that the security reduction is tight, i.e., it does not degrade with the number of evaluations of the hash function. This, together with Kyber's extremely fast performance, makes it very suitable for big-data scenarios. As of 2020, Kyber has been selected by the NIST as one of the finalists of its Post-Quantum Cryptography Standardization process for public-key encryption.[3]

4 Open Questions

Although the project significantly advanced our understanding of scalable security (and in particular scalable security *guarantees*), many questions remain. First, we are still missing technical tools to tackle the tight security of all cryptographic building blocks: the tight security (and thus the scalability) of *hierarchically organized* schemes (such as HIBE or hierarchical signature schemes) is not well-understood, and most known results (such as [30]) are negative. Besides, there are few results about the scalability of new and modern cryptographic building blocks such as obfuscation or functional or homomorphic encryption schemes. Even though these building blocks are extremely powerful (and imply a multitude of other building blocks and tasks), their scalability is currently unclear.

Moreover, the interplay between cryptanalytic attacks and the guarantees given by security reductions is generally not well-understood. The work of [4] is a promising step in this direction, but there remains a lot to be done.

References

1. Abe, M., Hofheinz, D., Nishimaki, R., Ohkubo, M., Pan, J.: Compact structure-preserving signatures with almost tight security. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10402, pp. 548–580. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63715-0_19
2. Ajtai, M.: Generating hard instances of lattice problems (extended abstract). In: 28th ACM STOC, pp. 99–108. ACM Press (1996). https://doi.org/10.1145/237814.237838

[3] https://csrc.nist.gov/projects/post-quantum-cryptography.

3. Auerbach, B., Cash, D., Fersch, M., Kiltz, E.: Memory-tight reductions. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10401, pp. 101–132. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63688-7_4
4. Auerbach, B., Giacon, F., Kiltz, E.: Everybody's a target: scalability in public-key encryption. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12107, pp. 475–506. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45727-3_16
5. Bader, C., Hofheinz, D., Jager, T., Kiltz, E., Li, Y.: Tightly-secure authenticated key exchange. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015. LNCS, vol. 9014, pp. 629–658. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46494-6_26
6. Bellare, M., Rogaway, P.: Optimal asymmetric encryption. In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 92–111. Springer, Heidelberg (1995). https://doi.org/10.1007/BFb0053428
7. Blazy, O., Kakvi, S.A., Kiltz, E., Pan, J.: Tightly-secure signatures from chameleon hash functions. In: Katz, J. (ed.) PKC 2015. LNCS, vol. 9020, pp. 256–279. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46447-2_12
8. Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 1–12. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055716
9. Blum, M., Goldwasser, S.: An efficient probabilistic public-key encryption scheme which hides all partial information. In: Blakley, G.R., Chaum, D. (eds.) CRYPTO 1984. LNCS, vol. 196, pp. 289–299. Springer, Heidelberg (1985). https://doi.org/10.1007/3-540-39568-7_23
10. Bogetoft, P., et al.: Secure multiparty computation goes live. In: Dingledine, R., Golle, P. (eds.) FC 2009. LNCS, vol. 5628, pp. 325–343. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03549-4_20
11. Bos, J.W., et al.: CRYSTALS - kyber: a CCA-secure module-lattice-based KEM. In: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, 24–26 Apr 2018, pp. 353–367. IEEE (2018). https://doi.org/10.1109/EuroSP.2018.00032
12. Cramer, R., Shoup, V.: A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 13–25. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055717
13. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Trans. Inf. Theory **22**(6), 644–654 (1976)
14. Dolev, D., Dwork, C., Naor, M.: Non-malleable cryptography (extended abstract). In: 23rd ACM STOC, pp. 542–552. ACM Press (1991). https://doi.org/10.1145/103418.103474
15. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Trans. Inf. Theory **31**, 469–472 (1985)
16. Gay, R., Hofheinz, D., Kiltz, E., Wee, H.: Tightly CCA-secure encryption without pairings. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9665, pp. 1–27. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49890-3_1
17. Gay, R., Hofheinz, D., Kohl, L.: Kurosawa-desmedt meets tight security. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10403, pp. 133–160. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63697-9_5
18. Gay, R., Hofheinz, D., Kohl, L., Pan, J.: More efficient (almost) tightly secure structure-preserving signatures. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018. LNCS, vol. 10821, pp. 230–258. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78375-8_8
19. Giacon, F., Kiltz, E., Poettering, B.: Hybrid encryption in a multi-user setting, revisited. In: Abdalla, M., Dahab, R. (eds.) PKC 2018. LNCS, vol. 10769, pp. 159–189. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-76578-5_6

20. Goldwasser, S., Micali, S.: Probabilistic encryption. J. Comput. Syst. Sci. **28**(2), 270–299 (1984)
21. Groth, J., Sahai, A.: Efficient non-interactive proof systems for bilinear groups. In: Smart, N. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 415–432. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78967-3_24
22. Hesse, J., Hofheinz, D., Rupp, A.: Reconfigurable cryptography: a flexible approach to long-term security. In: Kushilevitz, E., Malkin, T. (eds.) TCC 2016. LNCS, vol. 9562, pp. 416–445. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49096-9_18
23. Heuer, F., Jager, T., Kiltz, E., Schäge, S.: On the selective opening security of practical public-key encryption schemes. In: Katz, J. (ed.) PKC 2015. LNCS, vol. 9020, pp. 27–51. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46447-2_2
24. Heuer, F., Poettering, B.: Selective opening security from simulatable data encapsulation. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016. LNCS, vol. 10032, pp. 248–277. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53890-6_9
25. Hofheinz, D.: Algebraic partitioning: fully compact and (almost) tightly secure cryptography. In: Kushilevitz, E., Malkin, T. (eds.) TCC 2016. LNCS, vol. 9562, pp. 251–281. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49096-9_11
26. Hofheinz, D., Hövelmanns, K., Kiltz, E.: A modular analysis of the Fujisaki-Okamoto transformation. In: Kalai, Y., Reyzin, L. (eds.) TCC 2017. LNCS, vol. 10677, pp. 341–371. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70500-2_12
27. Hofheinz, D., Jia, D., Pan, J.: Identity-based encryption tightly secure under chosen-ciphertext attacks. In: Peyrin, T., Galbraith, S. (eds.) ASIACRYPT 2018. LNCS, vol. 11273, pp. 190–220. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03329-3_7
28. Hofheinz, D., Koch, J., Striecks, C.: Identity-based encryption with (almost) tight security in the multi-instance, multi-ciphertext setting. In: Katz, J. (ed.) PKC 2015. LNCS, vol. 9020, pp. 799–822. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46447-2_36
29. Kiltz, E., Masny, D., Pan, J.: Optimal security proofs for signatures from identification schemes. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. LNCS, vol. 9815, pp. 33–61. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53008-5_2
30. Lewko, A., Waters, B.: Why proving HIBE systems secure is difficult. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 58–76. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55220-5_4
31. Merkle, R.C., Hellman, M.E.: Hiding information and signatures in trapdoor knapsacks. IEEE Trans. Inf. Theory **24**(5), 525–530 (1978)
32. Naor, M., Yung, M.: Public-key cryptosystems provably secure against chosen ciphertext attacks. In: 22nd ACM STOC, pp. 427–437. ACM Press (1990). https://doi.org/10.1145/100216.100273
33. Rackoff, C., Simon, D.R.: Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 433–444. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-46766-1_35
34. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: 37th ACM STOC, pp. 84–93. ACM Press (2005). https://doi.org/10.1145/1060590.1060603
35. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978)
36. RSA laboratories: PKCS #1: RSA encryption standard, version 1.5 (1993)
37. RSA laboratories: PKCS #1: RSA Cryptography standard, version 2.1 (2002)
38. Shoup, V.: Why chosen ciphertext security matters. Technical report RZ 3076, IBM Zurich Research Laboratory (1998)
39. Zimmermann, P.R.: PGP: pretty good privacy (1991)

Distributed Data Streams

Jannik Castenow, Björn Feldkord, Jonas Hanselle, Till Knollmann, Manuel Malatyali, and Friedhelm Meyer auf der Heide(✉)

Paderborn University, Paderborn, Germany
{jannik.castenow,bjoern.feldkord,jonas.hanselle, manuel.malatyali,fmadh}@upb.de, tillk@mail.upb.de

Abstract. We consider a scenario where a server is wirelessly connected to countless sensor nodes that continuously measure data. The task of the server is to monitor the sensors' data. More precisely, at each time step the server calculates a function defined over the current measurements of the sensors. Since the sensors only have small computational power and tight battery constraints, the communication between the server and the sensors should be as small as possible, i.e., we aim at minimizing the total number of messages that is transferred.

There are various conceivable problems for the setting above. We demonstrate our approaches on the following three: In the Top-k-Value Monitoring Problem, the server aims at identifying the k largest values. The Top-k-Position Monitoring Problem shifts the task to identify the sensors observing these values. Finally, the Count Distinct Monitoring Problem obliges the server to determine the number of distinct values currently observed.

For all three problems, we not only present communication-efficient protocols for one time step, we also show how it can be exploited if the input at sensors is similar between consecutive time steps to reduce the total communication on the long term. Thereby, we utilize different techniques – involving sampling, dynamic data structures, filter-based approaches, and combinations of them – to demonstrate their potential and their limits in the broad setting described above.

Keywords: Top-k · Count distinct · Distributed monitoring · Distributed data streams

1 Introduction

Envision a scenario where a set of tiny, lightweight sensors is distributed in a hazardous area (e.g., an ocean, high mountains or in space) to monitor the environment. The sensors are connected to one or multiple central servers which have the task to track the measurements of the sensors, i.e., the servers have to compute a function of the sensor values at every point in time. This task is easy to solve as long as the sensors continuously send their current measurements to the servers and the latter ones have enough memory and computational power to do computations on the sensor data at every point in time. Realistic applications, however, require a huge number of sensors (e.g., because the area is very large, sensors are error-prone, sensors have only a limited battery lifetime, ...) that cannot be handled by modern server hardware, or the number of required severs might be uneconomically expensive. Additionally, sending the measured data of the sensors continuously to the servers leads to a rapid decrease of the sensors' batteries. Therefore, to build a feasible system, the communication between sensors and servers needs to be severely reduced.

We consider two types of randomized algorithmic approaches to reduce the communication: The first approach is based on Monte Carlo algorithms. Sensors decide randomly to communicate their current observed value to the server. The probability of sending a message depends on the significance of the current observed value: If the impact on the output function is small, the probability of sending a message is low; if the impact is high also the probability of sending a message is high. Thus, the server is not aware of all changes in the sensor values but with a high probability it gets to know all significant changes. With this approach, the server is able to compute a correct output with a high probability. In some scenarios (for instance in safety-critical systems), the application demands to always compute a correct output. Here, we exploit the idea of Las Vegas algorithms that reduce the number of sent messages with a high probability but always compute the exact output. With a low probability many messages may be sent, however the server can always be sure to compute the correct output. All in all, these two approaches build a trade-off between reducing the communication and computing correct outputs, while the randomization helps to keep the trade-off small.

Considering the scenario described above, we are interested in multiple problems. In the Top-k-Value Monitoring problem, the server is interested in the k largest values observed by the sensors at any time. In contrast to that, the Top-k-Position Monitoring problem tackles the case where the server is interested in the actual sensors measuring the k largest values, e.g., to track if large values and the set of sensors observing them are correlated. As in a lot of cases a rough estimate on the top k positions is sufficient, we also address the Approximate Top-k-Position Monitoring problem. Besides the largest values, the server might also be interested in how many different values are observed to get an overview on the global situation. This is captured in the (Approximate) Count Distinct Monitoring problem.

The aforementioned problems have in common that in practice a lot of communication can be avoided compared to a naïve approach that gathers all sensor data at every time step at the server. For example, consider the (Approximate) Count Distinct Monitoring problem. If a subset of the sensors observes identical values, not all of them need to communicate their observation to the server. See Fig. 1 for a depiction. Here, a horizontal block indicates a set of sensors observing the same value at the same time. Optimally, only one of them would need to communicate the value to the server. Additionally, if the value that is observed by a fixed sensor does not change significantly as time goes, the sensor does not need to notify the server all the time. Furthermore, observations of sensors that are not of interest should not be communicated. This can be seen when considering the Top-k-Value Monitoring problem. It would be best if all sensors not

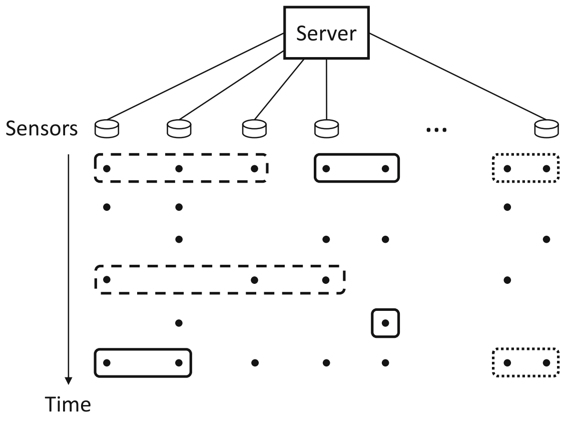


Fig. 1. We consider a central server that is connected to a set of sensor nodes. As time goes, each sensor observes a sequence of values (indicated by the dots below the sensors). Among others, communication can be avoided if a group of sensors observes the same value at the same time (horizontal blocks).

observing one of the k largest values do not communicate at all. Note, that for this it is required that all the sensors can receive information from the server. In our model, we allow the server to have a cheap broadcast channel. This can be assumed, as the central server has no need to reduce its power consumption as opposed to the sensors.

In this paper, we examine how communication can be minimized in the problems above. Our focus is especially a theoretical analysis of techniques that allow to capture the idea that sensor data might not change arbitrarily between consecutive time steps. We examine, among other things, how to use dynamic data structures and restrictions on the adversary dictating the inputs at sensor nodes, such that an algorithm can keep/update an existing solution for more than one time step and reduce the overall communication.

We begin in Sect. 2 by a formal introduction of our model and the problems we consider. We also introduce a major technique that we use called *filters*. Afterwards, we establish computational primitives in Sect. 2.3. In Sect. 3 we deal with the Top-k-Value Monitoring problem followed by the Top-k-Position Monitoring problem in Sect. 4 and the (Approximate) Count Distinct Monitoring problem in Sect. 5.

This paper surveys results from [1 SPP, 4 SPP, 8 SPP, 9 SPP, 10 SPP] . We only give short sketches of algorithms and proofs. For technical details, please look at the papers above. A detailed description of the current state of the art is presented in [10 SPP].

2 Model

In our setting there are n nodes connected to a single server. The nodes are uniquely identified by IDs from the set $\{1, \ldots, n\}$ and each node i receives

$(v_i^1, v_i^2, v_i^3 \ldots)$ as a stream of data. At time t, a node i observes $v_i^t \in \mathbb{N}$ and does not know any $v_i^{t'}$, $t' > t$. The superscript t is omitted if it is clear from the context.

Following the model in [3], we allow that between any two consecutive time steps, a *communication protocol* exchanges messages between the server and the nodes. The communication protocol is allowed to use a number of rounds polylogarithmic in n and $\max_{1\leq i\leq n}(v_i^t)$. Nodes can only send messages to the server and they are able to store a constant number of integers, compare two integers and perform Bernoulli trials with success probability $2^i/n$ for $i \in \{0, \ldots, \log n\}$. The server can communicate to one node directly or utilize a broadcast-channel to send one message to all nodes simultaneously. All communication methods described above incur unit communication cost per message, the delivery is instantaneous, and we allow a message at time t to have a size which is logarithmic in n and $\max_{1\leq i\leq n}(v_i^t)$.

A time step t defines a point in time at which the sensor nodes obtain a new piece of input (v_i^t for node i at time t). The protocol consists of multiple (communication) rounds: Each sensor node performs local computations and may send a message to the server. The server collects all messages, performs local computations and may send a message via the broadcast-channel to all sensor nodes.

Since all nodes are synchronized, the server can detect if no sensor sends a message and the sensor nodes can identify if the server did not send a message. Furthermore, the server has unrestricted capacity when receiving, i.e., it can always receive all messages that are send to it.

At the end of each round, when the communication protocol terminated, the server decides on the output of the function for the current time t and the whole network proceeds to the next time step $t+1$.

We assume that all observed values are pairwise different for the (Approximated) Top-k-Value and Top-k-Position Monitoring Problems and coupe with a large number of duplicates considering the (Approximate) Count Distinct Problem.

2.1 Problems

Our focus here is on three problems; the Top-k-Value Monitoring problem, the Top-k-Position Monitoring problem and the Count Distinct Monitoring problem. In the Top-k-Value Monitoring problem, we are interested in the largest observed values, i.e., the ordering of the values is of special interest. Let $s_1^t, \ldots, s_n^t$ be the values observed at time t $(v_1^t, \ldots, v_n^t)$ sorted in descending order.

Definition 1 (Top-k-Value Monitoring). *In the Top-k-Value Monitoring problem, the server has to output $s_1^t, \ldots, s_k^t$, $k \leq n$ at each time t.*

In contrast to keeping track of the values it might be more of interest to keep track of the nodes observing the largest values instead (for instance in safety critical applications). This is considered in the Top-k-Position Monitoring problem.

Definition 2 ((Approximate) Top-k-Position Monitoring). *In the Top-k-Position Monitoring problem, the server has to output at each time t the k nodes observing $s_1^t, \ldots, s_k^t$ – called the top-k. If we are interested in an approximation, we need some more notation. For any constant $\varepsilon \in (0,1)$ let $E(t) := \{i \,|\, v_i^t \in ((1-\varepsilon)^{-1}\, s_k^t, \infty)\}$ be the set of nodes observing values which are significantly larger than the kth largest one. In the Approximate Top-k-Position Monitoring problem, at each time t the server has to output $E(t)$ and $k - |E(t)|$ many nodes not in $E(t)$ observing a value which is at least $(1-\varepsilon)\, s_k^t$.*

In the case that multiple nodes observe the same value, one might be more interested in how many different values are observed. We approach this direction by the Count Distinct Monitoring problem. Note, that we do not assume all values to be distinct when discussing this problem.

Definition 3 ((Approximate) Count Distinct Monitoring). *For a fixed time step t, let d_t be the number of distinct values observed by all nodes, i.e., $d_t = |\{v_i^t \,|\, i \in \{1, \ldots, n\}\}|$. At each time step t the server has to output d_t. In the approximation variant, the server has to output an (ε, δ)-approximation at each time step t, i.e.; for two constants $0 \leq \epsilon, \delta \leq 1$, the server has to compute a value $x \in [(1-\varepsilon) \cdot d_t, (1+\varepsilon) \cdot d_t]$ with probability at least $1-\delta$.*

Explicitly, the values at times $t' < t$ do not matter for the output at time t.

2.2 Filter-Based Algorithms

One of our main techniques is the usage of *filters*. A filter defines for each sensor an interval of values that do not influence the output function. Filtering the input for an algorithm occurs in many different contexts. In algorithm engineering, filtering has turned out to be a valuable tool to decrease the input size to speed up the computation in certain cases. For instance the *Filter-Kruskal* algorithm can accelerate the computation of minimum spanning trees of graphs [12]. It improves the *qKruskal* algorithm which combines the original *Kruskal* algorithm with the partitioning idea of *QuickSort* – the edges are not sorted beforehand but a pivot edge is chosen, the problem is solved recursively on all edges with smaller weight and afterwards (provided the spanning tree is still incomplete) on all edges having a larger weight. *Filter-Kruskal* improves this by not using all edges of larger weight as an input for the second recursive call but only those edges which actually connect two different components of the graph, i.e., it filters all edges that cannot be part of the minimum spanning tree. This idea has also been applied to different problems later on, e.g., graph matching [11].

Another filtering approach with numerous applications is *Kalman Filtering*, also known as *Linear Quadratic Estimation*. Its goal is to predict the state of a system based on observations containing inaccuracies. It works in two steps: First, system parameters are predicted and afterwards, the predictions are updated as soon as the next observation (measurement with inaccuracies) arrives using a stochastic weighted average approach. Applications of Kalman

Filtering can be found in various areas, among others navigation control of vehicles, robot motion planning, and signal processing. It is also provably a valuable tool for data stream analysis. Similar to our goal, Kalman Filtering is used to reduce the communication in a sensor server architecture in [5]. Here, Kalman Filtering is applied on both the server and the sensor side (the sensors provide a data stream for the server). As long as the sensor observes values that are within a small deviation of its current prediction, the sensor does not communicate to the server. Once the deviation exceeds a certain threshold, the sensor updates the server.

Next, we introduce the formal notion of filters and necessary definitions for our model. A set of filters is a collection of intervals, one assigned to each node such that, as long as the observed values at each node are within the given interval, the value of the output function does not change.

Definition 4. *For a fixed time t, a* set of filters *is defined by an n-tuple of intervals $(F_1^t, \ldots, F_n^t)$, $F_i \subseteq \mathbb{N} \cup \{-\infty, \infty\}$ with $v_i^t \in F_i^t$, such that as long as the value of node i only changes within its interval,i.e., it holds $v_i^{t'} \in F_i^{t'} = F_i^t$ for $t' \geq t$, the value of the output function does not change. We use $F_i^t = [\ell_i^t, u_i^t]$ to denote the lower and upper bound of a filter interval, respectively.*

We assume that nodes are assigned filters by the server. If a node *violates* its filter, i.e., the currently observed value is not contained in its filter, the node may report the violation and its current value to the server. The server then computes a new set of filters and sends them to the affected nodes. To calculate a set of filters that works for the entire set of nodes, the server may need to probe some more nodes before sending out the new filters. At the end of each time step, no node is allowed to violate its filter. An algorithm following this approach is called *filter-based*.

The easiest way of defining a set of filters is to assign the value a node currently observes as its interval. In this case the usage of filters is not very beneficial, so we are looking for filters that are as large as possible to minimize the number of filter changes which is directly related to the number of exchanged messages.

Our analysis is based on the classical competitiveness approach first used in [13] and later on formalized by [6]; see also [2] for an overview. We compare the communication volume of our algorithms to one of an appropriately defined offline algorithm. In our model, a general offline algorithm knows all the input streams in advance and can trivially solve the aforementioned problems without any communication. To still get meaningful results regarding the quality of our algorithms, we assume the optimal offline algorithm OPT uses filters assigned by the server to the nodes. To lower bound the cost of OPT, we count the number of filter updates over time.

Definition 5 (Competitive Algorithms). *We call a (randomized) online algorithm* ALG c-competitive *if for every instance its (expected) communication volume is by a factor of at most c larger than the communication volume of* OPT*.*

2.3 Computational Primitives

This section is dedicated to three subroutines that will be used in later algorithms. Due to space constraints, we will use these protocols mainly as black-boxes, see the cited literature for more details. The first subroutine is a protocol for the EXISTENCE problem. In this problem, all nodes observe binary values, $\forall i \in \{1, \ldots, n\} : v_i \in \{0, 1\}$ and the goal for the server is to output the *logical disjunction.*

The EXISTENCE PROTOCOL solves this problem in $\log(n) + 1$ rounds. In each round $r = 0, 1, \ldots, \log n$, all nodes that have observed the value 1 send a message with probability $2^r/n$ to the server. As soon as the first message reaches the server, the protocol ends (latest if $r = \log(n)$ holds).

Theorem 1 (Existence). *[9 SPP] There exists an algorithm* EXISTENCE PROTOCOL *which uses $\mathcal{O}(1)$ messages in expectation and at most $\log n + 1$ communication rounds to solve the problem* EXISTENCE.

The EXISTENCE PROTOCOL has several applications. Most important for our research is the detection of filter violations. The server can detect a filter violation using only a constant number of messages on expectation.

Corollary 1 (Filter Violation). *[9 SPP] There is a protocol* EXISTENCE PROTOCOL *which uses $\mathcal{O}(1)$ messages in expectation to identify a filter violation. In case there are multiple filter violations one is drawn uniformly at random. If no filter violation occurs no message takes place.*

Additionally, the TOP-K PROTOCOL is able to solve the Top-k-Value Monitoring problem. The protocol uses similar ideas as the EXISTENCE PROTOCOL: Nodes draw a height from a geometric distribution and a tree like structure is built. Initially, s_1 is determined by collecting a sample of all values, broadcasting the largest one and continuing until s_1 is determined. The same idea is used to find $s_2, \ldots, s_k$.

Theorem 2 (Top-k). *[4 SPP] The* TOP-K PROTOCOL *uses $k + \log(n) + 2$ messages in expectation and $\mathcal{O}(k + \log n)$ expected number of rounds to solve the Top-k-Value Monitoring problem.*

3 Top-k-Value Monitoring

In this section we consider problems regarding the k largest values at the current time step t. We design and analyze Las Vegas algorithms, i.e., we always output the correct values and can show that the total communication and number of rounds are polylogarithmic with high probability.

Consider a general input for the Top-k problem over time. The values might change over time as well as the nodes holding the Top-k values. As a consequence, in a worst-case situation we cannot reuse any information from previous time steps and need to recompute the output from scratch. To counteract this possibility, we consider two different approaches.

In our first approach, we restrict the number of values which can change between queries, and parameterize the result in this number. We show that we can build up a data structure which preserves important information as long as there are not too many updates. This makes answering queries much more efficient, as we use the data structure to quickly reduce the number of candidate nodes which potentially hold the desired result.

In our second approach, we consider filter-based algorithms for the problem. These algorithms have the advantages discussed in Sect. 2.2, i.e., they are very effective if the changes in the output are not too large. To conduct a meaningful worst-case analysis, we consider the competitiveness of the algorithms against a filter-based offline algorithm.

Before we give the details of our solutions, we shortly mention that our protocols which compute the Top-k from scratch are essentially optimal with respect to the amount of communication. Intuitively, we can show that an algorithm cannot do much better than performing a binary search on n values. The algorithm can always ask a set of nodes for their value, and then broadcast the maximum to 'eliminate' all nodes with a smaller values for the process. Formally, Yao's minimax principle considering a random permutation as input can be applied. Each input occurs with probability $(1/n!)$ and it is shown that any deterministic algorithm needs at least $\Omega(\log n)$ messages on expectation which yields:

Theorem 3 **([8 SPP]).** *Every comparison-based randomized algorithm requires at least $\Omega(\log n)$ messages on expectation to compute the maximum in our model.*

3.1 Dynamic Distributed Data Structure

In this section we consider a data structure for the rank related problems of Top-k and k-Select. The k-Select problem asks to identify the data item with rank k. We consider the approximate version, where we have to output an item with rank in $[(1-\varepsilon)k, (1+\varepsilon)k]$ with probability at least $1-\delta$. An approximate version with weaker conditions will also help us to solve the Top-k problem. For the bounds on communication, we consider the following setting: Only when there is a query for the Top-k or for k-Select, the output is determined. We allow the parameters to be different from query to query and, furthermore, we allow multiple k-Select queries at the same time step.

Our results are based on the idea of maintaining a (distributed) data structure which is used to answer a query and is informed about each update. More precisely, at every point in time, the data structure keeps track of an approximation of a data item with rank k. These approximations can be exploited by the protocols for a Top-k or k-Select computation to significantly decrease the communication and, interestingly, also the time bounds, rendering this approach very powerful.

The data structure supports the following operations: `Top-k`: Output $\{s_1^t, \ldots, s_k^t\}$, `StrongSelect`: Output $d \in \{s_{(1-\varepsilon)k}^t, \ldots, s_{(1+\varepsilon)k}^t\}$ and `WeakSelect`:

Output d with $s^t_{k\cdot\log^{c_1} n} \leq d \leq s^t_{k\cdot\log^{c_2} n}$, with $c_1, c_2 > 1$. The `Top-k` and `StrongSelect` operations answer queries for the Top-k and k-Select problems, while the operation `WeakSelect` supports the other two. Our data structure guarantees the following:

Theorem 4 ([4 SPP]). *There is a distributed data structure with expected amortized total communication cost for an update of $\mathcal{O}(1/\operatorname{polylog} n)$. The amortized number of rounds for an update is $\mathcal{O}(1)$. The data structure is able to answer a k-Select query correctly with probability at least $1-\delta$. For that, $\mathcal{O}(1/\varepsilon^2 \log 1/\delta + (\log\log n)^2)$ messages and $\mathcal{O}(\log\log \frac{n}{k})$ rounds are required on expectation. Additionally, the expected total communication cost to answer a Top-k query is $\mathcal{O}(k+\log\log n)$ and the expected number of rounds is $\mathcal{O}(\log\log n)$. The output is always correct.*

Our data structure is designed as follows. We maintain a *Sketch(t)* about the data items received at time t in the server. The task of such a sketch is to maintain items to answer `WeakSelect` queries instantly. A *Sketch(t)* is a subset of data items denoted by $\{r^t_1, \ldots, r^t_m\}$, where $m \leq \log n$. We call *Sketch(t)* correct if it consists of a set of data items $\{r_1, \ldots, r_m\}$ such that, for each $k = 1, \ldots, n$, there exists r_k such that $s^t_{k\cdot\log^{c_1} n} \leq r_k \leq s^t_{k\cdot\log^{c_2} n}$. We say the data item r_k is the representative of the set of data items d with $s_{k\cdot\log^{c_1} n} \leq d \leq s_{k\cdot\log^{c_2} n}$. To answer the `WeakSelect` query for a specific rank in $[k\cdot\log^{c_1} n, k\cdot\log^{c_2} n]$, we simply output the representative r_{k+1}.

Computing a *Sketch* is somewhat expensive, hence we want it to be valid even after some values have been updated. It is easy to see that for appropriately chosen constants c_1, c_2, up to $\log^c n$ values can change without the property being lost. In conclusion, we can achieve the stated performance guarantees by computing a *Sketch* which is valid for $\log^c n$ updates, after which we recompute it from scratch. The `WeakSelect` operation simply returns an appropriate element from the *Sketch*.

Now, recall that there is a protocol for `Top-k` which uses $k + \log(n) + 1$ messages and $\mathcal{O}(k + \log n)$ rounds in expectation (Theorem 2). These bounds hold when the protocol is executed on n nodes without using any information from previous time steps. We can now utilize our *Sketch* in the following way: We execute a `WeakSelect` operation with input k, such that we receive a data item d of size at most $s^t_{k\cdot\log^{c_2} n}$. Then, we execute the `Top-k` protocol only for nodes which hold a data item smaller than d, i.e., we execute the protocol only on $\mathcal{O}(k\log n)$ nodes instead of n, yielding the desired bound. The bound on the `StrongSelect` operation can be obtained in a similar fashion.

3.2 Filter-Based Algorithm

We turn our attention to filter-based algorithms which we evaluate in the framework of competitive analysis. We are going to compare the algorithm against an optimal offline algorithm, which knows all of the future input in advance. To make this analysis meaningful, it is necessary to also restrict the offline algorithm to a filter-based approach. The important part of the filter-based approach is that the offline algorithm has to communicate a set of valid filters to the nodes. In accordance to Definition 4, this means that the offline algorithm at least has to communicate each time the output changes.

The algorithm works as follows: First, the k largest values are determined using the Top-k-Protocol of Theorem 2 . Afterwards, the server broadcasts s_k such that all nodes i with $v_i \geq s_k$ define their filter to $F_i := [v_i, v_i]$ and the remaining nodes i with $v_i < s_k$ to $F_i := [-\infty, s_k]$. Whenever a node with one of the k largest values observes a different value, a filter violation occurs such that the node sends a message to the server. Each of the other nodes (those with filters $F_i := [-\infty, s_k]$) that observes a filter violation executes the Top-k-Protocol (to prevent that every node sends a message). The server unifies and outputs the k largest values of the nodes without a filter violation from the past time step and the new values of the current time step. This algorithm has the following guarantees.

Theorem 5 **([4 SPP]).** *There is an online algorithm which monitors the Top-k-Values and is $\mathcal{O}(k + \log n)$-competitive against an optimal filter-based offline algorithm.*

4 Top-k-Position Monitoring

In this section we consider monitoring the IDs of the nodes which observe the Top-k values rather than the values themselves [8 SPP,9 SPP]. The intuitive advantage is that small updates to the values of the nodes holding the Top-k do not necessarily mean a change in the Top-k positions. Hence, in a scenario where there are a lot of small fluctuations in the observed values but the overall ranking of nodes stays the same, we have to utilize much less communication if we monitor the nodes.

We only consider filter-based algorithms in this section. For the general approach as in Sect. 3.1, there is no further benefit from monitoring only the positions, as the entire data structure approach aims at optimizing cases in which only a fraction of nodes observe new values. On the other hand, it directly provides a solution for the positions since nodes can always send their IDs along with their values.

For the filter-based algorithm, we expect less communication due to the reason explained above. In fact, we observe an increase in the competitive ratio for the position monitoring: Under worst-case input sequences, the offline algorithm can gain a greater advantage in comparison to the online algorithm.

Theorem 6 **([10 SPP]).** *Let each sensor node observe values from $1, \dots, \Delta$. There is an online algorithm which monitors the Top-k-Positions and has a competitiveness of $\mathcal{O}(k + \log n + \log \Delta)$ compared to a filter-based offline algorithm.*

4.1 Filter-Based Top-k-Position Monitoring

The main observation for our approach is that for this problem it is sufficient to send only a single value v which divides the Top-k from the remaining nodes, i.e., a value which is between the kth and the $(k+1)$st largest value. Based on this observation, the main task for the online algorithm is to decide where to set the value v which divides the Top-k and the remaining sensor nodes from each other. Since no information about the future is known, and the adversary has no restriction in the process of generating the values that the sensor nodes observe in future time steps, we simply take the median value.

Top-k Position protocol: Initially identify the kth and $(k+1)$st largest values and the respective sensor nodes (using the one-shot protocol). As long as the Top-k-Positions do not change, define the bound for the filters as the median value between the kth and the $(k+1)$st largest value.

In addition to the execution of the one-shot protocol from Theorem 2, this strategy yields additional $\mathcal{O}(\log \Delta)$ messages in expectation by applying the Existence Protocol from Theorem 1 for identifying a filter violation. Violations can occur until we have found the correct separation between the kth and $(k+1)$st largest value, which takes at most $\log \Delta$ steps, because by choosing the median value, we essentially perform a binary search for the correct value. Note, that since the adversary is offline adaptive, it is easy to see that every online algorithm needs at least $\Omega(\log \Delta)$ messages which easily translates to an overall lower bound of $\Omega(k + \log n + \log \Delta)$ on the competitiveness for any randomized online algorithm. By this, the bound in Theorem 6 is asymptotically tight.

While this strategy performs generally well under minimal changes to the input values, a lot of communication can occur if, e.g., the nodes holding the kth and $(k+1)$st largest values often switch positions, but these values are almost the same. In such a situation, it might be sufficient not to take note of the exact Top-k (e.g., for outdoor temperature one degree differences might not matter to us). We address this by proposing an algorithm calculation positions for Approximate Top-k as by Definition 2.

4.2 Filter-Based Approx. Top-k-Position Monitoring

In this section we allow the online algorithm to have some errors in its output and compare against an optimal offline algorithm which solves the exact problem. Recall that monitoring the Approximate Top-k-Positions allows (only) the online algorithm to choose nodes as an output which are 'close' to the kth largest value (see Definition 2). Observe that filters are allowed to overlap if we consider the relaxation of the Top-k-Position problem.

We want to make use of the allowed error in the following way: When solving the exact problem, we had to search the value domain for the correct separation between the kth and $(k+1)$st ranked value. Allowing an error means that we only need to find an approximation of this separating value, resulting in a faster search. In fact, if we introduce an additive error (say M), it is easy to see that the competitiveness compared to a filter-based offline algorithm which solves the exact problem is reduced from $\mathcal{O}(k+\log n+\log \Delta)$ to $\mathcal{O}(k+\log n+\log(\Delta - M))$.

However, if we use the standard notion of a multiplicative error the following disadvantage occurs: If the values we search for are smaller, the range of values which lie within the margin of error also becomes smaller. So in a way, the criterion for a valid outputs becomes stricter when dealing with smaller values.

To circumvent this shortcoming, we first apply a binary search strategy on a logarithmic scale which terminates after $\log \log \Delta$ filter violations and stops with the property that the allowed error can only vary within constant factors. Applying the approach from the algorithm in Theorem 6 with an early stopping rule, the following can be achieved:

Theorem 7 ([10 SPP]). *Let each sensor node observe values from $1, \ldots, \Delta$. There is an online algorithm which monitors the Approximate Top-k-Positions with a competitiveness of $\mathcal{O}(k + \log n + \log \log \Delta + \log 1/\varepsilon)$ compared to a filter-based offline algorithm which monitors the exact Top-k-Positions.*

4.3 Approximate Offline Algorithm

In this section, we study a variant in which the optimal offline algorithm is allowed to introduce an error, i.e., both the online and offline algorithms monitor the Top-k-Positions approximately. It turns out that it is much more challenging for online than for offline algorithms to take advantage of the relaxed conditions for a correct output, resulting in a significantly higher competitive ratio. This fact is formalized in a lower bound of $\Omega(n)$ (for constant k) [9 SPP], which is much larger than previous upper bounds of $\mathcal{O}(k + \log n + \log \Delta)$ for the exact problem. Intuitively speaking, the online algorithm has to choose where to set filters, but also has to choose a subset of nodes the output is based on which significantly increases the lower bound:

Theorem 8 ([10 SPP]). *Any filter-based online algorithm which solves the approximate Top-k Position Monitoring problem cannot be better than $\Omega(n + \log \Delta)$-competitive.*

We consider two settings in which we compare to an approximate offline algorithm and design algorithms for the respective settings: First, an online algorithm has the task to solve the problem with the same error ε as the offline algorithm and second, an online algorithm is allowed to use 2ε, i.e., twice the error of the offline algorithm.

For the first setting, the online algorithm is allowed to make use of the same error ε as the offline algorithm, which results in a competitiveness of $\mathcal{O}(n^2 \log \Delta)$ (assuming reasonable values of ε or simply assuming to be constant). Intuitively speaking, in this scenario the online algorithm has to solve two questions at the same time: The bounds of the filter intervals, and the choice of the subset of nodes for the output.

Theorem 9 ([9 SPP]). *Assuming ε is a constant, there is an online algorithm for the approximate Top-k Position Monitoring problem which is $\mathcal{O}(n^2 \cdot \log \Delta)$-competitive.*

This interaction between the two questions leads to a gap between the lower bound and the upper bound stated above. To reduce the power of the adversary but still to consider the problem of choosing a subset of nodes for the output, we consider an augmented version which allows the online algorithm to use an error of 2ε compared to ε in the offline algorithm. The algorithm is $\mathcal{O}(n)$-competitive (again with reasonable assumptions on ε and also on the relation of n and Δ). In this setting with a constant number of filter violations it is possible to argue on the placement of filters and thus the combination of filter placement and the subset of the nodes do not take that much of a role expressed in the following:

Theorem 10 **([9 SPP]).** *Assuming ε is a constant and $\log \Delta = \mathcal{O}(n)$, there is an online algorithm for the approximate Top-k Position Monitoring problem which is $\mathcal{O}(n)$-competitive against an optimal offline algorithm using an error of 2ε compared to the error of ε of the offline algorithm.*

5 (Approximate) Count Distinct Monitoring

In the following section, we consider the Count Distinct Monitoring problem where the server is tasked to count how many different values are observed at the sensors. More specifically, we establish an (ε, δ)-approximation of the number of distinct values d_t at time step t. On a high level, our approximation scheme shows how one can combine both a filter-based approach together with a sampling technique to shrink the required communication. Due to space constraints, we only explain our techniques on a high level. For details we refer to [1 SPP].

The key idea for estimating d_t is to follow a sampling approach on the values (not on the nodes). We create a sample out of all values and use the Existence Protocol (Theorem 1) to identify a representing node for each sampled value, i.e., one node per value of the sample set observing the value. Then, we monitor the identified representing nodes to keep track of d_t over time. For the monitoring, a filter-based approach is utilized, allowing us to compare the communication volume of our protocol to a minimal filter-based one as already done in previous sections.

We are able to achieve an (ε, δ)-approximation that is kept valid for multiple time steps depending on how much the values change in consecutive time steps (parameterized by σ). Using the filter-based approach, our analysis relates to the number of messages exchanged by an optimal filter-based approach (R^*). In total, we arrive at the theorem below.

Theorem 11 **([1 SPP]).** *There is an (ε, δ)-approximation for the Count Distinct Monitoring problem for T time steps that uses $\mathcal{O}\left((\sigma+\delta)\frac{R^* \log n}{d_t}\, T\, \frac{1}{\varepsilon^2}\, \log\frac{1}{\delta}\right)$ messages. Here, the change in the number of nodes observing a fixed value between consecutive time steps is upper bounded by a constant factor $\sigma \leq 1/2$ and R^* is the minimum number of changes of representatives for a given input.*

The bound stated above is comprised of different aspects which are reflected by factors stemming from a sampling approach $\Theta(1/\varepsilon^2 \log 1/\delta)$, the fact that the number of domain changes is bounded $\Theta(\sigma + \delta)$ and the competitiveness of monitoring the representative for one domain ($\mathcal{O}(\log n \cdot R_v^*)$) with respect to R_v^*, the number of representatives to monitor that a value v was observed used by an optimal offline algorithm.

The bound of the algorithm can also be expressed as $\mathcal{O}(\log n \cdot R_S^* \cdot 1/\varepsilon^2 \log 1/\delta)$ where R_S^* denotes the optimal number of representatives for the sample set S throughout the time period T. Furthermore, focusing on the aspect of dynamic algorithms, the bound can also be expressed as $\mathcal{O}((\sigma + \delta) \cdot T \cdot 1/\varepsilon^2 \log 1/\delta)$. Note that these bounds are different bounds for the same algorithm and only reflect different input sequences more properly.

5.1 Computation for One Time Step

The computation for one time step takes place in two phases. First, a constant factor approximation for d_t is created. In the second phase, the constant approximation is used to determine a sufficiently large probability that is broadcasted to the sensors, which in turn create a sample out of all observed values that is reported to the server. Based on the size of the sample set and the previously calculated probability, the server can estimate d_t up to a factor of ε with a probability of at least $1 - \delta$.

It is crucial here that we do a random experiment for a value, i.e., all sensors observing the same value should see the same outcome of the random experiment. This can be achieved by a *public coin* [7]. A public coin is a random string consisting of fully unbiased bits that is common for all sensor nodes. It can be implemented by having the same pseudorandom number generator at each sensor initialized by a common seed that is broadcasted by the server at the beginning of each phase of the algorithm. Note that such an approach only increases the communication complexity by an additive constant. A set of sensors (observing the same value) is able to do a random experiment together by considering the same substring of the public coin (which is predefined by the value the sensors are observing).

For a constant factor approximation we first let the sensors draw a random number with the public coin based on a geometric distribution, i.e., we generate a random height h_v for each value v. Then the server triggers a communication of the values of largest height by polling the heights from largest to smallest in synchronous rounds. Thereby, for each value that is communicated, the Existence Protocol is used (cf. Theorem 1) to bring down the number of communicated messages to a constant.

After we have a constant factor approximation, we calculate a probability p which is broadcasted to the sensors. With probability p a value is communicated to the server in the second phase. Whether or not a value is communicated is again decided for all sensors observing the value using the public coin. For each of the values selected in this phase, the Existence Protocol is used again (cf. Theorem 1) to identify a representing sensor. Such a representing sensor witnesses that the sampled value is observed. p is chosen with respect to ε,

δ and the constant factor approximation such that the server can compute an (ε, δ)-approximation based on the number of received values of the second phase.

In the end, most of the communication happens due to the chosen probability to have a sufficiently large sample of the observed value. Thus, we arrive at the theorem below.

Theorem 12 ([1 SPP]). *There is an (ε, δ)-approximation algorithm for the Count Distinct Monitoring problem for one time step using $\mathcal{O}\left(1/\varepsilon^2 \log 1/\delta\right)$ messages on expectation.*

5.2 Monitoring over Multiple Time Steps

In the worst case values might change arbitrarily between multiple consecutive time steps and a sensor that was used as a representative for a value might not be of use even after a single round. However, as argued before, one expects based on practical scenarios that the values that are observed at a fixed sensor are similar in consecutive time steps. To analyze the quality of our algorithm with regard to the significance of changes in consecutive time steps, we use a filter-based approach. The idea is to reuse the results of the (relatively) costly computation of one time step for consecutive time steps as long as the values are similar to a certain degree. The filter is implemented by the representing sensors that are identified, i.e., we compare how many times our protocol has to identify such a representing sensor compared to how many times an optimal filter-based algorithm has to do this (R^*).

Recapitulate that using a public coin, a sample set of values was determined. The server keeps track of the sample after an initial (ε, δ)-approximation is done. Thereby, any sensor sends a message to the server if it observes a value in the sample that has not been observed previously. The server estimates based on such messages how many values are in total newly observed. Similarly, if a representative for a value in the sample stops observing the latter, a new representative is searched (using the Existence Protocol, cf. Theorem 1) and if none is found, the server estimates how many values left in total. Since any filter-based algorithm has to communicate at some point when an optimal representative sensor stops observing its value, our result depends on the minimum possible number of such changes R^* as it can be seen in Theorem 11.

6 Conclusion

In this work we elaborated on models for dynamic input sequences and designed and analyzed algorithms which handle these settings. The respective bounds reflect this by comparing the communication to an optimal filter-based algorithm or by introducing parameters expressing how 'fast' an instance changes from time step to time step. We have also shown that there is an algorithm which combines these two techniques properly.

As a next step it would be interesting to see how these techniques perform in the presence of sliding windows. The fact that sensors are not capable of storing the entire history of the data stream has an influence on the output quality or the number of messages the sensors need to send to the server, although these values might not be relevant for the current time step.

Another aspect on the input streams might have a significant impact on communication bounds: Assuming the streams have a structured property, e.g., be provided by some random process and thus might be assumed to generate similar observations in consecutive rounds at a respective sensor node. With such an assumption in mind we assume to get bounds in return which reflect the communication complexity to be proportional in the ability of projecting future observations based on past observations.

References

1 SPP. Bemmann, P., et al.: Monitoring of domain-related problems in distributed data streams. In: Das, S., Tixeuil, S. (eds.) SIROCCO 2017. LNCS, vol. 10641, pp. 212–226. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72050-0_13
2. Borodin, A., El-Yaniv, R.: Online Computation and Competitive Analysis. Cambridge University Press, Cambridge (1998)
3. Cormode, G.: The continuous distributed monitoring model. SIGMOD Rec. **42**(1), 5–14 (2013). https://doi.org/10.1145/2481528.2481530
4 SPP. Feldkord, B., Malatyali, M., Meyer auf der Heide, F.: A dynamic distributed data structure for top-k and k-select queries. In: Böckenhauer, H.-J., Komm, D., Unger, W. (eds.) Adventures Between Lower Bounds and Higher Altitudes. LNCS, vol. 11011, pp. 311–329. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98355-4_18
5. Jain, A., Chang, E.Y., Wang, Y.: Adaptive stream resource management using kalman filters. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, 13–18 June 2004, pp. 11–22 (2004). https://doi.org/10.1145/1007568.1007573
6. Karlin, A.R., Manasse, M.S., Rudolph, L., Sleator, D.D.: Competitive snoopy caching. Algorithmica **3**(1), 77–119 (1988). https://doi.org/10.1007/BF01762111
7. Kremer, I., Nisan, N., Ron, D.: On randomized one-round communication complexity. Comput. Complex. **8**(1), 21–49 (1999). https://doi.org/10.1007/s000370050018
8 SPP. Mäcker, A., Malatyali, M., Meyer auf der Heide, F.: Online top-k-position monitoring of distributed data streams. In: IPDPS, pp. 357–364. IEEE Computer Society (2015). https://doi.org/10.1109/IPDPS.2015.40
9 SPP. Mäcker, A., Malatyali, M., Meyer auf der Heide, F.: On competitive algorithms for approximations of top-k-position monitoring of distributed streams. In: IPDPS, pp. 700–709. IEEE Computer Society (2016). https://doi.org/10.1109/IPDPS.2016.91
10 SPP. Malatyali, M.: Big data: sublinear algorithms for distributed data streams. Ph.D. thesis, University of Paderborn, Germany (2019)

11. Osipov, V.: Algorithm Engineering for fundamental Sorting and Graph Problems. Ph.D. thesis, Karlsruhe Institute of Technology (2014). http://digbib.ubka.uni-karlsruhe.de/volltexte/1000042377
12. Osipov, V., Sanders, P., Singler, J.: The filter-kruskal minimum spanning tree algorithm. In: Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments, ALENEX 2009, New York, New York, USA, 3 January 2009, pp. 52–61 (2009). https://doi.org/10.1137/1.9781611972894.5
13. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Commun. ACM **28**(2), 202–208 (1985). https://doi.org/10.1145/2786.2793

Energy-Efficient Scheduling

Susanne Albers(✉)

Technische Universität München, Munich, Germany
albers@in.tum.de

Abstract. We review algorithmic techniques for energy conservation in processing environments handling big data sets. Firstly, we address dynamic speed scaling, where processors can run at variable speed/frequency. The goal is to use the speed spectrum of the processors so as to minimize energy consumption while providing a desired service. Here we focus on multi-processor platforms with heterogeneous CPUs. Secondly, we examine power-down mechanisms where idle devices can be transitioned into low-power standby and sleep states. We consider power-down mechanisms in massively parallel systems, where the components have to coordinate their active and idle periods. In particular we focus on data centers with homogeneous as well as heterogeneous servers.

Keywords: Approximation algorithm · Competitive analysis · Dynamic speed scaling · Homogeneous processors · Online algorithm · Polynomial-time algorithm · Power-down mechanisms · Power-heterogeneous processors

1 Introduction

The processing of big data sets crucially depends on powerful hardware environments. Big data is typically processed in data and computing centers. Nonetheless, today even a single PC can solve problems with data volumes that were considered huge just a few years ago. In addition to speed, energy consumption has become a major concern in computing environments. Information and communications technology (ICT) systems consume a significant amount of energy. Currently personal computers, data centers and communication networks use 5–9% of the total electricity worldwide [10,31,34]. It is anticipated that electricity used by ICT could exceed 20% of the global total by 2030. Data centers consume about 200 terawatt hours per year, which corresponds to 1.5% of the global electricity demand [10,34]. This is more than the energy consumption of many (European) countries.

At the heart powerful hardware environments consist of processing units such as servers, PCs and – at the bottom level – CPUs. They may operate separately and sequentially but in most cases form parallel and, in particular, massively parallel systems. Nowadays standard PCs and laptops are equipped with multicore architectures. Moreover, in computing and data centers the available processors are interconnected so that hundreds or thousands of CPUs can work on the same application.

In this chapter we will review algorithmic techniques for energy savings in hardware and, in particular, processor systems. The study of such approaches has received quite some interest over the past 15 years, see e.g., [3, 14, 21, 32] and references therein. Essentially, there exists two general techniques towards an energy conservation in processor systems.

(1) *Dynamic speed scaling*: Many modern microprocessors can run at variable speed of frequency. Examples are the Intel Speed Step and the AMD Power Now! processors as well as the VIA Technologies LongHaul CPUs and the AsAP 1 chips. The speed changes are implemented at the hardware level and the operating system level. High processor speed implies high performance. However, the higher the speed the higher the energy consumption is. The goal is to use the full speed/frequency spectrum of a processor so as to minimize the overall energy consumption, while providing a certain service.

(2) *Power-down mechanisms*: A well-known technique for energy savings is to transition a given system – such as the display of a desktop, a laptop, or simply a CPU – into a standby or hibernate mode if it has been idle for a while. The design of power-down strategies becomes particularly challenging in multi-processor environments, where the active and idle periods of the components have to be coordinated so that the system can satisfy a desired processing demand.

In dynamic speed scaling, energy is conserved by optimally exploiting the speed spectrum of processors. Power-down mechanisms reduce energy consumption by transitioning idle systems into low-power sleep states. In the following sections we address both of the above techniques, focusing on results that were achieved within our project of the SPP 1736.

2 Dynamic Speed Scaling

Dynamic speed scaling has been studied extensively in the algorithms community. Prior work has considered single-processor environments as well as multi-processor platforms with homogeneous CPUs. In this context a fundamental algorithmic optimization problem was introduced in a seminal paper by Yao, Demers and Shenker [39]. Specifically, we are given a single variable-speed processor. If the processor runs at speed s, then the required power is (proportional to) $f(s) = s^\alpha$, where $\alpha > 1$ is a constant. In practice, α is typically a small value in the range $[2, 3]$. In fact the cube-root rule for CMOS devices states that the speed s of a processor is proportional to the cube-root of the power or, equivalently, that power is proportional to s^3. Obviously, when considering a time horizon, energy consumption is power integrated over time.

Yao et al. [39] define a deadline-based scheduling problem. We are given a sequence $\sigma = J_1, \dots, J_n$ of jobs, where each job J_j is specified by a release time r_j, a deadline d_j and a work volume w_j. If a job J_j is processed at fixed speed s, then it takes w_j/s time units to complete the job. Preemption of jobs is allowed. The goal is to find a feasible schedule, respecting the deadline constraints, that minimizes the total energy consumption. For simplicity it is assumed that a processor can run at any speed. In particular, there are no upper and lower bounds on the speeds. Also speed changes are instant. Yao et al. [39] prove that the offline variant of the problem, where all jobs are known in advance, is polynomially solvable.

In the online variant of the problem, the jobs are revealed at their release time. At any time a scheduling algorithm has to make a decision without knowledge of any future jobs. Given a job sequence σ, let $A(\sigma)$ denote the energy consumed by A on σ and let $OPT(\sigma)$ be the minimum energy consumption required for σ. Online algorithm A is called c-competitive [38] if there exists a constant d such that $A(\sigma) \leq c \cdot OPT(\sigma) + d$ holds for every job sequence σ [38]. The constant d must be independent of σ. We remark that, for the results presented in this article, the stated competitive ratios hold without an additive constant. Yao et al. [39] devised two elegant online algorithms, called *Average Rate* and *Optimal Available*. They showed that *Average Rate* achieves a competitive ratio of $\alpha^\alpha 2^{\alpha-1}$, for any $\alpha \geq 2$. Bansal et al. [21] analyzed *Optimal Available* and proved a competitive ratio of α^α.

Speed scaling on homogeneous parallel processors, considering again deadline-based scheduling, was studied in [6, 12, 23]. It is assumed that job migration is allowed, i.e. whenever a job is preempted, it may be moved to a different processor. Hence, over time, a job may be executed on various processors as long as the respective processing intervals do not overlap. Albers et al. [6] show that the offline problem can be solved optimally in polynomial time using a combinatorial algorithm. Furthermore they extend the algorithm *Optimal Available* and prove a competitiveness of α^α. An extension of *Average Rate* attains a competitive ratio of $\alpha^\alpha 2^{\alpha-1} + 1$.

2.1 Speed Scaling on Heterogeneous Processors

In [7 SPP, 8 SPP] we present a comprehensive study of dynamic speed scaling in heterogeneous multi-processor environments. This is a very timely problem as data and computing centers typically host a variety of hardware architectures. Prior to our work, Bampis et al. [18] examined a setting where the power functions of all the processors are convex. For the offline problem they devise an algorithm that returns a solution within an additive ε of the optimum and runs in time polynomial in the size of the instance and $1/\varepsilon$. Gupta et al. [29, 30] study speed scaling on heterogeneous platforms with the objective to minimize energy and the total flow time of jobs.

In [7 SPP, 8 SPP] we focus again on classical deadline-based scheduling and assume that m power-heterogeneous processors $P_1, \ldots, P_m$ are given. Let $f_p(s)$, $1 \leq p \leq m$, be the power function of processor P_p, depending on speed s. We consider two classes of functions.

1. *General power functions*: The function $f_p(s)$ of each processor P_p is an arbitrary continuous and monotonically increasing function of s.
2. *Standard power functions*: Each processor P_p has a power function of the form $f_p(s) = s^{\alpha_p}$, where $\alpha_p > 1$ is a constant. Let $\alpha = \max_{1 \leq p \leq m} \alpha_p$.

We assume that job preemption and migration is allowed. In the following let $t_1 < t_2 < \ldots < t_l < t_{l+1}$ be the sorted sequence of all possible different release times and deadlines of jobs. Let $I_i = [t_i, t_{i+1})$, for $i = 1, \ldots, l$.

2.2 The Offline Problem with General Power Functions

In a first step we develop an algorithm for the offline problem that is based on linear programming and applies to a wide family of continuous power functions. Our linear program (LP) formulation is more compact than the configuration LP proposed in [18]. The latter one contains an exponential number of variables and requires the use of the Ellipsoid method, which may not be very efficient in practice. Moreover, the formulation in [18] is solvable only for convex functions.

In order to define our LP, let s_{LB} and s_{UB} be a lower bound and an upper bound on the speed of any processor in an optimal schedule. We could choose $s_{LB} = w_{\min}/\sum_i |I_i|$ and $s_{UB} = \sum_j w_j / \min_i |I_i|$. Given any constant $\varepsilon > 0$, we geometrically discretize the interval $[s_{LB}, s_{UB}]$ and define the set of discrete speeds

$$D = \{s_{LB}, s_{LB}(1+\varepsilon), s_{LB}(1+\varepsilon)^2, \dots, s_{LB}(1+\varepsilon)^k\},$$

where $k = \min\{i \mid s_{LB}(1+\varepsilon)^i \geq s_{UB}\}$. This set contains $O(\frac{1}{\varepsilon}\log(\frac{s_{UB}}{S_{LB}}))$ speed levels.

We consider the wide class of continuous power functions satisfying the following invariant. For any small constant $\varepsilon > 0$, there exists a small value $\varepsilon' > 0$ such that $f((1+\varepsilon)s) \leq (1+\varepsilon')f(s)$ holds for any speed $s \in [s_{LB}, s_{UB}]$. Intuitively, a small increase in the speed does not increase the power function by too much. In the case of standard power functions we have that $\varepsilon' = (1+\varepsilon)^\alpha - 1$. Hence ε' may depend on ε and the power function; it is not necessarily smaller than 1. We first show that there exists a $(1+\varepsilon')$-approximate schedule such that, at any time, every processor uses a speed level that belongs to D.

For the definition of our LP, for each interval I_i and each job J_j such that $I_i \subseteq [r_j, d_j)$, we introduce a variable $x_{i,j,p,s}$, which corresponds to the total amount of time that J_j is processed during I_i on processor P_p using speed s.

$$\begin{aligned}
\min \quad & \sum_{i,j,p,s} x_{i,j,p,s} f_p(s) \\
\text{s.t.} \quad & \sum_{i,p,s} x_{i,j,p,s}\, s \geq w_j \quad \forall j \\
& \sum_{p,s} x_{i,j,p,s} \leq |I_i| \quad \forall i,j \\
& \sum_{j,s} x_{i,j,p,s} \leq |I_i| \quad \forall i,p \\
& x_{i,j,p,s} \geq 0 \quad \forall i,j,p,s
\end{aligned}$$

A solution to the above LP specifies an operation of job J_j on processor P_p with processing time $\sum_s x_{i,j,p,s}$ during interval I_i. Hence, for each I_i, we obtain an instance of the preemptive open shop problem, which can be solved in polynomial time using the algorithm by Gonzalez and Sahni [28].

Theorem 1. *There exists an algorithm that produces a $(1+\varepsilon')$-approximate schedule in $O(poly(n,m,\frac{1}{\varepsilon},\log(\frac{s_{UB}}{S_{LB}}))$ time.*

2.3 The Offline Problem with Standard Power Functions

In this section we focus on standard power functions $f_p(s) = s^{\alpha_p}$, $1 \leq p \leq m$. Such functions were considered by Yao et al. [39]. In fact, most of the literature on dynamic speed scaling focuses on this family of functions. As a main result in [7 SPP, 8 SPP] we prove that the offline problem can be solved in polynomial time using a fully combinatorial algorithm that is based on repeated maximum flow computations. In a first step we show that there exists an optimal schedule that exhibits four specific properties. These properties will be essential in the design of our algorithm.

First we demonstrate that for any job J_j, $1 \leq j \leq n$, the processor speeds at which the job is executed are related through the derivative of the power functions. More specifically, if J_j is partially executed by processors P_p and P_q with speeds $s_{j,p}$ and $s_{j,q}$, respectively, then $f'_p(s_{j,p}) = f'_q(s_{j,q})$. This follows from the convexity of the power functions when analyzing the energy consumed by J_j on processors P_p and P_q. Therefore, for any job J_j, let $Q_j = f'_p(s_{j,p})$ be the *hypopower* on processor P_p.

Property 1: Each job J_j is executed with constant hypopower Q_j.

The next property implies that, at any time, the available jobs with the greatest hypopower are executed.

Property 2: For any pair of jobs J_j, J_k and $t \in [r_j, d_j) \cap [r_k, d_k)$ such that J_j is executed at time t and J_k is not executed at t, it holds that $Q_j \geq Q_k$.

We assume that the density $\delta_j := w_j/(d_j - r_j)$ of each job J_j satisfies $\delta_j \geq \max_{p,q}(\alpha_p/\alpha_q)^{1/(\alpha_q - 1)}$. Observe that δ_j is equal to the minimum average speed necessary to complete J_j if no other jobs were present. With the assumption on the job densities we can then show that in an optimal schedule, for each job J_j and processor P_p, the speed $s_{j,p}$ is at least 1. This allows us to define an order on the processors. We number the processors $P_1, \ldots, P_m$ such that, for any $s \geq 1$, it holds that $f_1(s) \leq \ldots \leq f_m(s)$. This implies, $\alpha_1 \leq \ldots \leq \alpha_m$ and $f'_1(s) \leq \ldots \leq f'_m(s)$. We say that P_p is cheaper than P_q if $p < q$. The next property states that cheap processors execute, in general, jobs with greater hypopower, compared to expensive processors.

Property 3: Let I be an interval and J_j, J_k be any pair of jobs executed by processors P_p and P_q during I, respectively. If $p < q$, then $Q_j \geq Q_k$.

The final property states that at each time the cheapest processors are occupied.

Property 4: For each interval I_i, there exists an m_i with $0 \leq m_i \leq m$ such that $P_1, \ldots, P_{m_i}$ are occupied throughout I_i while $P_{m_i+1}, \ldots, P_m$ are idle.

We proceed with the description of our algorithm. To this end we define problem instances specified by triples $(\mathbb{J}, \mathbb{P}, \mathbb{I})$. Here $\mathbb{J}$ is a set of jobs, $\mathbb{P}$ is a set of processors and $\mathbb{I}$ is a set of disjoint intervals. Initially, $\mathbb{J} = \{J_1, \ldots, J_n\}$, $\mathbb{P} = \{P_1, \ldots, P_m\}$ and $\mathbb{I} = \{I_1, \ldots, I_l\}$. In general, during each $I_i \in \mathbb{I}$, there is a subset $\mathbb{J}(I_i) \subseteq \mathbb{J}$ of *alive* jobs J_j with $I_i \subseteq [r_j, d_j)$ and a subset $\mathbb{P}(I_i) \subseteq \mathbb{P}$ of available processors that are unused throughout I_i. Let $n_i = |\mathbb{J}(I_i)|$ and $a_i = |\mathbb{P}(I_i)|$

Let S^* be an optimal schedule satisfying Properties 1–4. Consider any interval $I_i \in \mathbb{I}$. In Property 4, considering S^*, we have $m_i = \min\{n_i, a_i\}$ because the number of used processors cannot exceed the number of available processors or the number of alive jobs. This equation specifies the exact amount of time, say t_p, that a processor $P_p \in \mathbb{P}$ is used in S^* as well as the corresponding intervals. The most energy-efficient though not necessarily feasible way to schedule the jobs in $\mathbb{J}$ is to use the same constant hypopower Q satisfying

$$\sum_{p\in\mathbb{P}} t_p \left(\frac{Q}{\alpha_p}\right)^{\frac{1}{\alpha_p-1}} = \sum_{J_j\in\mathbb{J}} w_j.$$

We assume for simplicity that the value of Q satisfying the above equation can be computed with arbitrary precision.

If there is a feasible schedule in which all jobs are executed with constant hypopower Q, then this schedule is optimal and we are done. As we will explain below, this feasibility problem and the calculation of the corresponding schedule can be solved using a maximum flow computation. If such a feasible schedule does not exist, then $(\mathbb{J},\mathbb{P},\mathbb{I})$ can be partitioned into two independent subproblems $(\mathbb{J}_{\geq Q},\mathbb{P}_{\geq Q},\mathbb{I})$ and $(\mathbb{J}_{<Q},\mathbb{P}_{<Q},\mathbb{I})$. Here $\mathbb{J}_{\geq Q}$ and $\mathbb{J}_{<Q}$ are the subsets of $\mathbb{J}$ that are executed with hypopower at least Q and smaller Q, respectively, in the optimal schedule S^*. In each interval $I_i \in \mathbb{I}$, Properties 2 and 3 specify the subsets of available processors $\mathbb{P}_{\geq Q}(I_i), \mathbb{P}_{<Q}(I_i) \subseteq \mathbb{P}$ dedicated to the jobs of $\mathbb{J}_{\geq Q}$ and $\mathbb{J}_{<Q}$ that are alive during I_i. The jobs of $\mathbb{J}_{\geq Q}$ occupy the cheapest $\min\{a_i, |\mathbb{J}_{\geq Q}(I_i)|\}$ processors during I_i, while the jobs of $\mathbb{J}_{<Q}$ use the remaining processors of $\mathbb{P}(I_i)$.

The feasibility of $(\mathbb{J},\mathbb{P},\mathbb{I})$ w.r.t. the hypopower Q is based on a maximum flow computation in an appropriate network $N(\mathbb{J},\mathbb{P},\mathbb{I},Q)$. Consider an interval $I_i \in \mathbb{I}$ and a processor $P_p \in \mathbb{P}(I_i)$. If P_p runs with hypopower Q in I_i, then its speed is $s_{i,p} = (Q/\alpha_p)^{1/(\alpha_p-1)}$. We slightly abuse notation and let $s_{i,p}$ be the speed of the p-th cheapest available processor during I_i and $\mathbb{P}(I_i)$ be the set of the the m_i cheapest available processors during I_i.

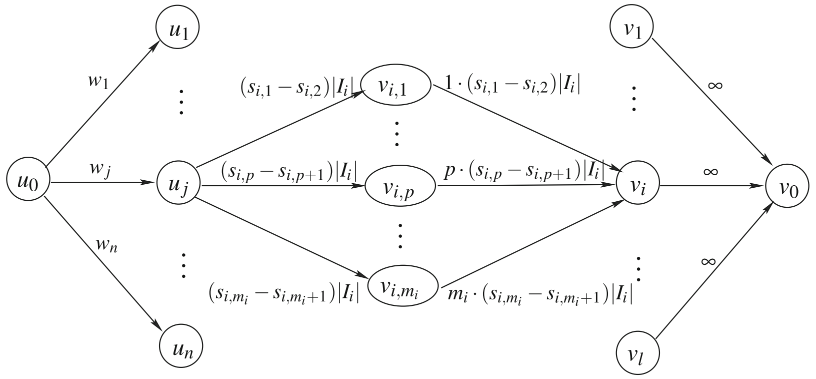
In the network, there is a source node u_0, a node u_j for each $J_j \in \mathbb{J}$, a node $v_{i,p}$ for each pair of interval $I_i \in \mathbb{I}$ and processor $P_p \in \mathbb{P}(I_i)$, a node v_i for each interval $I_i \in \mathbb{I}$, and a sink node v_0. The network contains the arc (u_0,u_j) with capacity w_j for each job $J_j \in \mathbb{J}$, the arc $(u_j,v_{i,p})$ with capacity $(s_{i,p}-s_{i,p+1})|I_i|$ for each interval I_i, job $J_j \in \mathbb{J}(I_i)$ and processor $P_p \in \mathbb{P}(I_i)$, the arc $(v_{i,p},v_i)$ with capacity $p(s_{i,p}-s_{i,p+1})|I_i|$ for each interval $I_i \in \mathbb{I}$ and processor $P_p \in \mathbb{P}(I_i)$ as well as the arc (v_i,v_0) with infinite capacity for each $I_i \in \mathbb{I}$. We set $s_{i,m+1} := 0$. This is depicted in Fig. 1, was also introduced by Federgruen and Groenevelt [25].

If there does not exist a feasible schedule for $(\mathbb{J},\mathbb{P},\mathbb{I})$ with hypopower Q, then the biseparation into $(\mathbb{J}_{\geq Q},\mathbb{P}_{\geq Q},\mathbb{I})$ and $(\mathbb{J}_{<Q},\mathbb{P}_{<Q},\mathbb{I})$ is based on the following crucial property. Let $\mathbb{J}' \subseteq \mathbb{J}_{<Q}$ be any subset of jobs. A job $J_j \in \mathbb{J}\setminus\mathbb{J}'$ belongs to $\mathbb{J}_{\geq Q}$ if and only if, in the network $N(\mathbb{J}\setminus\mathbb{J}',\mathbb{P},\mathbb{I},Q)$, there exists a minimum (u_0,v_0)-cut that does not contain arc (u_0,u_j). This allows us to identify $\mathbb{J}_{\geq Q}$ and $\mathbb{J}_{<Q}$. The technical details are omitted here. In summary Algorithm 1 show a pseudocode description of our strategy. The following theorem gives the main result.

Theorem 2. *Algorithm 1 generates an optimal schedule and runs in polynomial time* $O(n^4 m)$.

2.4 An Online Algorithm

The online algorithm *Average Rate (AVR)*, proposed by Yao et al. [39] for single-processor speed scaling with power function $f(s) = s^\alpha$, works with the concept of job



u_1
w_1
u_0
w_j
u_j
w_n
u_n
$(s_{i,1}-s_{i,2})|I_i|$
$v_{i,1}$
$1\cdot(s_{i,1}-s_{i,2})|I_i|$
$(s_{i,p}-s_{i,p+1})|I_i|$
$v_{i,p}$
$p\cdot(s_{i,p}-s_{i,p+1})|I_i|$
$(s_{i,m_i}-s_{i,m_i+1})|I_i|$
v_{i,m_i}
$m_i\cdot(s_{i,m_i}-s_{i,m_i+1})|I_i|$
v_i
v_1
v_l
v_0
∞
∞
∞

Fig. 1. The flow network

Algorithm 1: $\text{OPT}(\mathbb{J},\mathbb{P},\mathbb{I})$

```
1 Compute the optimum hypopower $Q$ for executing $(\mathbb{J},\mathbb{P},\mathbb{I})$;
2 $(\mathbb{J}_{\geq Q},\mathbb{P}_{\geq Q},\mathbb{I})$, $(\mathbb{J}_{<Q},\mathbb{P}_{<Q},\mathbb{I}) \leftarrow$ BISEPARATION$(\mathbb{J},\mathbb{P},\mathbb{I},Q)$;
3 if $\mathbb{J}=\mathbb{J}_{\geq Q}$ then
4     return CONSTANTHYPOPOWERSCHEDULE$(\mathbb{J},\mathbb{P},\mathbb{I},Q)$;
5 else
6     $S_{\geq Q} \leftarrow$ OPT$(\mathbb{J}_{\geq Q},\mathbb{P}_{\geq Q},\mathbb{I})$;
7     $S_{<Q} \leftarrow$ OPT$(\mathbb{J}_{<Q},\mathbb{P}_{<Q},\mathbb{I})$;
8 return $S_{\geq Q}\cup S_{<Q}$;
```

densities. Again, the density δ_j of job J_j is equal to $\delta_j = w_j/(d_j - r_j)$. Recall that this is the minimum average speed necessary to complete the job if no other jobs were present. At any time t, the processor speed $s(t)$ is set to the accumulated density of active jobs, i.e. $s(t) = \sum_{j:t\in[r_j,d_j)} \delta_j$. With this speed profile, available jobs are scheduled according to the Earliest Deadline First policy.

In order to generalize *AVR* to the multi-processor setting, we consider a variation of the above single-processor algorithm, which uses the same processor speed at any time but applies a different job selection rule. Assume w.l.o.g. that all release times and deadlines are integers. Moreover, assume that $r_{\min} = \min_{1\leq j\leq n} r_j = 0$ and $d_{\max} = \max_{1\leq j\leq n} d_j = T$. We partition the time horizon into unit-length intervals $I_t = [t, t+1)$, $0 \leq i < T$. For each job J_j with $I_t \subseteq [r_j, d_j)$, the algorithm assigns a work volume of δ_j to interval I_t. Then it produces an arbitrary schedule of the total work assigned to I_t using a fixed speed of $s(t) = \sum_{j:I_t\subseteq[r_j,d_j)} \delta_j$ during the whole I_t. This modified algorithm attains the same competitive ratio as the original algorithm *AVR* because both strategies always employ the same speed and consume the same energy.

Next we turn our attention to the setting with multiple heterogeneous processors. Based on the above algorithm variation, we say that a schedule S is an *AVR-schedule* if, for every job J_j and interval $I_t \subseteq [r_j, d_j)$, the total amount of work of J_j executed during I_t on all the processors in S is equal to δ_j. We prove that, for each input sequence

$\sigma = J_1, \ldots, J_n$, there exists a feasible AVR-schedule S_{AVR} on heterogeneous processors with general power functions, as described in Sect. 2.2, whose energy consumption is at most $\max_p c_p + 1$ times that of the optimum schedule for σ. Here c_p is the competitive ratio of the single-processor *AVR* algorithm when executed on processor P_p with power function $f_p(s)$.

We are ready to describe our algorithm *H-AVR* for heterogeneous processors. The main idea is to generate a $(1+\varepsilon)$-approximate AVR-schedule using the LP-algorithm described in Sect. 2.2. More specifically, given the assignment of work into intervals implied by the definition of AVR-schedules, for each interval $I_t = [t, t+1)$ we compute an offline $(1+\varepsilon)$-approximate schedule for this subinstance of the heterogeneous speed-scaling problem.

Theorem 3. *H-AVR is* $(1+\varepsilon)(\max_p c_p + 1)$*-competitive for speed scaling with heterogeneous processors, where* c_p *is the competitiveness of the single-processor AVR algorithm when applied to processor* P_p *with general power function* $f_p(s)$.

Corollary 1. *H-AVR is* $(1+\varepsilon)(\alpha^\alpha 2^{\alpha-1} + 1)$*-competitive for speed scaling with heterogeneous processors having standard power functions.*

2.5 Further Results

We briefly review work by postdoctoral scientists when they were funded within our project. Article [19] explores dynamic speed scaling, assuming that job preemptions are not allowed. In some applications it might not be feasible or too expensive to interrupt and later resume the execution of a job. For the setting with a single processor, we develop a polynomial time algorithm achieving an improved approximation guarantee of $(1+\varepsilon)^\alpha B_\alpha$, where B_α is a generalization of the Bell number [19]. For multi-processor environments we develop the first approximation algorithm for the fully power-heterogeneous setting, where each processor P_p has an individual power function $f_p(s) = s^{\alpha_p}$. The performance factor is equal to $B_\alpha((1+\varepsilon)(1+w_{\max}/w_{\min}))^\alpha$. Here $w_{\max}$ and $w_{\min}$ are the maximum and minimum work volumes of the jobs. Again $\alpha = \max_{1 \le p \le m} \alpha_p$.

In [11] we examine the scenario where jobs must be executed subject to an energy budget. The goal is to maximize the throughput. As a main result we develop polynomial time algorithms based on dynamic programming. In [26] we introduce the new problem of scheduling jobs over scenarios. In [27] we study a dynamic market scheduling problem where an intermediary interacts with an unknown sequence of agents.

3 Power-Down Mechanisms in Data Centers

Power-down strategies for a single device have been investigated by Irani et al. [33] and Augustine et al. [17]. The goal is to minimize the energy consumed in an idle period when the device is not in use. In our work we focus on power-down mechanisms in massively parallel systems and, in particular, data centers.

Energy management is a key issue in data center operations [24]. Electricity costs are a dominant and rapidly growing expense in such centers; about 30–50% of their budget is invested into energy. Surprisingly, the servers of a data center are only utilized 20–40% of the time on average [16,22]. When idle and in active mode, they consume about half of their peak power. Hence a fruitful approach for energy conservation and capacity management is to transition idle servers into standby and sleep states. Servers have a number of low-power states [1]. However state transitions, and in particular power-up operations, incur energy/cost. Therefore, dynamically matching the varying demand for computing capacity with the number of active servers is a challenging problem.

3.1 Heterogeneous Servers

In [4 SPP,5 SPP] we formulate and study an optimization problem that arises in the energy management of data centers, hosting a large number of heterogeneous servers. Each server has an active state and several standby/sleep states with individual power consumption rates. The demand for computing capacity varies over time. Idle servers may be transitioned to low-power modes so as to rightsize the pool of active servers. The goal is to find a state transition schedule for the servers that minimizes the total energy consumed. On a small scale the same problem arises in multi-core architectures with heterogeneous processors on a chip. One has to determine active and idle periods for the cores so as to minimize the consumed energy.

More formally, we define the optimization problem *Dynamic Power Management (DPM)*. A problem instance $I = (\mathbb{S}, \mathbb{D})$ is specified by a set of servers and varying computing demands over a time horizon. Let $\mathbb{S} = \{S_1, \ldots, S_m\}$ be a set of *heterogeneous servers*. Each server S_i, $1 \le i \le m$, has an active state as well as one or several standby/sleep states. The states of S_i are denoted by $s_{i,0}, \ldots, s_{i,\sigma_i}$. Here $s_{i,0}$ is the active state and $s_{i,1}, \ldots, s_{i,\sigma_i}$ are the low-power states. The modes have individual power consumption rates. Let $r_{i,j}$ be the power consumption rate of $s_{i,j}$, i.e., $r_{i,j}$ energy units are consumed per time unit while S_i resides in $s_{i,j}$. The states are numbered in order of decreasing rates such that $r_{i,0} > \ldots > r_{i,\sigma_i} \ge 0$. A server can transition between its states. Let $\Delta_{i,j,j'}$ be the non-negative energy needed to move S_i from state $s_{i,j}$ to state $s_{i,j'}$, for any pair $0 \le j, j' \le \sigma_i$. The transition energies satisfy the triangle inequality, i.e., the energy to move directly from $s_{i,j}$ to $s_{i,j'}$ is upper bounded by that of visiting an intermediate state $s_{i,k}$. Formally, $\Delta_{i,j,j'} \le \Delta_{i,j,k} + \Delta_{i,k,j'}$.

Over a time horizon the computing demands are given by a *demand profile* $\mathbb{D} = (T, D)$. Tuple $T = (t_1, \ldots, t_n)$ contains the points in time when the computing demands change. There holds $t_1 < t_2 < \ldots < t_n$ so that the time horizon is $[t_1, t_n)$. Tuple $D = (d_1, \ldots, d_{n-1})$ specifies the demands. More precisely, $d_k \in \mathbb{N}_0$ servers are required for computing during interval $[t_k, t_{k+1})$, for any $1 \le k \le n-1$. Thus at least d_k servers must reside in the active state during $[t_k, t_{k+1})$. We have $d_k \le m$, for any $1 \le k \le n-1$, so that the requirements can be met.

Given $I = (\mathbb{S}, \mathbb{D})$, a *schedule* Σ specifies, for each S_i and any $t \in [t_1, t_n)$, in which state server S_i resides at time t. Schedule Σ is *feasible* if during any interval $[t_k, t_{k+1})$ at least d_k servers are in the active state, $1 \le k \le n-1$. The energy $E(\Sigma)$ incurred by Σ is the total energy consumed by all the m servers. Whenever server S_i, $1 \le i \le m$, resides in state $s_{i,j}$ it consumes energy at a rate of $r_{i,j}$. Whenever the server transitions from state $s_{i,j}$ to state $s_{i,j'}$, the incurred energy is $\Delta_{i,j,j'}$. The goal is to find an *optimal schedule*, i.e., a feasible schedule Σ that minimizes $E(\Sigma)$. We assume that initially, immediately before t_1, and at time t_n all servers reside in the deepest sleep state, i.e. S_i is in s_{i,σ_i}, $1 \le i \le m$.

In DPM the demand for computing capacity is specified by the number of servers needed at any time. In data centers it is common practice that a number of required servers is determined as a function of the current total workload, ignoring specific jobs. DPM focuses on energy conservation instead of individual job placement. Again, in the active state, a processor has a fixed energy consumption rate. We investigate DPM as an offline problem, i.e. the varying computing demands are known in advance. From an algorithmic point of view it is important to explore the tractability and approximability of the problem. The offline setting is also relevant in practice. Data centers usually analyze past workload traces to identify long-term patterns. The findings are used to specify demands in future time windows.

Given a problem instance I, we first characterize optimal solutions. Property 1 below implies that there exists an optimal schedule in which a server never changes state while being in low-power mode. Property 2 states that there exists an optimal schedule executing state transitions only when the computing demands change. A server *powers up* if it transitions from a low-power state to the active state (indexed 0). A server *powers down* if it moves from the active state to a low-power state.

Property 1: There exists an optimal schedule with the following property. Suppose that S_i powers down at time t and next powers up at time t'. Then between t and t' S_i resides in a single state $s_{i,j}$, where $j > 0$.

Property 2: There exists an optimal schedule that satisfies Property 1 and performs state transitions only at the times of T.

Finally we may assume w.l.o.g. that the power-down energies $\Delta_{i,0,j}$ are equal to 0, $1 \le i \le m$ and $1 \le j \le \sigma_i$. If this is not the case we case we can simply fold the power-down energy $\Delta_{i,0,j} > 0$ into the corresponding power-up energy $\Delta_{i,j,0}$.

3.2 Servers with Two States

In [4 SPP, 5 SPP] we first investigate the variant of DPM in which each server S_i has exactly two states, an active state $s_{i,0}$ and a sleep state $s_{i,1}$, $1 \le i \le m$. As a main result we show that an optimal schedule can be computed in polynomial time using an algorithm that resorts to a min-cost flow computation.

In a first step we argue that we may assume w.l.o.g. that the power consumption rates in the sleep states are equal to 0. If this is not the case and $r_{i,1} > 0$, for some i, then we can subtract $r_{i,1}$ from both $r_{i,0}$ and $r_{i,1}$. This changes the energy consumption by a fixed amount of $r_{i,1}(t_n - t_1)$ over the entire time horizon. To simplify notation let $r_i := r_{i,0}$ be the power consumption rate of S_i in the active state, $1 \le i \le m$. Moreover, let $\Delta_i := \Delta_{i,1,0}$ be the energy needed to transition S_i from the sleep state to the active state.

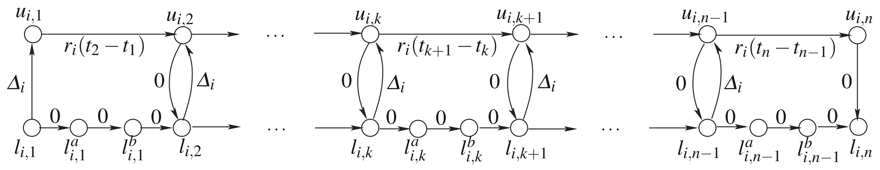


Fig. 2. The component C_i for server S_i

In the following let $I = (\mathbb{S}, \mathbb{D})$ be a given problem instance. We develop an algorithm that computes an optimal schedule. Based on Property 2, we focus on schedules that perform state transitions only at the times of T. Given I, our strategy constructs a flow network $N(I)$ that we describe in the next paragraphs.

Network Components. Network $N(I)$ contains a *component* C_i, for each server S_i, $1 \le i \le m$. Such a component C_i, which is depicted in Fig. 2, consists of an *upper path* and a *lower path*. The upper path represents the active state of S_i; the lower path models the server's sleep state. The computing demands change at the times $t_1 < \ldots < t_n$ in T. For any t_k, $1 \le k \le n$, there is a vertex $u_{i,k}$ on the upper path. Vertices $u_{i,k}$ and $u_{i,k+1}$ are connected by a directed edge $(u_{i,k}, u_{i,k+1})$ of cost $r_i(t_{k+1} - t_k)$, $1 \le k \le n-1$. This cost is equal to the energy consumed if S_i is in the active state during $[t_k, t_{k+1})$. Similarly, for any t_k, $1 \le k \le n$, there is a vertex $l_{i,k}$ on the lower path. In order to ensure that at least d_k servers are in the active state during $[t_k, t_{k+1})$, if $k < n$, we need two auxiliary vertices $l^a_{i,k}$ and $l^b_{i,k}$. These vertices are again connected by directed edges. There is an edge $(l_{i,k}, l^a_{i,k})$, followed by two edges $(l^a_{i,k}, l^b_{i,k})$ and $(l^b_{i,k}, l_{i,k+1})$, for any k with $1 \le k \le n-1$. The cost of each of these edges is 0 because the energy consumption in the sleep state is 0.

The lower and the upper paths are connected by additional edges that model state transitions. Recall that all servers are in the sleep state at times t_1 and t_n. For any k with $1 \le k \le n-1$, there is a directed edge $(l_{i,k}, u_{i,k})$ of cost Δ_i, representing a power-up operation of S_i at time t_k. For any k with $1 < k \le n$, there is a directed edge $(u_{i,k}, l_{i,k})$ of cost 0, modeling a power-down operation of S_i at time t_k. The capacity of each edge of C_i is equal to 1.

The Entire Network. In $N(I)$ components $C_1, \ldots, C_m$ are aligned in parallel and connected to a source a_0 and a sink b_0. The general structure of $N(I)$ is depicted in Fig. 3. There is a directed edge from a_0 to $l_{i,1}$ in C_i, for any $1 \le i \le m$. Furthermore, there is a directed edge from $l_{i,n}$ to b_0, for any $1 \le i \le m$. Each of these edges has a cost of 0 and a capacity of 1. Vertex a_0 has a supply of m, and b_0 has a demand of m. Hence m units of flow must be shipped through $C_1, \ldots, C_m$. Since all edges have a capacity of 1, one unit of flow must be routed through each C_i, $1 \le i \le m$. Whenever the unit traverses the upper path, S_i is in the active state. Whenever the unit traverses the lower path, S_i is in the sleep state.

In order to ensure that at least d_k servers are in the active state during $[t_k, t_{k+1})$, $1 \le k \le n-1$, we introduce additional sources and sinks. Network $N(I)$ has a source a_k and a sink b_k with supply/demand d_k, for any $1 \le k \le n-1$. There is a directed edge from a_k to $l^a_{i,k}$ on the lower path of each C_i, $1 \le i \le m$. Furthermore, there is a directed

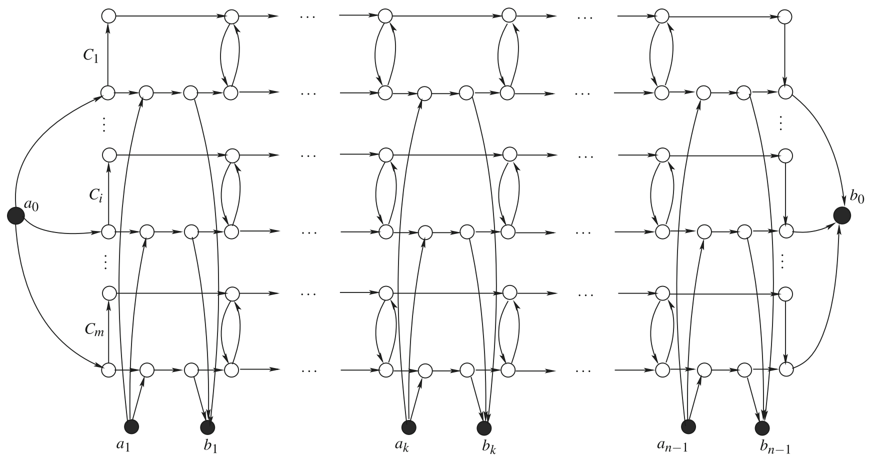


Fig. 3. The network $N(I)$

edge from each $l^b_{i,k}$ to b_k, $1 \leq i \leq m$. The cost and capacity of each of these edges is equal to 0 and 1, respectively. Since d_k flow units have to be shipped from a_k to b_k, there must exist at least d_k components C_i in which the flow unit from a_0 to b_0 traverses the upper path from $u_{i,k}$ to $u_{i,k+1}$. Hence the corresponding servers are in the active state during $[t_k, t_{k+1})$.

Obviously, any feasible schedule Σ in which state transitions are performed only at the times of T corresponds to a feasible flow of cost $E(\Sigma)$ in $N(I)$. Unfortunately, the reverse statement is not true. Since $N(I)$ is a single-commodity flow network, a feasible flow f does not necessarily represent a feasible schedule. It may happen that flow shipped out of a source a_k is not necessarily routed to b_k, $0 \leq k \leq n-1$. In particular, flow leaving a_k may be routed to a sink $b_{k'}$, where $k' > k$, or to b_0. Observe that in $N(I)$ all edge capacities and supplies/demands are integer values. Hence in $N(I)$ there exists a minimum-cost flow that is integral, i.e., the flow along any edge takes an integer value. Moreover, there exist polynomial time combinatorial algorithms that compute such an integral minimum-cost flow [2]. In [5 SPP,4 SPP] we prove that any feasible integral flow f of cost C in $N(I)$ can be transformed so that it corresponds to a feasible schedule Σ consuming energy C. More specifically, using (non-trivial) flow modification operations, we ensure that each network component C_i ships exactly on flow unit in each interval $[t_k, t_{k+1})$. The transformation takes a polynomial number of steps.

Theorem 4. *Let I be an instance of DPM in which each server has exactly two states. An optimal schedule for I can be computed in polynomial time by a combinatorial algorithm that uses a minimum-cost flow computation.*

3.3 Servers with Multiple States

In [4 SPP,5 SPP] we also investigate DPM in the general scenario that each server has multiple sleep states. In this case DPM becomes NP-hard. We extend our approach based on flow computations to design an approximation algorithm. More specifically, we develop a second algorithm that works with a more complex network in which each component has several lower paths, representing the various low-power states of a server. Furthermore, we need a second commodity to ensure that computing demands are met. With only a single commodity, flow units could switch between lower paths at no cost, and infeasible schedules would result.

Given a fractional two-commodity minimum-cost flow, our algorithm executes advanced flow rounding and packing procedures. First, by repeatedly traversing components, the algorithm modifies flow so it becomes integral on the upper paths. Then flow on the lower paths is packed. The final integral flow allows the constructing of a schedule for DPM. Our algorithm achieves an approximation factor of τ, where τ is the number of server types in the problem instance. Specifically, the servers can be partitioned into τ classes such that, within each class, the servers are identical. Of course, the servers of a class are independent and not synchronized. In practice, a data center has a large collection of machines but a relatively small number of different server architectures.

Theorem 5. *Let I be an instance of DPM with τ server types. A schedule whose energy consumption is at most τ times the minimum one for I can be computed in polynomial time based on a min-cost two-commodity flow computation.*

3.4 Homogeneous Servers

In [9] we investigate another algorithmic problem with the objective of dynamically resizing a data center. Specifically, we resort to a framework that was introduced by Lin, Wierman, Andrew and Thereska [35,37].

Consider a data center with m homogeneous servers, each of which has two states, an active state and a sleep state. An optimization is performed over a discrete, finite time horizon consisting of time steps $t = 1,\dots,T$. At any time t, $1 \le t \le T$, a non-negative convex cost function $f_t(\cdot)$ models the operating cost of the data center. More precisely, $f_t(x_t)$ is the incurred cost if x_t servers are in the active state at time t, where $0 \le x_t \le m$. This operating cost captures e.g., the energy cost and service delay, for an incoming workload, depending on the number of active servers.

Furthermore, at any time t there is a switching cost, taking into account that the data center may be resized by changing the number of active servers. This switching cost is equal to $\Delta(x_t - x_{t-1})^+$, where Δ is a positive real constant and $(x)^+ = \max(0,x)$. Again we assume that transition cost is incurred when servers are powered up from the sleep state to the active state. A cost of powering down servers may be folded into this cost. The constant Δ incorporates e.g., the energy needed to transition a server from the sleep state to the active state, as well as delays resulting from a migration of data and connections. We assume that at the beginning and at the end of the time horizon all servers are in the sleep state, i.e., $x_0 = x_{T+1} = 0$. The goal is to determine a vector $X = (x_1,\dots,x_T)$ called *schedule*, specifying at any time the number of active servers, that minimizes

$$\sum_{t=1}^{T} f_t(x_t) + \Delta \sum_{t=1}^{T} (x_t - x_{t-1})^+. \tag{1}$$

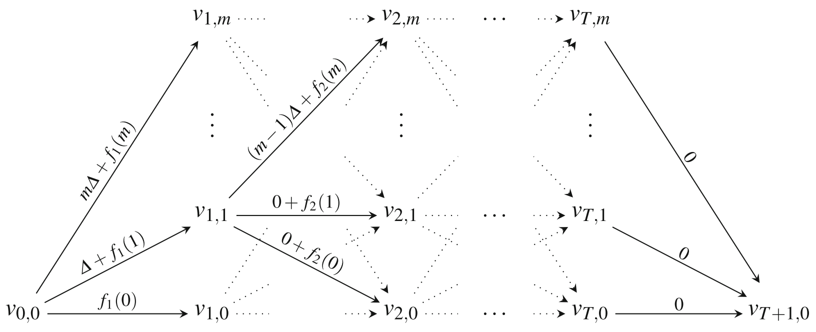


Fig. 4. Construction of the graph

All previous work [13,15,20,35–37] on the data-center optimization problem assumes that the server numbers x_t, $1 \leq t \leq T$, may take fractional values. That is, x_t may be an arbitrary real number in the range $[0,m]$. From a practical point of view this is acceptable because a data center has a large number of machines. Nonetheless, from an algorithmic and optimization perspective, the proposed algorithms do not compute feasible solutions. Important questions remain if the x_t are indeed integer valued: (1) Can optimal solutions be computed in polynomial time? (2) What is the best competitive ratio achievable by online algorithms?

In [9] we present the first study of the above data-center optimization problem assuming that the x_t take integer values. In a first step we examine the offline variant of the problem, where the convex functions f_t, $1 \leq t \leq T$, are known in advance. Lin et al. [37] developed an algorithm based on a convex program that computes optimal solutions if fractional values x_t are allowed.

Considering the discrete setting with integer valued x_t, we prove that optimal solutions can also be computed in polynomial time. Our algorithm is different from the convex optimization approach by Lin et al. [37]. More precisely, our strategy works with an underlying directed, weighted graph $G = (V,E)$. Let $[k] := \{1,2,\dots,k\}$ and $[k]_0 := \{0,1,\dots,k\}$ with $k \in \mathbb{N}$. For each $t \in [T]$ and each $j \in [m]_0$, there is a vertex $v_{t,j}$, representing the state that exactly j servers are active at time t. Furthermore, there are two vertices $v_{0,0}$ and $v_{T+1,0}$ for the initial and final states $x_0 = 0$ and $x_{T+1} = 0$. For each $t \in \{2,\dots,T\}$ and each pair $j,j' \in [m]_0$, there is a directed edge from $v_{t-1,j}$ to $v_{t,j'}$ having weight $\Delta(j'-j)^+ + f_t(j)$. This edge weight corresponds to the switching cost when changing the number of servers between time $t-1$ and t and to the operating cost incurred at time t. Obviously, $\Delta(j'-j)^+ + f_t(j) = f_t(j) + \Delta(j'-j)^+$ so that the edge cost properly represents the cost contribution in the objective function, see (1), at time t. Similarly, for $t = 1$ and each $j' \in [m]_0$, there is a directed edge from $v_{0,0}$ to $v_{1,j'}$ with weight $f_1(j) + \Delta(j')^+$. Finally, for $t = T$ and each $j \in [m]_0$, there is a directed edge from $v_{T,j}$ to $v_{T+1,0}$ of weight 0. The structure of G is depicted in Fig. 4. In the following, for each $j \in [m]_0$, vertex set $R_j = \{v_{t,j} \mid t \in [T]\}$ is called *row j*.

A path between v_0 and v_{T+1} represents a schedule. If the path visits $v_{t,j}$, then $x_t = j$ servers are active at time t. The total length (weight) of a path is equal to the cost of the corresponding schedule. An optimal schedule can be determined using a shortest path computation, which takes $O(Tm)$ time in the particular graph G. However, this running time is not polynomial because the encoding length of an input instance is linear in T and $\log m$, in addition to the encoding of the functions f_t. In [9] we present a polynomial time algorithm that improves an initial schedule iteratively using binary search. In each iteration the algorithm constructs and uses only a constant number of rows of G.

Theorem 6. *An optimal schedule can be computed in polynomial time $O(T \log m)$.*

In [9] we also examine the online variant of the data center optimization problem where the functions f_t, $1 \le t \le T$, are revealed over time. We extend an algorithm *Lazy Capacity Provisioning* proposed by Lin et al. [37] and prove that it achieves a competitive ratio of 3. We also show that this is best possible. No deterministic online algorithm can attain a competitive ratio smaller than 3.

References

1. The Advanced Configuration and Power Interface. The latest specification 6.3 (2019) is available e.g. at UEFI.org
2. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows - Theory, Algorithms and Applications. Prentice Hall (1993)
3. Albers, S.: Energy-efficient algorithms. Commun. ACM **53**(5), 86–96 (2010). https://doi.org/10.1145/1735223.1735245
4 SPP. Albers, S.: On energy conservation in data centers. In: Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, pp. 35–44 (2017). https://doi.org/10.1145/3087556.3087560
5 SPP. Albers, S.: On energy conservation in data centers. ACM Trans. Parallel Comput. **6**(3), 13:1-13:26 (2019). https://doi.org/10.1145/3364210
6. Albers, S., Antoniadis, A., Greiner, G.: On multi-processor speed scaling with migration. J. Comput. Syst. Sci. **81**(7), 1194–1209 (2015). https://doi.org/10.1016/j.jcss.2015.03.001
7 SPP. Albers, S., Bampis, E., Letsios, D., Lucarelli, G., Stotz, R.: Scheduling on power-heterogeneous processors. In: Proceeding of the 12th American Symposium on Theoretical Informatics, LATIN, pp. 41–54 (2016). https://doi.org/10.1007/978-3-662-49529-2_4
8 SPP. Albers, S., Bampis, E., Letsios, D., Lucarelli, G., Stotz, R.: Scheduling on power-heterogeneous processors. Inf. Comput. **257**, 22–33 (2017). https://doi.org/10.1016/j.ic.2017.09.013
9. Albers, S., Quedenfeld, J.: Optimal algorithms for right-sizing data centers. In: Proceedings of the 30th Symposium on Parallelism in Algorithms and Architectures, SPAA, pp. 363–372 (2018). https://doi.org/10.1145/3210377.3210385
10. Andrae, A.S.G., Edler, T.: On global electricity usage of communication technology: trends to 2030. Challenges **6**(1), 117–157 (2015). https://doi.org/10.3390/challe6010117
11. Angel, E., Bampis, E., Chau, V., Letsios, D.: Throughput maximization for speed scaling with agreeable deadlines. J. Sched. **19**(6), 619–625 (2015). https://doi.org/10.1007/s10951-015-0452-y
12. Angel, E., Bampis, E., Kacem, F., Letsios, D.: Speed scaling on parallel processors with migration. J. Comb. Optim. **37**(4), 1266–1282 (2018). https://doi.org/10.1007/s10878-018-0352-0

13. Antoniadis, A., Barcelo, N., Nugent, M., Pruhs, K., Schewior, K., Scquizzato, M.: Chasing convex bodies and functions. In: Kranakis, E., Navarro, G., Chávez, E. (eds.) LATIN 2016. LNCS, vol. 9644, pp. 68–81. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49529-2_6
14. Antoniadis, A., Garg, N., Kumar, G., Kumar, N.: Parallel machine scheduling to minimize energy consumption. In: Proceedings of the 31st ACM-SIAM Symposium on Discrete Algorithms, SODA, pp. 2758–2769 (2020). https://doi.org/10.1137/1.9781611975994.168
15. Antoniadis, A., Schewior, K.: A tight lower bound for online convex optimization with switching costs. In: Solis-Oba, R., Fleischer, R. (eds.) WAOA 2017. LNCS, vol. 10787, pp. 164–175. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89441-6_13
16. Armbrust, M., et al.: A view of cloud computing. Commun. ACM **53**(4), 50–58 (2010). https://doi.org/10.1145/1721654.1721672
17. Augustine, J., Irani, S., Swamy, C.: Optimal power-down strategies. SIAM J. Comput. **37**(5), 1499–1516 (2008). https://doi.org/10.1137/05063787X
18. Bampis, E., Kononov, A., Letsios, D., Lucarelli, G., Sviridenko, M.: Energy-efficient scheduling and routing via randomized rounding. J. Sched. **21**(1), 35–51 (2016). https://doi.org/10.1007/s10951-016-0500-2
19. Bampis, E., Letsios, D., Lucarelli, G.: Speed-scaling with no preemptions. In: Proceeding of the 25th International Symposium on Algorithms and Computation, ISAAC, pp. 259–269 (2014). https://doi.org/10.1007/978-3-319-13075-0_21
20. Bansal, N., Gupta, A., Krishnaswamy, R., Pruhs, K., Schewior, K., Stein, C.: A 2-competitive algorithm for online convex optimization with switching costs. In: Proceedings 18th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, APPROX, pp. 96–109 (2015). https://doi.org/10.4230/LIPIcs.APPROX-RANDOM.2015.96
21. Bansal, N., Kimbrel, T., Pruhs, K.: Speed scaling to manage energy and temperature. J. ACM **54**(1), 3:1-3:39 (2007). https://doi.org/10.1145/1206035.1206038
22. Barroso, L.A., Hölzle, U.: The case for energy-proportional computing. IEEE Comput. **40**(12), 33–37 (2007). https://doi.org/10.1109/MC.2007.443
23. Bingham, B.D., Greenstreet, M.R.: Computation with energy-time trade-offs: models, algorithms and lower-bounds. In: IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA, pp. 143–152 (2008). https://doi.org/10.1109/ISPA.2008.127
24. Dayarathna, M., Wen, Y., Fan, R.: Data center energy consumption modeling: a survey. IEEE Commun. Surv. Tutorials **18**(1), 732–794 (2016). https://doi.org/10.1109/COMST.2015.2481183
25. Federgruen, A., Groenevelt, H.: Preemptive scheduling of uniform machines by ordinary network flow techniques. Manag. Sci. **32**(3), 341–349 (1986). https://doi.org/10.1287/mnsc.32.3.341
26. Feuerstein, E., et al.: Minimizing worst-case and average-case makespan over scenarios. J. Sched. **20**(6), 545–555 (2016). https://doi.org/10.1007/s10951-016-0484-y
27. Giannakopoulos, Y., Koutsoupias, E., Lazos, P.: Online market intermediation. In: Proceedings of the 44th International Colloquium on Automata, Languages, and Programming, ICALP, pp. 47:1–47:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.ICALP.2017.47
28. Gonzalez, T.F., Sahni, S.: Open shop scheduling to minimize finish time. J. ACM **23**(4), 665–679 (1976). https://doi.org/10.1145/321978.321985
29. Gupta, A., Im, S., Krishnaswamy, R., Moseley, B., Pruhs, K.: Scheduling heterogeneous processors isn't as easy as you think. In: Proceedings of the 23rd Annual ACM-SIAM

Symposium on Discrete Algorithms, SODA, Kyoto, Japan, 17–19 January 2012, pp. 1242–1253 (2012). https://doi.org/10.1137/1.9781611973099.98
30. Gupta, A., Krishnaswamy, R., Pruhs, K.: Scalably scheduling power-heterogeneous processors. In: Proceedings of the 37th International Colloquium on Automata, Languages and Programming, ICALP, pp. 312–323 (2010). https://doi.org/10.1007/978-3-642-14165-2_27
31. Heddeghem, W.V., et al.: Trends in worldwide ICT electricity consumption from 2007 to 2012. Comput. Commun. **50**, 64–76 (2014). https://doi.org/10.1016/j.comcom.2014.02.008
32. Irani, S., Pruhs, K.: Algorithmic problems in power management. SIGACT News **36**(2), 63–76 (2005). https://doi.org/10.1145/1067309.1067324
33. Irani, S., Shukla, S.K., Gupta, R.: Algorithms for power savings. ACM Trans. Algorithms **3**(4), 41 (2007). https://doi.org/10.1145/1290672.1290678
34. Jones, N.: How to stop data centres from gobbling up the world's electricity. Nature **561**, 163–166 (2018). https://doi.org/10.1038/d41586-018-06610-y
35. Lin, M., Wierman, A., Andrew, L.L.H., Thereska, E.: Dynamic right-sizing for power-proportional data centers. In: Proceedings 30th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, INFOCOM, pp. 1098–1106 (2011). https://doi.org/10.1109/INFCOM.2011.5934885
36. Lin, M., Wierman, A., Andrew, L.L.H., Thereska, E.: Online dynamic capacity provisioning in data centers. In: Proceedings of the 49th Annual Allerton Conference on Communication, Control, and Computing, Allerton, pp. 1159–1163 (2011). https://doi.org/10.1109/Allerton.2011.6120298
37. Lin, M., Wierman, A., Andrew, L.L.H., Thereska, E.: Dynamic right-sizing for power-proportional data centers. IEEE/ACM Trans. Netw. **21**(5), 1378–1391 (2013). https://doi.org/10.1109/TNET.2012.2226216
38. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Commun. ACM **28**(2), 202–208 (1985). https://doi.org/10.1145/2786.2793
39. Yao, F.F., Demers, A.J., Shenker, S.: A scheduling model for reduced CPU energy. In: Proceedings of the 36th Annual Symposium on Foundations of Computer Science, FOCS, pp. 374–382 (1995). https://doi.org/10.1109/SFCS.1995.492493

The GENO Software Stack

Joachim Giesen(✉), Lars Kuehne, and Sören Laue

Friedrich-Schiller-Universität Jena, Jena, Germany
{joachim.giesen,lars.kuehne,soeren.laue}@uni-jena.de

Abstract. GENO (generic optimization) is a domain specific language for mathematical optimization. The GENO software generates a solver from a specification of an optimization problem class. The optimization problems, that is, their objective function and constraints, are specified in a formal language. The problem specification is then translated into a general normal form. Problems in normal form are then passed on to a general purpose solver. In its Iterations, the solver evaluates expressions for the objective function, constraints, and their derivatives. Hence, computing symbolic gradients of linear algebra expressions is an important component of the GENO software stack. The expressions are evaluated on the available hardware platforms including CPUs and GPUs from different vendors. This becomes possible by compiling the expressions into BLAS (Basic Linear Algebra Subroutines) calls that have been optimized for the different hardware platforms by their vendors. The compiler, called autoBLAS, that translates formal linear algebra expressions into optimized BLAS calls is another important component in the GENO software stack. By putting all the components together the generated solvers are competitive with problem-specific hand-written solvers and orders of magnitude faster than competing approaches that offer comparable ease-of-use. While this article describes the full GENO software stack, its components are of also of interest on their own and thus have been made available independently.

Keywords: Constrained optimization · Tensor calculus · BLAS

1 Introduction

GENO makes state-of-the-art performance in solving optimization problems easily accessible. Since optimization problems are ubiquitous in science, engineering and economics, it is not surprising that they come in many different flavors. Traditionally, a main distinction is made between discrete and continuous optimization problems. The focus of GENO is on the continuous case. Prominent examples for classes of continuous optimization problems are linear programs (LPs), quadratic programs (QPs), second-order cone programs (SOCPs), and semi-definite programs (SDPs). For these classes, efficient algorithms and well engineered implementations (solvers) exist for many years. The solvers are typically called from a programming environment. The optimization problems' data are passed to the solver through function calls. It is the responsibility of the programmer to provide the data in the right format, that is compliant to a standard form for the specific problem class. The burden of reformulating the problems in standard form is alleviated by modeling languages that transform a problem specification into standard form. Popular modeling languages are CVX [12,17] for MATLAB and its Python extension CVXPY [2,13], Pyomo [19,20] for Python, and JuMP [14] which is bound to Julia. These languages take an instance of an optimization problem and transform it into some standard form of an LP, QP, SOCP, or SDP, respectively. The transformed problem is then passed to a solver that expects the standard form. However, the transformation can be computationally inefficient, because the representation in standard form can be large in terms of the problem size. Also, the solver is called from within the programming environment only for the given problem instance. The modeling language plus solver approach has been made deployable in the CVXGEN [31], QPgen [16], and OSQP [5] projects. In these projects code is generated for the specified problem class and not just for one problem instance. However, the problem dimensions need to be fixed and the generated code is optimized only for very small or sparse problems. There also exist implementations of the modeling language plus solver approach that are independent from a specific programming environment. Prominent examples are AMPL [15] and GAMS [8] that are popular in the operations research community.

GENO differs from previous work by a much tighter coupling of the language and the solver. GENO does not transform problem instances but whole problem classes, including constrained problems, into a very general standard form. Since the standard form is independent of any specific problem instance it does not grow for larger instances. Hence, the generated solvers can be used like hand-written solvers. They even reach or surpass the efficiency of hand-written solvers for large dense problems. Typically, they are orders of magnitude faster than state-of-the-art modeling language plus solver approaches.

In this article, that is based on the original publications [24,25,28,29], we describe the full GENO software stack. The tight coupling of modeling language and solver is achieved in GENO by computing symbolic gradients that are evaluated by the solver on the given data of the optimization problem. Hence, an important part of GENO's software stack is a facility for computing derivatives of linear algebra expressions. GENO's modeling language allows to specify whole classes of optimization problems in terms of the objective function and constraints that are given as vectorized linear algebra expressions. Neither the objective function nor the constraints need to be differentiable. Non-differentiable problems are transformed into constrained, differentiable problems. A general purpose solver for constrained, differentiable problems is then instantiated with the objective function, the constraint functions and their respective gradients. Using vectorized linear algebra allows a direct mapping onto optimized implementations of BLAS (Basic Linear Algebra Subroutines) routines. BLAS and its close relative LAPACK [3] are the de facto standard for the language independent high performance evaluation of linear algebra expressions. Almost all major hardware vendors provide individual BLAS implementations for their particular hardware, including CPUs (AMD Blis [43], Intel MKL [10], Arm Performance Libraries [30]) and GPUs (NVIDIA cuBLAS [11], AMD clBLAS [4]). GENO supports different hardware platforms through the autoBLAS precompiler that translates linear algebra expressions into optimized BLAS library calls for the addressed hardware.

The GENO software stack comprises a modeling language (Sect. 2), a generic solver (Sect. 3), a matrix and tensor calculus (Sect. 4), and an automatic mapping to BLAS (Sect. 5). The latter three components of GENO's software stack are of interest in a broader context than GENO and hence have been made available independently. GENO is available at www.geno-project.org [27].

2 Modeling Language

GENO's modeling language uses a MATLAB-like syntax for specifying optimization problems. MATLAB is a platform for numerical computations using matrices. The advantages of using matrix expressions are two-fold: First, it allows the user to phrase an optimization problem without the need of specifying the number of variables nor the number of constraints. Hence, the generated solver is not tied to a specific instance but can handle arbitrary-sized problems. Second, it enables direct mappings to BLAS routines that are much more efficient than the corresponding for-loops.

A specification in GENO has four blocks:

1. Declaration of the problem parameters that can be of type *Matrix*, *Vector*, or *Scalar*,
2. declaration of the optimization variables that also can be of type *Matrix*, *Vector*, or *Scalar*,
3. specification of the objective function, and finally
4. specification of the constraints, also in a MATLAB-like syntax that supports the following operators and functions: `+`, `-`, `*`, `/`, `.*`, `./`, ^, .^, ′, `log`, `exp`, `sin`, `cos`, `tanh`, `abs`, `norm1`, `norm2`, `sum`, `tr`, `det`, `inv`.

See Fig. 1 for some illustrative examples.

```
parameters
  Matrix A symmetric
variables
  Vector x
min
  -x'*A*x / (x'*x)
```

```
parameters
  Matrix A
  Vector b
variables
  Vector x
min
  norm2(A*x - b)^2
st
  x >= 0
```

```
parameters
  Matrix A
  Vector b
variables
  Vector x
min
  norm1(x)
st
  A*x == b
  sum(x) == 1
  x >= 0
```

Fig. 1. A few optimization problems formulated in the GENO modeling language. The problem on the left is an unconstrained optimization problem that computes the Rayleigh quotient, the problem in the middle is the non-negative least squares problem, and the problem on the right shows an ℓ_1-norm minimization problem from the domain of compressed sensing over the unit simplex.

GENO's modeling language also allows the specification of non-smooth optimization problems, for instance, problems that employ the `norm1` function, that is, the non-smooth ℓ_1-norm. The non-smooth optimization problems that are allowed by GENO can be written as $\min_x\{\max_i f_i(x)\}$ with smooth functions $f_i(x)$ [36], which is a fairly flexible class that accommodates many of the commonly-encountered non-smooth objective functions. All problems within this class can be transformed into constrained, smooth problems of the form

$$\min_{t,x} t \quad \text{s.t.} \quad f_i(x) \le t \ \forall i.$$

The transformed problems are then solved by a solver for constrained, smooth optimization problems. Hence, within the GENO software stack only a solver for constrained, smooth optimization problems is needed. In the next section we describe the solver that is implemented in the GENO software stack.

3 Generic Optimizer

GENO's generic optimizer employs a solver for unconstrained, smooth optimization problems. This solver is then extended to handle also constraints. The choice for the solver that is implemented within the GENO software stack is motivated by applications in machine learning. Optimization problems in machine learning typically exhibit a few dozen up to a few million variables, and the involved data matrices do not have any special structure and are typically not sparse, that is, at least 10% of the entries are non-zero entries. These properties exclude second-order optimization algorithms and justify our choice to implement a slightly modified version of the L-BFGS-B algorithm [9,44] that can handle smooth optimization problems that have no general constraints, except possibly bound constraints on the variables. It provides a good trade-off between the number of iterations and the complexity per iteration. It also does not assume any structure on the problem data and it is numerically quite robust. On quadratic problems it shares the same convergence guarantees [22,34] as Nesterov's optimal gradient descent method [35] but compared to Nesterov's method it is parameter free, i.e., no parameters need to be tuned or known for the specific problem.

3.1 Solver for Bound-Constrained Smooth Problems

The solver for bound-constrained, smooth optimization problems combines a standard limited memory quasi-Newton method with a projected gradient path approach. In each iteration, the gradient path is projected onto the box constraints and the quadratic function based on the second-order approximation (L-BFGS) of the Hessian is minimized along this path. All variables that are at their boundaries are fixed and only the remaining free variables are optimized using the second-order approximation. Any solution that is not within the bound constraints is projected back onto the feasible set by a simple min/max operation [32]. Only in rare cases, a projected point does not form a descent direction. In this case, instead of using the projected point, one picks the best point that is still feasible along the ray towards the solution of the quadratic approximation. Then, a line search is performed for satisfying the strong Wolfe conditions [41,42].

This ensures convergence also in the non-convex case. The line search also removes the need for a predefined step length parameter. We use the line search proposed in [33] which we enhance by a backtracking line search in case the solver enters a region where the function is not defined.

3.2 Solver for Constrained Smooth Problems

There are quite a few options for solving smooth, constrained optimization problems. We decided to use the augmented Lagrangian approach [21,38]. It allows to (re-)use our solver for smooth, unconstrained problems, it is fairly robust, and does not need to tune any parameters. The augmented Lagrangian method can be used for solving the following general standard form of an abstract constrained optimization problem

$$\min_x f(x) \quad \text{s.t.} \quad h(x) = 0 \text{ and } g(x) \leq 0, \tag{1}$$

where $x \in \mathbb{R}^n$, $f\colon \mathbb{R}^n \to \mathbb{R}$, $h\colon \mathbb{R}^n \to \mathbb{R}^m$, $g\colon \mathbb{R}^n \to \mathbb{R}^p$ are differentiable functions, and the equality and inequality constraints are understood component-wise.

The augmented Lagrangian of Problem (1) is the following function

$$L_\rho(x,\lambda,\mu) = f(x) + \frac{\rho}{2}\left\| h(x) + \frac{\lambda}{\rho}\right\|^2 + \frac{\rho}{2}\left\|\left(g(x) + \frac{\mu}{\rho}\right)_+\right\|^2,$$

where $\lambda \in \mathbb{R}^m$ and $\mu \in \mathbb{R}^p_{\geq 0}$ are the Lagrange multipliers, also known as dual variables, $\rho > 0$ is a constant, $\|\cdot\|$ denotes the Euclidean norm, and $(v)_+$ denotes $\max\{v, 0\}$. The augmented Lagrangian is the standard Lagrangian of Problem (1) augmented by a quadratic penalty term. The quadratic term provides increased stability during the optimization process which can be seen, for example, in the case that Problem (1) is a linear program.

The Augmented Lagrangian Algorithm 1 runs in iterations. Upon convergence, it will return an approximate solution x to the original problem along with an approximate solution of the Lagrange multipliers for the dual problem. If Problem (1) is convex, then the algorithm returns the global optimal solution. Otherwise, it returns a local optimum [6]. The update of the multiplier ρ can be ignored and the algorithm still converges [6]. However, in practice it is beneficial to increase it depending on the progress in satisfying the constraints [7]. If the infinity norm of the constraint violation decreases by a factor less than $\tau = 1/2$ in one iteration, then ρ is multiplied by a factor of two.

4 Matrix and Tensor Calculus

The solver at the core of GENO's generic optimizer, an implementation of the L-BFGS-B algorithm for bound-constrained smooth problems, runs in iterations. In each iteration expressions for the objective function and its gradients are evaluated. Within GENO these expressions, especially the gradients, have to be made available to the solver. Expressions for objective functions are given in GENO's modeling language that uses a vectorized notation, that is, a notation that avoids explicit indices.

Algorithm 1. Augmented Lagrangian Method

```
Input  : instance of Problem (1)
Output: approximate solution $x \in \mathbb{R}^n, \lambda \in \mathbb{R}^p, \mu \in \mathbb{R}^m_{\geq 0}$
1 initialize $x^0 = 0$, $\lambda^0 = 0$, $\mu^0 = 0$, and $\rho = 1$
2 repeat
3     $x^{k+1} := \text{argmin}_x\, L_\rho(x, \lambda^k, \mu^k)$
4     $\lambda^{k+1} := \lambda^k + \rho h(x^{k+1})$
5     $\mu^{k+1} := \left(\mu^k + \rho g(x^{k+1})\right)_+$
6     update $\rho$
7 until convergence
8 return $x^k, \lambda^k, \mu^k$
```

The advantage of a vectorized notation is that expressions can be mapped more or less directly to BLAS calls and thus to highly optimized BLAS implementations. For GENO we also want this advantage for the gradients. Hence, we need to compute derivatives of matrix expressions. Although computing derivatives of matrix and tensor expressions is a fundamental and frequent task, surprisingly, no algorithm existed that would solve this problem in the general case. In the following, we describe our approach [24,28] that for the first time allowed to compute derivatives of general tensor expressions. It was shown in [24] that evaluating derivatives of non-scalar valued functions computed by this approach is two orders of magnitude faster than previous state-of-the-art approaches when evaluated on the CPU and up to three orders of magnitude faster when evaluated on the GPU. An implementation of our approach is integrated into the GENO software stack. It is also available as a standalone tool at www.MatrixCalculus.org [26].

4.1 Problems with Matrix Notation

Computing derivatives for scalar functions, i.e., $f(x)\colon \mathbb{R} \to \mathbb{R}$ is a straightforward task and is taught already in high school. One just applies the chain rule repeatedly and the partial derivatives are multiplied together. For instance, consider the function $f(x) = sin(x^2)$. Its derivative is $f'(x) = cos(x^2) \cdot 2 \cdot x$. However, this no longer works in the matrix and tensor case. Compared to the scalar case where only one type of multiplication operator exists, there are several types of multiplication in the matrix and tensor case. It has been shown that 24 types of different multiplications are necessary for representing the derivatives of matrix expressions only in the linear case [37]. Hence, it is essential to find a good representation of matrix and tensor multiplications.

Furthermore, when computing derivatives of vector and matrix expressions, even matrix notation is not sufficient to express all derivatives. For instance, for function $f(x)\colon \mathbb{R}^n \to \mathbb{R}^m$, the derivative will be a matrix $M \in \mathbb{R}^{m\times n}$. But already its second derivative will be $T \in \mathbb{R}^{m\times n\times n}$, i.e., a third order tensor, which cannot be represented in standard matrix notation. One usually circumvents this by using the vec-operator that maps a matrix to a vector by stacking its columns on top of each other and using the Kronecker product. This way, one can flatten some dimensions and emulate higher order tensors and their multiplications. However, still not all necessary multiplications can be represented this way and it unnecessarily complicates the representation. And even in the two-dimensional case, i.e., when the derivative is a tensor of order two, it might have no corresponding representation as a matrix. For instance, consider the simple quadratic function $f(x) = x^\top Ax$, where $x \in \mathbb{R}^n$ is a vector and $A \in \mathbb{R}^{n\times n}$ is a matrix. When computing the derivative of f with respect to x using the chain rule, one has to compute the derivative of $x^\top$ with respect to x, i.e., the derivative of a function that maps x to its transpose. This is not the identity matrix. In fact, it is not even representable as a matrix. In the more powerful Ricci notation [39] it would be written as the tensor δ_{ij}. Hence, the right representation of tensors and operators on them, especially multiplications between them is crucial. In fact, choosing the *right* representation has led to the first general and coherent matrix and tensor calculus theory [24]. Before, only a number of cases could be treated systematically. While the first theory used Ricci notation to represent tensors and their multiplications it turned out that using a generalized form of Einstein notation makes the process of computing derivatives even simpler and more coherent [28].

4.2 Einstein Notation

In tensor calculus one can distinguish three types of multiplication, namely inner, outer, and element-wise multiplication. Indices are used for distinguishing between these types. For tensors A, B, and C any multiplication of A and B can be written as

$$C[s_3] = \sum_{(s_1\cup s_2)\setminus s_3} A[s_1]\cdot B[s_2], \tag{2}$$

where C is the result tensor and s_1, s_2, and s_3 are the index sets of the left argument, the right argument, and the result tensor, respectively. The summation is over all indices that appear in at least one of the two multiplication's arguments A and B and are not present in the result tensor C. The index set of the result tensor is always a subset of the union of the index sets of the multiplication's arguments, that is, $s_3 \subseteq (s_1 \cup s_2)$. In the following we denote the generic tensor multiplication as defined in Eq. (2) simply as

$$C = A *_{(s_1,s_2,s_3)} B.$$

This notation is basically identical to the tensor multiplication `einsum` in NumPy, TensorFlow, and PyTorch, and to the notation used in the Tensor Comprehension Package [40].

Note, that the $*_{(s_1,s_2,s_3)}$-notation comes close to standard Einstein notation. In Einstein notation the index set s_3 of the output is omitted and the convention is to sum over all shared indices in s_1 and s_2. However, this restricts the types of multiplications that can be represented. The set of multiplications that can be represented in standard Einstein notation is a proper subset of the multiplications that can be represented by our notation. For instance, standard Einstein notation is not capable of representing element-wise multiplications directly. Still, in the following we refer to the $*_{(s_1,s_2,s_3)}$-notation simply as Einstein notation as it is standard practice in many linear algebra packages.

4.3 Tensor Calculus

In the following, let $\|A\| = \sqrt{\sum_s A[s]^2}$ denote the norm of a tensor A. For vectors it coincides with the Euclidean norm and for matrices with the Frobenius norm. The following definition generalizes the standard derivative to the multi-dimensional case.

Definition 1 (Fréchet Derivative). *Let $f\colon \mathbb{R}^{n_1 \times n_2 \times \ldots \times n_k} \to \mathbb{R}^{m_1 \times m_2 \times \ldots \times m_l}$ be a function that takes an order-k tensor as input and maps it to an order-l tensor as output. Then, $D \in \mathbb{R}^{m_1 \times m_2 \times \ldots \times m_l \times n_1 \times n_2 \times \ldots \times n_k}$ is called the derivative of f at x if and only if*

$$\lim_{h \to 0} \frac{\|f(x+h) - f(x) - D \circ h\|}{\|h\|} = 0,$$

where $\circ$ is an inner tensor product.

Here, the dot product notation $D \circ h$ is short for the inner product $D *_{(s_1 s_2, s_2, s_1)} h$, where $s_1 s_2$ is the index set of D and s_2 is the index set of h. For instance, if $D \in \mathbb{R}^{m_1 \times n_1 \times n_2}$ and $h \in \mathbb{R}^{n_1 \times n_2}$, then $s_1 = \{i, j, k\}$ and $s_2 = \{j, k\}$.

With this definition at hand, we can compute derivatives of matrix and tensor expressions in Einstein notation. As noted in the beginning of this section, derivatives are usually computed using the chain rule. There are two major orderings in which we can apply the chain rule; in a forward fashion and in a reverse fashion. These ways are known as forward mode and reverse mode in the area of algorithmic differentiation (AD, aka. automatic differentiation) [18]. They will both result in the same derivative but not necessarily in the same expression for the derivative. The forward mode coincides with what is usually taught in high school and commonly refers to as symbolic computation of derivatives [23]. Here, we will only describe the reverse mode since this is the mode that is used within the GENO software stack.

Any expression can be represented as a directed acyclic expression graph (expression DAG). Figure 2 shows the expression DAG for the objective function of the logistic regression, i.e.,

$$1^\top (y \odot \log(\exp(Xw) + 1)), \tag{3}$$

where $\odot$ denotes the element-wise multiplication.

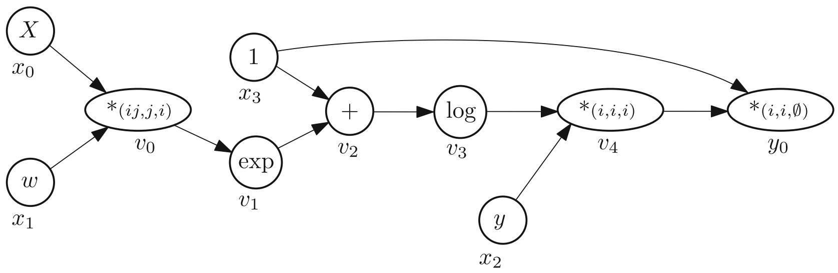


Fig. 2. Expression DAG for Expression (3).

The nodes of the DAG that have no incoming edges represent the variables or constants of the expression and are referred to as input nodes. The nodes of the DAG that have no outgoing edges represent the functions that the DAG computes and are referred to as output nodes. Let the DAG have n input nodes (variables) and m output nodes (functions). Note, that the DAG in Fig. 2 has only one output node. We label the input nodes as $x_0,...,x_{n-1}$, the output nodes as $y_0,...,y_{m-1}$, and the internal nodes as $v_0,\ldots,v_{k-1}$. Every internal and every output node represents an operator whose arguments are supplied by the incoming edges.

When evaluating the DAG, i.e., computing the function values that the DAG represents for some given input, one proceeds from the input nodes to the output nodes. In forward mode automatic differentiation one proceeds in the same direction for computing the derivative and in reverse mode in reverse order, i.e., from output to input nodes. Each node v_i will eventually store the derivative $\frac{\partial y_j}{\partial v_i}$ which is usually denoted as $\bar{v}_i$, where y_j is the function to be differentiated. This partial derivative is often referred to as adjoint. These derivatives are computed as follows: First, the derivatives $\frac{\partial y_j}{\partial y_i}$ are stored at the output nodes of the DAG. Then, the derivatives that are stored at the remaining nodes, here called z, are iteratively computed by summing over all their outgoing edges as follows

$$\bar{z} = \frac{\partial y_j}{\partial z} = \sum_{f:(z,f)\in E} \frac{\partial y_j}{\partial f} \cdot \frac{\partial f}{\partial z} = \sum_{f:(z,f)\in E} \bar{f} \cdot \frac{\partial f}{\partial z}, \tag{4}$$

where the multiplication is again tensorial. The following theorems specify the type of tensor multiplication for reverse mode Eq. (4). Their proofs can be found in [29].

Theorem 1. *Let Y be an output node with index set s_4 and let $C = A *_{(s_1,s_2,s_3)} B$ be a multiplication node of the expression DAG. Then the contribution of C to the adjoint $\bar{B}$ of B is $\bar{C} *_{(s_4s_3,s_1,s_4s_2)} A$ and its contribution to the adjoint $\bar{A}$ of A is $\bar{C} *_{(s_4s_3,s_2,s_4s_1)} B$.*

If the output function Y in Theorem 1 is scalar-valued, then we have $s_4 = \emptyset$ and the adjoint coincides with the function implemented in all modern deep learning frameworks including TensorFlow and PyTorch. Hence, our approach can be seen as a direct generalization of the scalar case.

Theorem 2. *Let Y be an output function with index set s_3, let f be a general unary function whose domain has index set s_1 and whose range has index set s_2, let A be a node in the expression DAG, and let $C = f(A)$. The contribution of the node C to the adjoint $\bar{A}$ is*

$$\bar{f} *_{(s_3s_2,s_2s_1,s_3s_1)} f'(A),$$

where f' is the derivative of f.

In case that the general unary function is simply an elementwise unary function that is applied element-wise to a tensor, Theorem 2 simplifies as follows.

Theorem 3. *Let Y be an output function with index set s_2, let f be an elementwise unary function, let A be a node in the expression DAG with index set s_1, and let $C = f(A)$ where f is applied element-wise. The contribution of the node C to the adjoint $\bar{A}$ is*

$$\bar{f} *_{(s_2s_1,s_1,s_2s_1)} f'(A),$$

where f' is the derivative of f.

Table 1 shows the individual steps of the reverse mode applied to the expression graph in Fig. 2. Note, that reverse mode manages to compute the derivative of the output function with respect to all input variables in one pass. Again, the last column shows the derivatives in matrix notation when a few simplifications have been applied, like removal of zero and identity tensors. From the first two rows we can read off the derivative of f with respect to X and the derivative with respect to w. The values of the intermediate results and common subexpressions v_1 and v_2 can be substituted again to obtain the final expression $X^\top \cdot (y \odot \exp(Xw) \oslash \exp(Xw+1))$. This expression can then be mapped very easily to a NumPy expression. In the next section, we will discuss how to map such expressions also to different hard- and software backends.

Table 1. Individual steps of the reverse mode automatic differentiation of the logistic regression function, i.e., $1^\top(y \odot \log(\exp(Xw)+1))$ with respect to all input variables.

Forward trace			Reverse derivative trace				
$\downarrow$ x_0	$=$	X	$\uparrow$ $\bar{x}_0$	$=$	$\bar{v}_0 *_{(i,j,ij)} x_1$	$=$	$(y \oslash v_2 \odot v_1) \cdot w^\top$
x_1	$=$	w	$\bar{x}_1$	$=$	$\bar{v}_0 *_{(i,ij,j)} x_0$	$=$	$X^\top \cdot (y \oslash v_2 \odot v_1)$
x_2	$=$	y	$\bar{x}_2$	$=$	$\bar{v}_4 *_{(i,i,i)} v_3$	$=$	v_3
x_3	$=$	1	$\bar{x}_3$	$=$	$\bar{v}_2 + \bar{y}_0 *_{(\emptyset,i,i)} v_4$	$=$	$y \oslash v_2 + v_4$
v_0	$=$	$x_0 *_{(ij,j,i)} x_1$	$\bar{v}_0$	$=$	$\bar{v}_1 *_{(i,i,i)} v_1$	$=$	$y \oslash v_2 \odot v_1$
v_1	$=$	$\exp(v_0)$	$\bar{v}_1$	$=$	$\bar{v}_2$	$=$	$y \oslash v_2$
v_2	$=$	$v_1 + x_3$	$\bar{v}_2$	$=$	$\bar{v}_3 *_{(i,i,i)} v_2^{-1}$	$=$	$y \oslash v_2$
v_3	$=$	$\log(v_2)$	$\bar{v}_3$	$=$	$\bar{v}_4 *_{(i,i,i)} x_2$	$=$	y
v_4	$=$	$v_3 *_{(i,i,i)} x_2$	$\bar{v}_4$	$=$	$\bar{y}_0 *_{(\emptyset,i,i)} x_3$	$=$	1
y_0	$=$	$v_4 *_{(i,i,\emptyset)} x_3$	$\bar{y}_0$	$=$	1		

5 autoBLAS

GENO aims at providing state-of-the-art performance on a wide variety of backends that include multicore CPUs and GPUs. Hence, it is necessary to generate efficient code for all these backends. This is the purpose of autoBLAS. GENO does not need to directly compile the specification of an optimization problem into executable code but it can map it to an intermediate representation where linear algebra expressions are given as blocks of autoBLAS code. The autoBLAS precompiler then compiles the intermediate code into standard code for the specified backends. autoBLAS itself features an intuitive syntax for linear algebra expressions that is easy to read and comprehend, and delegates the details about their execution to highly-efficient implementations of BLAS routines for the respective backends like for instance AMD Blis [43], Intel MKL [10], Arm Performance Libraries [30], NVIDIA cuBLAS [11], and AMD clBLAS [4].

5.1 A Simple autoBLAS Example

For illustrating autoBLAS, we discuss a minimal example, namely, a matrix-vector product. Listing 1.1 shows a snippet of C++ code initializing a set of *std::vectors* that represent vectors and matrices, followed by a pragma-style declaration of an autoBLAS section. The autoBLAS section first declares two vectors *x* and *y*, and a matrix *A*. Each declaration comes with a set of name-value pairs, like `data=x.data()` or `rows=rows`, that describe required properties for generating code to evaluate expressions of the associated variables. The set of supported names and restrictions on values lies in the responsibility of the selected host-language-*context*. Here, the C-language context has been chosen by setting `c=c`. The currently supported contexts are the C-language, the Eigen library, the NumPy library, and cuBLAS (CUDA).

Listing 1.1. Example embedding autoBLAS within C++

```
int rows = 10;
int col  = 20;
std::vector<double> x(rows);
std::vector<double> A(rows * cols);
std::vector<double> y(cols);
// init y, A, and x with application specific values
#autoblas c=c {
  Vector x data=x.data();
  Vector y data=y.data();
  Matrix A data=A.data()
           rows=rows
           cols=cols;
  y = A' * x;
}
// continue using x, y, and A in C++
```

The code in Listing 1.1 is, of course, no valid C++ code and cannot be compiled directly with a standard C++ compiler. In order to get host-language code for the expressions that are stated in embedded autoBLAS sections, the autoBLAS precompiler has to be called first. Listing 1.2 shows how to invoke the autoBLAS precompiler on a file *example.c.in*. In this simple example, the *-b* flag is set to select the target routines for the host-language mappings, here, the standard C-binding *cblas*.

Listing 1.2. Compiling autoBLAS code

```
$ autoblas -b cblas < example.c.in > example.c
$ gcc example.c -o example
```

In our specific example, the autoBLAS precompiler replaces the autoBLAS section in the host-language file with a call to `gemv`, which is the BLAS routine that computes matrix-vector products [1]. The generated code, shown in Listing 1.3, is now valid C++ code that can be passed to a conforming compiler like `gcc`.

Listing 1.3. C++ code generated by the autoBLAS precompiler

```
int rows = 10;
int col  = 20;
std::vector<double> y(rows);
std::vector<double> A(rows * cols);
std::vector<double> x(cols);
```

```
// init y, A, and x with application specific values
cblas_dgemv(CblasColMajor, CblasTrans, rows, cols, 1.0,
            A.data(), cols, y.data(), 1, 0.0, x.data(), 1);
// continue using y, A, and x in C++
```

If we want to generate code for the CUDA backend, then we just have to invoke the autoBLAS precompiler with `autoblas -b cuda< example.c.in> example.c`. The generated code, shown in Listing 1.4, is now valid C++/Cuda code that again can be passed directly to a conforming compiler.

Listing 1.4. C++/CUDA code generated by the autoBLAS precompiler

```
int rows = 10;
int col  = 20;
std::vector<double> y(rows);
std::vector<double> A(rows * cols);
std::vector<double> x(cols);
// init y, A, and x with application specific values
cublasHandle_t handle;
cublasCreate(&handle);
double * d_x; cudaMalloc(&d_x, x.size() * sizeof(double));
cudaMemcpy(d_x, x.data(), x.size() * sizeof(double),
           cudaMemcpyHostToDevice);
double * d_y; cudaMalloc(&d_y, y.size() * sizeof(double));
double * d_A; cudaMalloc(&d_A, A.size() * sizeof(double));
cudaMemcpy(d_A, A.data(), A.size() * sizeof(double),
           cudaMemcpyHostToDevice);
const double alpha = 1.0;
const double beta = 0.0;
cublasDgemv(handle, CUBLAS_OP_T, rows, cols, &alpha, d_A,
            cols, d_x, 1, &beta, d_y, 1);
cudaMemcpy(d_y, y.data(), y.size() * sizeof(double),
           cudaMemcpyDeviceToHost);
cudaFree(d_A);
cudaFree(d_y);
cudaFree(d_x);
cublasDestroy(handle);
// continue using y, A, and x in C++
```

5.2 Design

By defining an embedded language of its own, autoBLAS is as intuitive to use as task specific frameworks like MATLAB when it comes to expressing *what* to compute. Additionally, by not being bound to a particular programming language, autoBLAS can perform any transformation and optimization necessary on the symbolic level at compile time, even beyond the scope of a single statement. Finally, autoBLAS delegates the task of *how* to evaluate the optimized expressions by generating the corresponding BLAS calls. This allows the user to utilize highly efficient BLAS implementations for the target platform without having to write these calls by hand.

Figure 3 illustrates the three abstract steps of the autoBLAS compiler. The *frontend* is the user-facing part of autoBLAS and comprises both the expression syntax as well as the context selection. The context specifies attributes of the variables like, for instance, their memory layout and the BLAS selections available at compile time.

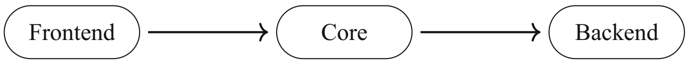


Fig. 3. The design of autoBLAS is divided in three independent components: the user-facing frontend, the optimizing core, and the executing backend.

The *core* implements a set of symbolic optimizations to increase execution performance at runtime, while also allocating and reusing memory for temporaries, if necessary. By performing these syntax-tree optimizations independent of a specific target API, autoBLAS provides a uniform evaluation semantic across different target platforms, hereby, minimizing unpleasant surprises like different operator semantics or optimization behavior when switching between libraries.

The *backend* generates code for the optimized expressions for the respective linear algebra library selected by the caller. Backends define a set of necessary attributes for evaluating expressions into code. For instance, for the cblas [1] backend, a dense matrix is often represented by a data pointer, a storage orientation, the number of rows and columns and the size of the leading dimension. A context is compatible with a specific backend if it provides all necessary attributes for a particular data type.

An advantage of selecting a BLAS-like backend is, that when later profiling the code, the user is able to directly refer to potential bottlenecks. This is in contrast to template-based libraries like Eigen, where the actually called routines are not directly visible and do not correspond to a particular line within the host-language code.

A major benefit of the separation into frontend, core and backend is that extending autoBLAS with a new backend is rather simple and in practice merely requires to derive from a class and implement BLAS-expression to target code mappings. At the same time, a developer who is extending autoBLAS in this way still benefits from all the symbolic optimizations implemented in the autoBLAS core.

6 Conclusions

Making generic optimization (GENO) work efficiently requires several fairly different interoperable software components. In this chapter we have described such components and their integration into the GENO software stack. By carefully designing, implementing and integrating the components in the GENO software we are able to generate optimization code that is competitive with problem-specific hand-written solvers and orders of magnitude faster than competing approaches that are comparably easy to use. Furthermore, the components, specifically, the generic optimizer, the matrix and tensor calculus, and autoBLAS, are of independent interest and are also used in other projects than GENO.

References

1. Blackford, L.S., et al.: An updated set of basic linear algebra subprograms (BLAS). ACM Trans. Math. Softw. **28**(2), 135–151 (2002)
2. Agrawal, A., Verschueren, R., Diamond, S., Boyd, S.: A rewriting system for convex optimization problems. J. Control Decis. **5**(1), 42–60 (2018)
3. Anderson, E., et al.: LAPACK Users' Guide. 3rd edn. Society for Industrial and Applied Mathematics, Philadelphia, PA (1999)
4. various authors: clblas (2017). https://github.com/clMathLibraries/clBLAS. Accessed 21 Aug 2017
5. Banjac, G., Stellato, B., Moehle, N., Goulart, P., Bemporad, A., Boyd, S.P.: Embedded code generation using the OSQP solver. In: CDC, pp. 1906–1911 (2017)
6. Bertsekas, D.P.: Nonlinear Programming. Athena Scientific, Belmont, MA (1999)
7. Birgin, E.G., Martínez, J.M.: Practical augmented Lagrangian methods for constrained optimization, Fundamentals of Algorithms, vol. 10. SIAM (2014)
8. Brooke, A., Kendrick, D., Meeraus, A.: GAMS: release 2.25 : a user's guide. The Scientific press series, Scientific Press (1992)
9. Byrd, R.H., Lu, P., Nocedal, J., Zhu, C.: A limited memory algorithm for bound constrained optimization. SIAM J. Sci. Comput. **16**(5), 1190–1208 (1995)
10. Corporation, I.: Intel math kernel library (2017). https://software.intel.com/en-us/mkl. Accessed 21 Aug 2017
11. Corporation, N.: Nvidia cublas (2017). https://developer.nvidia.com/cublas. Accessed 21 Aug 2017
12. CVX Research Inc: CVX: Matlab software for disciplined convex programming, version 2.1 (2018). http://cvxr.com/cvx
13. Diamond, S., Boyd, S.: CVXPY: a Python-embedded modeling language for convex optimization. J. Mach. Learn. Res. **17**(83), 1–5 (2016)
14. Dunning, I., Huchette, J., Lubin, M.: JuMP: a modeling language for mathematical optimization. SIAM Rev. **59**(2), 295–320 (2017)
15. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: a modeling language for mathematical programming. Thomson/Brooks/Cole (2003)
16. Giselsson, P., Boyd, S.: Linear convergence and metric selection for Douglas-Rachford splitting and ADMM. IEEE Trans. Autom. Control **62**(2), 532–544 (2017)
17. Grant, M., Boyd, S.: Graph implementations for nonsmooth convex programs. In: Blondel, V.D., Boyd, S.P., Kimura, H. (eds.) Recent Advances in Learning and Control. Lecture Notes in Control and Information Sciences, vol. 371, pp. 95–110. Springer, Cham (2008). https://doi.org/10.1007/978-1-84800-155-8_7
18. Griewank, A., Walther, A.: Evaluating derivatives - principles and techniques of algorithmic differentiation, 2 edn. SIAM (2008)
19. Hart, W.E., et al.: Pyomo-Optimization Modeling in Python, vol. 67, 2nd edn. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58821-6
20. Hart, W.E., Watson, J.P., Woodruff, D.L.: Pyomo: modeling and solving mathematical programs in Python. Math. Program. Comput. **3**(3), 219–260 (2011)
21. Hestenes, M.R.: Multiplier and gradient methods. J. Optim. Theory Appl. **4**(5), 303–320 (1969)
22. Huang, H.Y.: Unified approach to quadratically convergent algorithms for function minimization. J. Optim. Theory Appl. **5**(6), 405–423 (1970)
23. Laue, S.: On the equivalence of forward mode automatic differentiation and symbolic differentiation. arXiv e-prints abs/1904.02990 (2019)

24. Laue, S., Mitterreiter, M., Giesen, J.: Computing higher order derivatives of matrix and tensor expressions. In: NeurIPS, pp. 2755–2764 (2018)
25. Laue, S., Mitterreiter, M., Giesen, J.: GENO - generic optimization for classical machine learning. In: NeurIPS, pp. 2187–2198 (2019)
26. Laue, S., Mitterreiter, M., Giesen, J.: Matrixcalculus.org - computing derivatives of matrix and tensor expressions. In: ECML-PKDD, pp. 769–772 (2019)
27. Laue, S., Mitterreiter, M., Giesen, J.: GENO - optimization for classical machine learning made fast and easy. In: AAAI, pp. 13620–13621 (2020)
28. Laue, S., Mitterreiter, M., Giesen, J.: A simple and efficient tensor calculus. In: AAAI, pp. 4527–4534 (2020)
29. Laue, S., Mitterreiter, M., Giesen, J.: A simple and efficient tensor calculus for machine learning. Fund. Inform. **177**(2), 157–179 (2020)
30. Limited, A.: Arm performance libraries - optimized blas, lapack and fft (2017). https://developer.arm.com/products/software-development-tools/hpc/arm-performance-libraries. Accessed 21 Aug 2017
31. Mattingley, J., Boyd, S.: CVXGEN: a code generator for embedded convex optimization. Optim. Eng. **13**(1), 1–27 (2012)
32. Morales, J.L., Nocedal, J.: Remark on "algorithm 778: L-BFGS-B: fortran subroutines for large-scale bound constrained optimization". ACM Trans. Math. Softw. **38**(1), 7:1–7:4 (2011)
33. Moré, J.J., Thuente, D.J.: Line search algorithms with guaranteed sufficient decrease. ACM Trans. Math. Softw. **20**(3), 286–307 (1994)
34. Nazareth, L.: A relationship between the BFGS and conjugate gradient algorithms and its implications for new algorithms. SIAM J. Numer. Anal. **16**(5), 794–800 (1979)
35. Nesterov, Y.: A method for unconstrained convex minimization problem with the rate of convergence $O(1/k^2)$. Doklady AN USSR (translated as Soviet Math. Docl.) 269 (1983)
36. Nesterov, Y.: Smooth minimization of non-smooth functions. Math. Program. **103**(1), 127–152 (2005)
37. Olsen, P.A., Rennie, S.J., Goel, V.: Efficient automatic differentiation of matrix functions. In: Forth, S., Hovland, P., Phipps, E., Utke, J., Walther, A. (eds.) Recent Advances in Algorithmic Differentiation. Lecture Notes in Computational Science and Engineering, vol. 87, pp. 71–81. Springer, Cham (2012). https://doi.org/10.1007/978-3-642-30023-3_7
38. Powell, M.J.D.: Algorithms for nonlinear constraints that use Lagrangian functions. Math. Program. **14**(1), 224–248 (1969)
39. Ricci, G., Levi-Civita, T.: Méthodes de calcul différentiel absolu et leurs applications. Math. Ann. **54**(1–2), 125–201 (1900)
40. Vasilache, N., et al.: Tensor comprehensions: framework-agnostic high-performance machine learning abstractions. arXiv preprint arXiv:1802.04730 (2018)
41. Wolfe, P.: Convergence conditions for ascent methods. SIAM Rev. **11**(2), 226–235 (1969)
42. Wolfe, P.: Convergence conditions for ascent methods. ii: some corrections. SIAM Rev. **13**(2), 185–188 (1971)
43. Van Zee, F.G., van de Geijn, R.A.: BLIS: a framework for rapidly instantiating BLAS functionality. ACM Trans. Math. Softw. **41**(3), 14:1-14:33 (2015)
44. Zhu, C., Byrd, R.H., Lu, P., Nocedal, J.: Algorithm 778: L-BFGS-B: fortran subroutines for large-scale bound-constrained optimization. ACM Trans. Math. Softw. **23**(4), 550–560 (1997)

Algorithms for Big Data Problems in de Novo Genome Assembly

Anand Srivastav, Axel Wedemeyer(✉), Christian Schielke, and Jan Schiemann

Kiel University, Kiel, Germany
{srivastav,wedemeyer,schielke,schiemann}@math.uni-kiel.de

Abstract. De novo genome assembly is a fundamental task in life sciences. It is mostly a typical big data problem with sometimes billions of reads, a big puzzle in which the genome is hidden. Memory and time efficient algorithms are sought, preferably to run even on desktops in labs. In this chapter we address some algorithmic problems related to genome assembly. We first present an algorithm which heavily reduces the size of input data, but with no essential compromize on the assembly quality. In such and many other algorithms in bioinformatics the counting of k-mers is a botleneck. We discuss counting in external memory. The construction of large parts of the genome, called contigs, can be modelled as the longest path problem or the Euler tour problem in some graphs build on reads or k-mers. We present a linear time streaming algorithm for constructing long paths in undirected graphs, and a streaming algorithm for the Euler tour problem with optimal one-pass complexity.

Keywords: De novo genome assembly · Data reduction · Euler tour · Semi-streaming longest path · External memory counting

1 Reduction of Input Data in Genome Assembly

Sequencing is a chemical and physical process in which DNA is 'crushed' into very small parts ('fragments') which are 'read' to strings called reads, containing information of the sequence of nucleotides. Reads are of limited length and contain errors (Fig. 1).

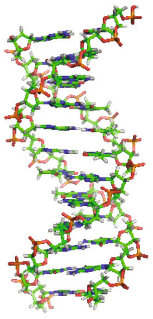
Sequencing of big genomes and other samples is a computationally challenging recent trend for two main reasons:

– sequencing became much cheaper (price decreased by more than 100.000× since year 2000), so researchers can afford to create much bigger data sets than ever before (Fig. 2)
– it was discovered that most bacteria (90%–99%) can't be cultivated, so metagenomic sequencing is (nearly) the only way to assess them .

1.1 Reads, Coverage and Assembly

A **read** is a string over the alphabet $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}, \mathsf{N}\}$ where A, C, G, T are the four nuclobases and N is a place holder for an unknown nucleotide. The maximal read length depends on the sequencing technology used. For the Illumina sequencing technology the read length initially was 34 and now can be as high as 300. Other sequencers allow

Biochemical molecule $\rightarrow \Sigma^* = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}, \mathsf{N}\}^*$



```
@EAS18:1:1:1:1:971:0/1
NGTCTTTTATAACCAGTTCACCTATAGCAAAGACGCTATTCATACCGAAGGCGTCATT
+
B`bbbb`bb`_abaXU\Za\ab_X_`Xa_R[X\a_b_YX`a__X_b`YH\T`b`_\_]
@EAS18:1:1:1:1:971:0/2
NAAANCGGCAANTTAANNNTAACGCCCATAATGCCGATAAACCCCCATTTAAAAAANT
+
B_YaBbaaaTWBa^P`BBBb[aa`a__P_Z`aab`Z]__^\``a_\[abb[IZPPbBa
```

Fig. 1. A shortened paired (Illumina) read

for longer reads at the price of a higher error rate (and higher costs). Illumina produces substitution errors with an error rate of roughly 1%. In most cases, so called paired reads are generated where in a first step pieces of DNA of a known length are produced which are than sequenced from both sides (e.g.: a paired read with a read length of 150 contains one string over Σ for the first 150 nucleotides and a second string over Σ for the last 150 nucleotides).

The sequencer also outputs so-called phred scores quantifying the error probability of each nucleotide read (Quality $Q = -10\log_{10} P$ where P is the error probability).

Genome assembly (or just assembly) is the task to reconstruct the complete genome of the sequenced species using the reads only (*de novo assembly*) or the reads and a reference genome (*mapping* or *reference based assembly*). It's like a puzzle with millions of small parts, unknown overlaps and a lot of the parts containing errors.

In our work we focus on de novo assembly, or just assembly for the rest of this chapter.

For a sequencing data set, the **coverage** of a genomic position A is the number of reads in the data set which contain A. The coverage of the whole data set is the average over the coverages of all genomic positions. The empirical 'optimal' coverage for a de novo assembly is about 20 *at every position*. A coverage higher than 20 means redundant data. Some sequencing protocols, especially single cell MDA (multiple displacement amplification), produce read sets with an extreme uneven coverage distribution. Metagenomic data sets may have an uneven coverage distribution, too, when both abundant and rare species are sequenced. Given a string σ over the nucleotide alphabet, a k**–mer** is a sub-string of σ of length k.

```
GTCTTTTATAAC
GTCTTT
 TCTTTT
  CTTTTA
   TTTTAT
    TTTATA
     TTATAA
      TATAAC
```

the 6–mers of a string

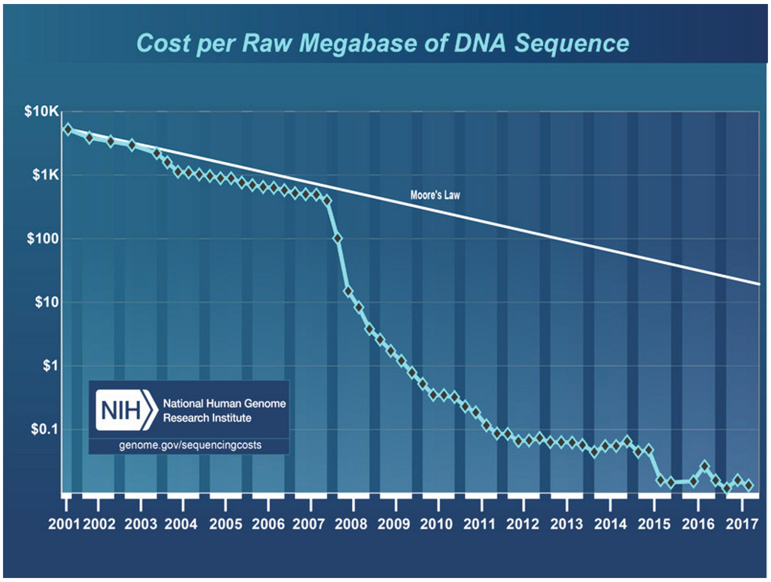


Fig. 2. The cost of sequencing a human genome, source: NIH

Most bacteria can't be cultivated in the lab. Therefore, it is not possible to create a homogeneous sample of thousands or millions of equal cells as in a 'normal' sequencing setting.

As a consequence, **single cell** sequencing protocols, like the multiple displacement amplification (MDA) have been developed which are able to amplify the genome of a single bacterial cell. A drawback of these methods is a strong amplification bias (called 'Preferential amplification' and 'Allelic dropout') between different regions of the genome, meaning that the coverage of some regions of the genome might overshot 100.000X, while other regions are not covered at all.

A **metagenome**, introduced by [14], is, according to wiktionary, '*All the genetic material present in an environmental sample, consisting of the genomes of many individual organisms*'. In other words, in a metagenomic experiment, you are interested in

– all the genes/DNA
– of everything living
– at a specific location

The experiment is conducted by collecting a sample from the desired environment, isolating the DNA from it and sequencing it with a Next Generation Sequencing (NGS) system. There are three different types of metagenomic experiments with different goals:

– *phylogenetic profiling:* based upon the 16S ribosomal RNA found in the sample, reconstruct which families of bacteria live in the probed environment (and how abundant they are). Basis: each (bacterial) cell has ribosomes. The coding genes for these essential proteins are widely conserved (which makes it possible to identify these

genes), but they also include less conserved regions which differ between different families or even species.
– *directed/guided assembly of specific genes:* based upon some known variants of a gene (or even whole genomes), all existing variants in a specific environment are to be assembled.
– *de novo assembly of all species in the sample:* all genomes of all species in the probed environment are to be assembled using the output of the sequencer only.

In our work, we focus on de novo assembly.

The main problem of metagenome assembly is non-uniform coverage: some species in the sample are much more abundant than others. The goal is to assemble all their genomes. The following issues may arise:

– to be able to assemble the less abundant species in the sample, a high number of reads have to be generated ($\rightarrow$ high coverage sequencing).
– the huge input files force the assembler programs to use huge amounts of RAM and running time. For bigger projects, even $1TB$ of RAM might not be enough.
– for the assembler, its often hard to tell whether a rare sequence belongs to a rare species or whether it is a sequencing error.

1.2 The Bignorm Algorithm

The basic idea of read filtering is to remove reads from a single cell or metagenome data set without losing information, and in this way to reduce the size of the problem, possibly escaping the 'big data curse'. This is possible if only those reads which have overlapping genomic regions with high coverage are removed. A good read filter should remove as many reads as possible, without lowering the coverage of the sequenced genome below the desired threshold at any position and without increasing the error rate of the data set.

Highly memory efficient algorithms are sought to solve this problem. Brown et al. invented an algorithm named *Diginorm* [1] for read filtering that rejects or accepts reads based on the abundance of their *k*–mers. The name *Diginorm* is a short form for *digital normalization*: the goal is to normalize the coverage over all loci, using a computer algorithm after sequencing. The idea is to remove those reads from the input which mainly consist of *k*–mers that have already been observed many times in other reads. Diginorm processes reads one by one, splits them into *k*–mers, and counts these *k*–mers. In order to save RAM, Diginorm does not keep track of those numbers exactly, but instead keeps appropriate estimates using the count-min sketch CMS [4]. A read is accepted if the median of its *k*–mer counts is below a fixed threshold, usually 20. It was demonstrated that successful assemblies are still possible after Diginorm removed high amount of the data.

Diginorm is a pioneering work. However, the following points, which are important from the biological or computational point of view, are not covered by Diginorm. We have included them in our algorithm called Bignorm [29 SPP]:

(i) we incorporate the important phred quality score into the decision whether to accept or to reject a read, using a quality threshold. This allows a tuning of the filtering process towards high-quality assemblies by using different thresholds.

(ii) when deciding whether to accept or to reject a read, we do a detailed analysis of the numbers in the count vectors. Diginorm merely considers their medians.
(iii) we offer a better handling of the N case, that is, when the sequencing machine could not decide for a particular nucleotide. Diginorm simply converts all N to A, which can lead to false k–mer counts.
(iv) we provide a substantially faster implementation. For example, we include fast hashing functions (see [10, 30]) for counting k–mers through the count-min sketch data structure (CMS), and we use the C programming language and OpenMP.

Let us fix the following parameters:

– N-*count threshold* $N_0 \in \mathbb{N}$, which is 10 by default;
– *quality threshold* $Q_0 \in \mathbb{Z}$, which is 20 by default;
– *rarity threshold* $c_0 \in \mathbb{N}$, which is 3 by default;
– *abundance threshold* $c_1 \in \mathbb{N}$, which is 20 by default;
– *contribution threshold* $B \in \mathbb{N}$, which is 3 by default.

When our algorithm has to decide whether to accept or reject a read $i \in \mathbb{N}$, it performs the following steps: If the number of N symbols counted over all read positions is larger than N_0, the read is rejected. Otherwise, those parts of the read having phred scores of or above Q_0 are converted into a vector H of *high-quality k–mers*.

Using the CMS, it is then checked how many times these k–mers have been seen in the accepted reads so far (function $\widehat{c}(\mu)$) and two counters hold the results:

$$b_0 := |\{\mu \in H\,;\ \widehat{c}(\mu) < c_0\}|,$$
$$b_1 := |\{\mu \in H\,;\ c_0 \leq \widehat{c}(\mu) < c_1\}|$$

Note that the frequencies are determined via CMS counters and do not consider the position p at which the k–mer is found in the read string. The read is accepted if and only if at least one of the following conditions is met:

$$b_0 > k, \tag{1}$$
$$\sum_{s=1}^{m(i)} b_1 \geq B. \tag{2}$$

The motivation for condition (1) is as follows. According to [15], most errors of the Illumina sequencing platform are single substitution errors and the probability of appearance of an erroneous k–mer in the genome, caused by an incorrect reading of a nucleotide, is quite low. Thus, k–mers produced by single substitution errors are likely to have very small counter values in the CMS (less than c_0 times) and can be considered as rare k–mers. One such error can only effect at most k k–mers. So if we count more than k rare k–mers, they most likely are not a result of one single substitution error. If we assume that the probability of multiple single substitution errors in a read is smaller than the probability of error-free rare k–mers, we should accept this read.

Condition (2) says that in the read, there are enough (namely at least B) k–mers where each of them appears too frequently to be a read error (CMS counters at least c_0), but not that abundant that it should be considered redundant (CMS counters less than c_1).

Algorithm 1: Bignorm

```
Input: fastq–files as produced by an (Illumina–)sequencer
Result: filtered fastq–files
Parameter: $k$–mer size $k$
Parameter: quality threshold $Q_0$
Parameter: rarity threshold $c_0$
Parameter: abundance threshold $c_1$
Parameter: contribution threshold B
Parameter: N-count threshold $N_0$
1  begin
2      initCMS ()
3      foreach read r in input do
4          if count of N in r < $N_0$ then
5              $b_0 = 0$
6              $b_1 = 0$
7              foreach canonical k–mer $\kappa$ in r do
8                  if min (phred_scores ($\kappa$)) $\geq Q_0$ then
9                      $t_\kappa = getCount(\kappa)$
10                     if $t_\kappa < c_0$ then
11                         $b_0 += 1$
12                     else if $t_\kappa \leq c_1$ then
13                         $b_1 += 1$
14             if $b_0 > k$ OR $b_1 \geq B$ then
15                 Add r to Output
16                 foreach canonical k–mer $\kappa$ in r do
17                     $incCount(\kappa)$
```

Results for Single-Cell Assemblies. We tested Bignorm on 13 bacterial single-cell data sets and were able to remove up to 90% of the reads without significant loss of the assembly quality. Some results (median of all samples) (Fig. 3):

Measurement	Filtered/Unfiltered (%)
Read count	2.85
Run time SPAdes Assembler	3.57
Largest Contig	97.56
N50	90.84
Mean Phred Score	103.00

Bignorm heavily cuts away redundant reads (mean, Fig. 4, left-hand side) but is careful in critical regions (P10, Fig. 4, right-hand side).

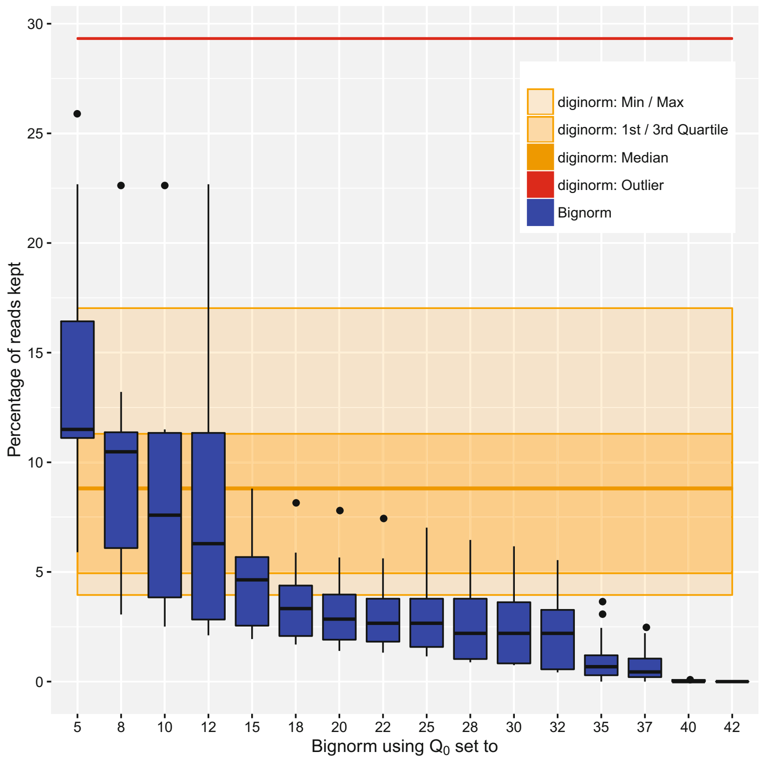


Fig. 3. Reads kept

Results for Metagenomic Assemblies. We tested Bignorm on metagenomic data sets. For data sets with reads of length about 250 base pairs, the results are quite promising and stable. Compared to the single cell case, the results are not that impressive, but compared to the State–of–the–art approach of *sub-sampling* data sets which are too big to be assembled on the given hardware (this means a certain proportion of reads is selected randomly), we could show that by read filtering it is possible to get results which are nearly as good as those of assembling the complete data set, using about the same amount of RAM and in run time as using the sub-sampling approach. The following table gives an impression on the results:

	Raw	Filtered	Sub-sampled (3x)
Largest Contig	2183	1731	1356 ± 143
Total length	385282	358036	136552 ± 8406
Genome fraction (%)	15.0	14.0	5.4 ± 0.3
Predicted genes	689	648	262 ± 12
RAM needed (GB)	212	100	96 ± 0.6
Run time (h)	151	52	52 ±2

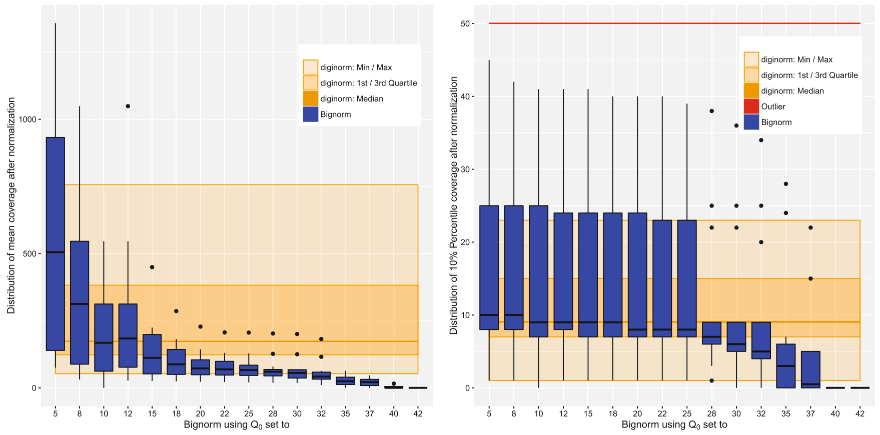


Fig. 4. Coverage: mean and critical region

2 Counting *k*–mers in External Memory (EM)

Many bioinformatics algorithms (e.g., assemblers, error correctors, read normalization) are based on *k*–mers, and that requires to count them (mostly for $21 \le k \le 127$). As bioinformatics data sets are growing much faster than RAM sizes, new computational models are needed. (We could show that hash–based counting, which is state of the art in current software, will produce $\mathcal{O}(n^2)$ hash table dumps when the number of different *k*–mers is much bigger than the number of slots in the hash table.)

Some examples of recent *k*–mer counting algorithms are:

- `jellyfish (2)` [19]: the standard, *hash-based k*–mer counter
- `dsk` [26]: the first *EM-based* counter
- `kmc (2/3)` [8,9,17]: the state–of–the–art *EM-based* counter.
- `bloomfish` [12]: *MPI-based Map–Reduce* framework for counting
- `squeakr` [21]: based on *counting quotient filter* (a probabilistic data structure)
- `turtle` [27]: using a *Bloomfilter* and *sort–and–compact* algorithm

We need some notations:

- All strings are based on the biological alphabet $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$.
- So the base set for a *k*–mer is $\mathscr{M} := \Sigma^k$ and $m := |\mathscr{M}| = 4^k$.
- The input of a *k*–mer counter is $\eta \in \underbrace{\mathscr{M} \times \mathscr{M} \times \cdots \times \mathscr{M}}_{n\ times} = \mathscr{M}^n$.
- Denote by $\mathscr{C} := \{p \in \mathscr{M} \mid \exists_{i \le n} : p = \eta_i\}$ the set of *k*–mers occurring in the input at least once, $c := |\mathscr{C}|$.
- Let R be the size (in bytes) of the RAM available.
- Let B the number of bytes needed to count each element of $\mathscr{C}$.

2.1 Counting in RAM

The **straightforward algorithm** for small values of k, counting can be done in RAM. If $m \leq R/B$, the following $\mathcal{O}(n+m)$ algorithm can be used:

Algorithm 2: trivial counting

```
Input: fastq–files as produced by an (Illumina–)sequencer
Result: k–mers of input and their number of occurrence in the input
1 begin
2     initialize $c_0^{m-1} = 0$
3     foreach canonical k–mer $\kappa$ in input do
4         $c_\kappa = c_\kappa + 1$
5     for $i \leftarrow 0$ to $(m-1)$ do
6         if $c_i > 0$ then
7             output $i, c_i$
```

For $k = 19$ and one Byte per counter, $4^{13} \approx 275GB$ of RAM is needed.

Hash–Based Counting. Most state–of–the–art k–mer counting programs are based on hash algorithms using open addressing:

– Hash table with h entries of size $B + \lceil \frac{\log_2 m}{8} \rceil = B + \lceil \frac{\log_2 4^k}{8} \rceil = B + \lceil \frac{k}{4} \rceil \rightarrow h(B + \lceil \frac{k}{4} \rceil) \leq R$
– For hash size $h \gg c$ the time complexity is $\mathcal{O}(n+h)$. But if $h \approx c$ the run time may increase to $\mathcal{O}(nh)$ and if $c > h$, the program will fail (or dump to external memory)
– Most existing programs will dump the full hash tables and merge them afterwards — for bigger data sets, this merging phase may need days and terabytes of external memory. The runtime depends linearly on the expected value $\mathbb{E}[d(h,n)]$ of the number of hash table dumps. We can show the following formula for the expected value.

Theorem 1 (Gallus, Srivastav, Wedemeyer 2021). *Counting a set of n elements of a population with K different, normally distributed types using a hash table of size h, the expected value of hash table dumps $d(h,n)$ is*

$$\mathbb{E}[d(h,n)] = n \log_{(1-\frac{h}{c_n})} \left(1 - \frac{1}{c_n}\right), \tag{3}$$

where $c_n = K(1-(1-\frac{1}{K})^n)$ gives the number of different (normally distributed) types in a set of size n.

This formula is the basis for further quantifying the log-term in (3). If one can show that this log-term behaves linearly or sublinearly in n in case of including singletons in the the set of k-mers, it would match experimental observations. In fact, a constant portion of the k–mers can be assumed as sequencing errors of which each occures exactly once.

2.2 Counting in External Memory

kmc3 is the presently leading program using the external memory model. It works as follows:

- the input is parsed into $k-x$–mers (a combination of up to 3 k–mers)
- they are split into a prefix and a suffix, the suffixes are written to one temporary file per prefix
- each temporary file is loaded one by one into RAM, sorted (radix sort, library *radul*)
- the sorted $k-x$–mers are unified and counted
- written to a pair of special binary files (one index-file for the prefixes and one with the suffixes and the counts)

Drawback of kmc3: The output files of kmc3 are not completely sorted (due to the introduction of $k-x$–mers in kmc2). Therefore,

- they need to be loaded into RAM completely for read out
- exporting to other formats takes more time than counting
- no compression is in place (although the suffix–files are highly compressible)

As a result, even though kmc3 is the fastest EM k–mer counter available (and the fastest k–mer counter overall under RAM restriction), it is not the perfect choice to be used as a counting module for an EM assembler.

Based on STXXL 1.4.1 [5], in 2018 Christopher Nehls [20] from Kiel University developed a k–mer counter called **xsc** which uses a sorting based approach:

- generate k–mers from input
- sort the k–mers (using the STXXL EM sorter)
- count the k–mers

For $k \leq 32$, xsc outperformed jellyfish and was at least competitive to dsk, but kmc3 was always faster. For $k > 33$ (using uint128 and uint256 classes), xsc was not competitive to the existing counters. The main bottleneck of xsc is the overloaded relational operator (operator<).

2.3 Counting Using a Bloomfilter

Roy et.al. [27] stated that *more than 50% of all k–mers in a sequencing data set may be singletons* — which are not of interest as they were probably introduced by errors. To utilise this, their k–mer counter Turtle uses a upstream Bloomfilter to save space and time in a sorting based approach named ‘sort–and–compact’.

We developed a program which combines the ideas of kmc and the usage of a Bloomfilter, experiments show that the cost of running a bloomfilter is higher than the savings (Fig. 5). What is wrong? Say, we have 100 k–mers,

- 50 singletons (occurring once)
- 50 ‘good’ k–mers occurring $100\times$ on average

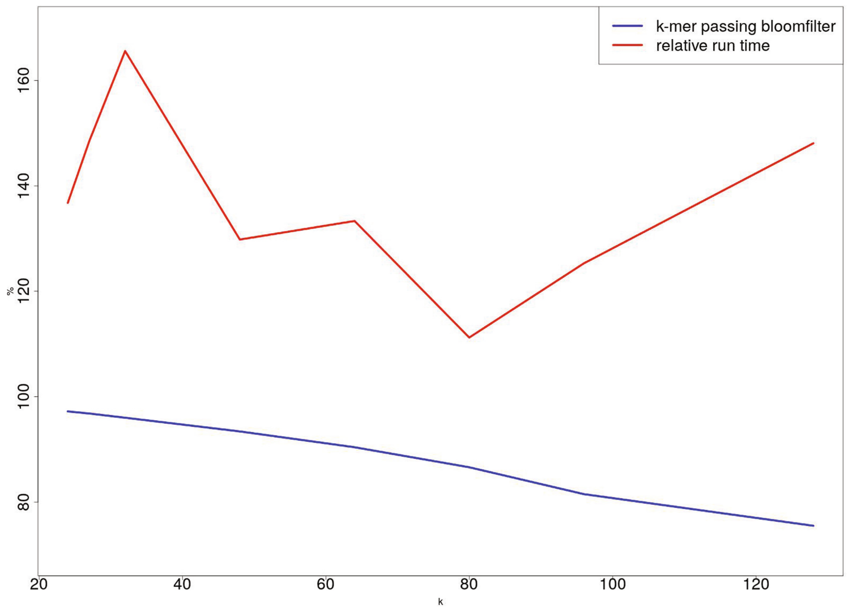


Fig. 5. Comparison of run times using or not using a bloomfilter

Our input contains 5050 *k*–mers, the bloomfilter removes $100 \rightarrow \approx 2\%$ of the input, not enough to compensate for the running time of the bloomfilter.

Our Current Approach. We have developed the following algorithm which combines sorting and kmc. Experiments are ongoing work:

```
Input: fastq–files as produced by an (Illumina-)sequencer
Result: k–mers of input and their number of occurrence in the input
1 begin
2     foreach canonical k–mer $\kappa$ in input do
3         split $\kappa$ into prefix and suffix
4         use turtle–like sort–and–compact per prefix
5         if an array is full then
6             dump the sorted array to EM
7     merge arrays
```

3 A Streaming Algorithm for the Longest Path Problem

In de novo genome assembly, finding a large genome sequence called contig is the fundamental problem. It can be understood as computing a very long path in the associated graph, for example the de Bruijn graph ([3]). Unfortunately, computing the longest path in a graph is an NP-hard problem and the situation is even more worse if the graph is very large. In this chapter, we present a new algorithm for computing a long path, which is surprisingly competitive with RAM-based algorithms.

Graph streaming is a very efficient concept to handle big graphs, where the number of edges is far too large for computations in the main memory. The semi-streaming model was introduced by Feigenbaum et al. [11], and can be briefly described as follows:

In the semi-streaming model, the algorithm is allowed to use at most $\mathcal{O}(n \cdot \text{polylog}(n))$ bits of RAM where n is the number of vertices of the input graph. Because of this restriction, dense graphs where the number of edges is in the order of $\omega n \cdot \text{polylog}(n)$, cannot be processed entirely in RAM. Instead, the edges are presented in a stream where the edges are in no particular order. Typically, it is desired to call only a small number of passes (over the input stream).

3.1 Our Tree-Based Algorithm

We give a streaming algorithm for the longest path problem in undirected graphs with a proven per-edge processing time of $\mathcal{O}(n)$ published in the proceedings of the European Symposium on Algorithms in 2016 [16 SPP]. Our algorithm works in two phases, which we outline here briefly and explain in detail in Sect. 3.1. In the first phase, global information on the graph is gathered in form of a constant number of spanning trees $T_1, \ldots, T_\tau$. This is possible in the streaming model since roughly speaking, for a spanning tree we can "take edges as they come". A spanning tree can be constructed in just one pass—we however use multiple passes and limit the maximum degree during the first passes in order to favor path-like structures and avoid clusters of edges. Experiments clearly indicate that this degree-limiting is essential for solution quality. The spanning trees fit into RAM, since we consider τ as constant (we will in fact have $\tau = 1$ or $\tau = 2$ in the experiments). After construction of the τ trees, they are merged into one graph U by taking the union of their edges. Then we use standard algorithms to determine a long path P in U, isolate P, and finally add enough edges around P to obtain a tree T.

Then, in the second phase, we conduct further passes during which we test if the exchange of single edges of T can improve the longest path in it. (A longest path in a tree can be found by conducting DFS two times [2]; the length of a longest path in a tree is its diameter.) The main challenge in the second phase is to quickly determine which edges should be exchanged. We show that this decision can be made in linear time, hence yielding a per-edge processing time of $\mathcal{O}(n)$.

For a set X, we write x unif X to express that x is drawn uniformly at random from X.

An example run of the Algorithm is shown in Fig. 6.

3.2 Linear Complexity of the Streaming Algorithm

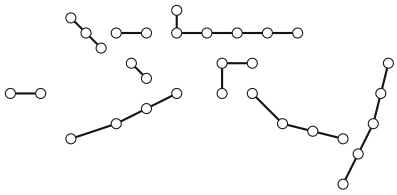
If the cycle C is of length $\Omega(n)$, then a naive implementation requires $\Omega(n^2)$ to find an edge e' to remove (temporarily remove each edge on the cycle and invoke the Dijkstra algorithm). However, we have:

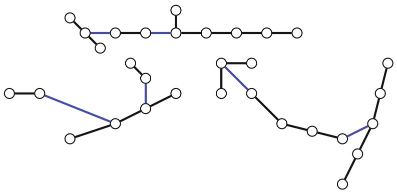
Theorem 2 (Kliemann, Schielke, Srivastav 2016). *Phase 2 can be implemented with per-edge processing time $\mathcal{O}(n)$.*

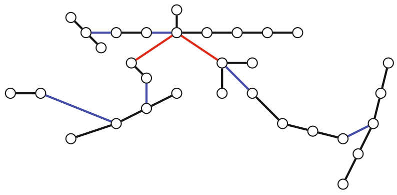
```
Algorithm 3: Streaming Phase 1: Spanning Tree Construction
  Input: connected graph $G = (V, E)$ as a stream of edges, parameter $\tau$,
         degree limit sequence $D = (D_1 \dots D_{q_1})$
  Output: spanning tree of $G$
1 foreach $i = 1, \dots, \tau$ do
2     $T_i := (V, \emptyset)$
3     SpanningTree($T_i$)
4 $U := (V, \bigcup_{i=1}^{\tau} E(T_i))$
5 find a long path $P$ in $U$ using Warnsdorf's algorithm
6 $T := (V, E(P))$
7 SpanningTree($T$)
8 return $T$
```

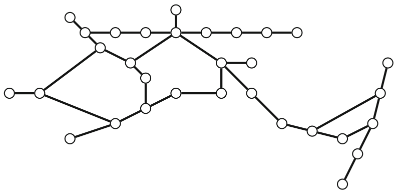
```
Procedure SpanningTree(T)
  Input: forest $T$ on $V$, possibly empty
  Output: spanning tree on $V$
1 $r =_{\text{unif}} [m]$
2 fast-forward the stream to position $r$
3 for $p = 1, \dots, q_1$ do
4     while not at the end of the stream do
5         get next edge $vw$ from the stream
6         if $T + vw$ is cycle-free and $\max\{\deg_T(v), \deg_T(w)\} < D_p$ then $T := T + vw$
7         if $|T| = n - 1$ then break
8     rewind the stream to its beginning
```

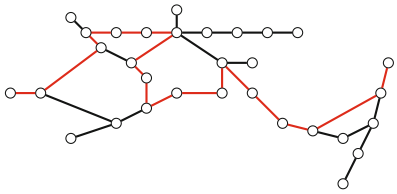
```
Algorithm 4: Streaming Phase 2: Improvement
  Input: connected graph $G$ as a stream of edges, spanning tree $T$, pass limit $q_2$
  Output: a (long) path in $G$
1  compute longest path $P$ in $T$ with Dijkstra algorithm
2  for $q_2$ times do
3      rewind the stream to its beginning
4      while not at the end of the stream do
5          get next edge $e = vw$ from stream
6          if $v \in V(P)$ and $w \in V(P)$ then discard and continue with next iteration
7          $T' := T + e$
8          compute fundamental cycle $C$ in $T'$
9          $\ell^* := \max_{f \in E(C) \setminus \{e\}} \ell(T' - f)$
10         if $\ell^* > |P|$ then
11             pick any $e'$ from the set $\{f \in E(C) \setminus \{e\} : \ell(T' - f) = \ell^*\}$
12             $T := T' - e'$
13             update $P$ with longest path in $T$
14 return $P$
```

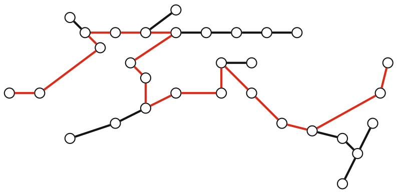

a: Degree Limit $D = 2$

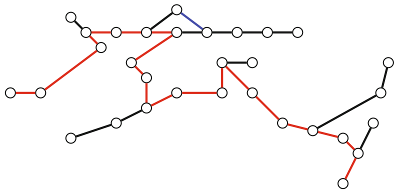

b: Degree Limit $D = 3$

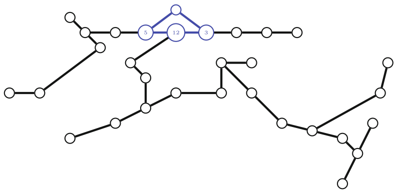

c: Degrees Unlimited


d: Union of Trees

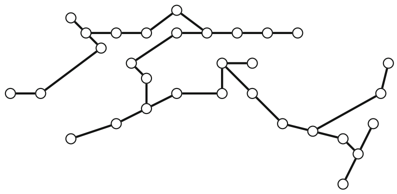

e: Path Found by Warnsdorf's Algorithm


f: New Spanning Tree Built around Path


g: Added Edge from Stream



h: Depths of Trees Extending from Fundamental Cycle


i: Remove Edge from the Cycle

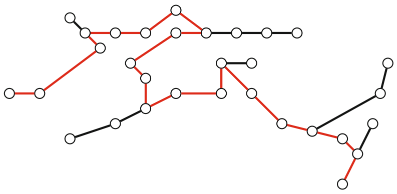

j: New Longest Path

Fig. 6. Example run of the algorithm's steps.

Proof. An $\mathcal{O}(n)$ bound is clear for all lines of Algorithm 4, except Line 9 and Line 11. Denote

$$\ell' := \max_{f \in E(C) \setminus \{e\}} \max\{|P| : P \text{ is path in } T' - f \text{ and } e \in E(P)\}$$

and let $R' \subseteq E(C) \setminus \{e\}$ be the set of edges where this maximum is attained. Then the following implications hold: $\ell' \leq |P| \implies \ell^* \leq |P|$ and $\ell' > |P| \implies \ell' = \ell^*$. This is because if a longest path in $T' - f$ is supposed to be longer than P, it must use e (since otherwise it would be a path in T). Hence it suffices to determine ℓ', and if $\ell' > |P|$, to find an element of R'.

Denote $C = (v_i, \ldots, v_k)$ the fundamental cycle for some $k \in \mathbb{N}$ written so that $e = v_1 v_k$. When computing ℓ', we can restrict to paths in T' of the form

$$(\ldots, v_s, v_{s-1}, \ldots, v_1, v_k, v_{k-1}, \ldots, v_t, \ldots) \tag{4}$$

for $1 \leq s < t \leq k$, where v_s is the first and v_t is the last common vertex, respectively, of the path and C. For each i, let T_i be the connected component of v_i in $T - E(C)$, i.e., T_i is the part of T that is reachable from v_i without using the edges of C. Denote $\ell(T_i)$ the length of a longest path in T_i that starts at v_i and denote $c_i := \ell(T_i) + i - 1$ and $a_i := \ell(T_i) + k - i$. Then a longest path entering C at v_s and leaving it at v_t, as in (4), has length exactly $c_s + a_t$. Hence we have to determine a pair (s,t) such that $c_s + a_t$ is maximum (this maximum value is ℓ'); we call such a pair an *optimal pair*. If the so determined value ℓ' is not greater than $|P|$, then nothing further has to be done (the edge e cannot give an improvement). Otherwise, having constructed our optimal pair (s,t), we pick an arbitrary edge (e.g., uniformly at random) from $\{v_i v_{i+1} : s \leq i < t\}$, which are the edges between v_s and v_t on C. We show that the following algorithm computes the value ℓ' and an optimal pair in $\mathcal{O}(n)$.

```
1  compute $c_1, \ldots, c_{k-1}$ and $a_2, a_k$ using DFS
2  $M := 0; L := 0$
3  for $i = 1, \ldots, k-1$ do
4      if $c_i > M$ then
5          $M := c_i$
6          $s := i$
7      if $M + a_{i+1} > L$ then
8          $L := M + a_{i+1}$
9          $t := i + 1$
10 return $(s,t)$
```

The total of computations in Line 1 can be done by DFS in $\mathcal{O}(n)$, and the loop in $\mathcal{O}(k) \leq \mathcal{O}(n)$. We prove that the final (s,t) is optimal. For fixed t, the best possible length $c_s + c_t$ is obtained if t is combined with an $s < t$ where $c_s \geq c_j$ for all $j < t$. In the algorithm, for each t (when $t = i + 1$ in the loop) we combine a_t with the maximum $\max_{j<t} c_j$ (stored in the variable M). Thus, when the algorithm terminates, $L = \ell'$ and $c_s + c_t = \ell'$.

Corollary 1. *Our streaming algorithm (with the two phases as in Algorithm 3 and Algorithm 4) can be implemented with a per-edge processing time of* $\mathcal{O}(n)$.

We turn to the memory requirement. Denote by b the amount of RAM required to store one vertex or one pointer (e.g., $b = 32bit$ or $b = 64bit$) and call $n \cdot b$ one *unit*.

Theorem 3. *Our streaming algorithm (with the two phases as in Algorithm 3 and Algorithm 4) conducts at most $2q_1 + q_2$ passes. Moreover, the algorithm can be implemented such that the RAM requirement is at most $(\max\{4\tau, 2\tau+4\} \cdot n + c) \cdot b$ with a constant c.*

The proof can be found in [16 SPP].

An **experimental study** was conducted on randomly generated instances with different structure, including ones created with the generator for hyperbolic geometric random graphs [18 SPP]. Different variants of our streaming algorithm are compared with four RAM algorithms: Warnsdorf and Pohl-Warnsdorf (two related classical heuristics [23,24]), Pongrácz (a recently published heuristic [25]), and a simple randomized DFS. Experiments show that although we never do more than 11 passes, results delivered by our algorithm are competitive. We deliver at least 71% of the best result delivered by any of the tested RAM algorithms, with the exception of preferential attachment graphs. By considering low percentiles, we observe a similar quality without any restriction on the graph class. This is a good result also in absolute terms, since we observe that for each graph class and set of parameters, there is one algorithm that on average gives a path of length $0.84 \cdot n$, i.e., 84% of a Hamilton path. On some graph classes, we outperform any of the tested RAM algorithms, which makes our algorithm interesting even outside of the streaming setting.

4 An One Pass Streaming Algorithm for Computing the Euler Tour in Graphs

Large genome sequences (contigs) can be computed in de novo genome assembly with so-called de Bruijn graphs on k-mers ([3,22]). Such graphs are directed. For very large graphs, the computation of an Euler tour cannot be done with known RAM-based algorithms and techniques like semi-streaming or external memory algorithms are sought. In this chapter, we present a survey on our optimal one-pass streaming algorithm for computing an Euler tour in an undirected graph. Our algorithm might be helpful to design a semi-streaming algorithm to compute Euler tours in a directed graph, which is an open problem.

Let G be a graph on n nodes and m edges given in the form of a data stream. We study the problem of finding an Euler tour in G. We present a survey on the first one-pass streaming algorithm computing an Euler tour of G in the form of an edge successor function with only $\mathcal{O}(n\log(n))$ RAM based on our paper [13 SPP]. The memory requirement is optimal for this setting according to Sun and Woodruff [28].

4.1 The W-Streaming Model and a Lower Bound

The *W-streaming model* was introduced by Demetrescu et al. [7]. It is a relaxation of the classical streaming model. At each pass, an output stream is written, which becomes the input stream of the next pass. For an Euler tour the successor of each edge in the tour is uniquely defined by its successor function, say δ. Then the output stream has the following form, where the edges are unordered.

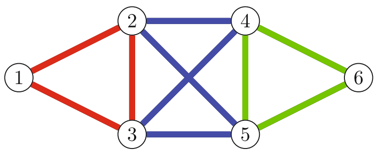
…	e	$\delta(e)$	$\delta(\delta(e))$	…

Finding an Euler tour in trees in W-streaming has been studied in multiple papers (e.g., [6]), but the general Euler tour problem has hardly been considered in a streaming model. There are some general results for transferring PRAM algorithms to the W-streaming model. In general, lower bounds for the complexity of streaming algorithms are hard to prove. Interestingly, Sun and Woodruff [28] showed that even a one-pass streaming algorithm for verifying whether a graph is Eulerian needs $\Omega(n\log(n))$ RAM, and this amount of RAM is also required for a one pass streaming algorithm for finding an Euler tour.

4.2 The Problem of Cycle Merging

The Euler tour problem in the RAM model can be easily solved by computing edge-disjoint cycles and merging them. We will see, why this is a problem with limited RAM. A *cycle* is a closed walk on the edges of G such that every node is visited at most once. The following result is well-known in graph theory.

Theorem 4. *If a graph with m edges contains an Euler Tour, it can be decomposed into at most $\frac{m}{3}$ pairwise edge-disjoint cycles.*



In fact, this can be accomplished in one pass.

Theorem 5. *During the pass, the edges from the input-stream can be ordered in form of a sequence of edge-disjoint cycles.*

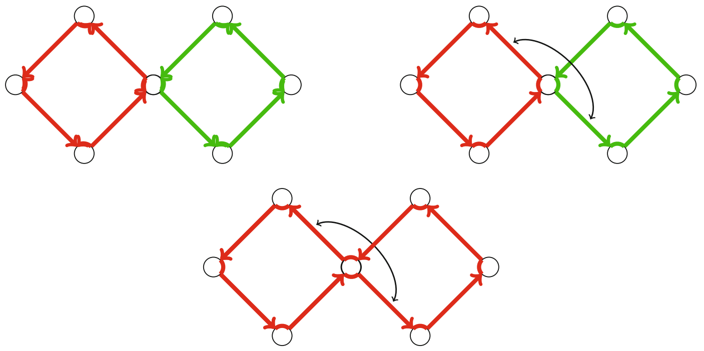
Proof. 1. Start with $T := \emptyset$
2. While T is cycle-free, add edges from the input stream to T
3. When a cycle occurs in T, store it and delete all its edges from T. Go to Step 2.

At every time, T contains at most n edges.

If $T \neq \emptyset$ at the end, there are some nodes of odd degree, thus G does not contain an Euler Tour.

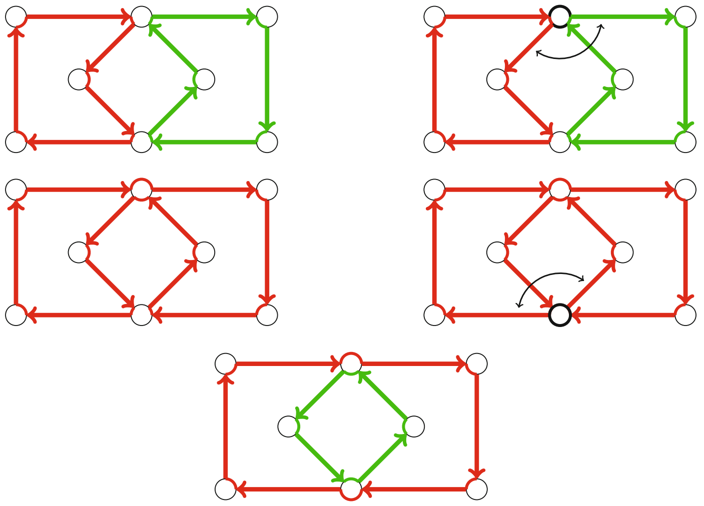
Obviously and unfortunately, we cannot store all the cycles in the semi-streaming model. The challenge is to merge cycles, when they are appearing with respect to the memory limitation of $\mathcal{O}(nlog(n))$. We will use the notion of tours or subtours for cycles, too.

The merging of two tours at one node is easy. We just flip edges in canonical way and get the new tour:



Similarly, one can merge several tours at one common node.

The problematic case is the simultaneous merging at two nodes. There is an example.



Unfortunately, the result of this merging is two tours, and the merging failed. A problem only occurs if the cycle shares more than one node with an already existing tour. In this case, we have to make sure that edge-swapping is performed at exactly one of these nodes. Every node belongs to at most one tour at a time, thus all nodes of a tour can get the same label.

4.3 The W-Streaming Algorithm and Its Analysis

We proceed to the pseudo-code statement of our streaming algorithm.

```
Algorithm 5: EULER-TOUR
input  : Undirected graph $G = (V, E)$, edge by edge on a stream $S$
output: Euler tour for $G$, i.e. a successor function $\delta^*$, if there is one
1  $c := 0$;  $F := \emptyset$;  $E_{\text{int}} := \emptyset$;  for every $v \in V$: $s(v) := 0, t(v) := 0$
2  for every edge e on S do
3      $E_{\text{int}} := E_{\text{int}} \cup \{e\}$
4      if $G_{\text{int}} = (V, E_{\text{int}})$ contains a cycle $C$ then
5          node MERGE-CYCLE ($C$)
6  if $E_{\text{int}} = \emptyset$ then
7      ERROR: At least one node with odd degree exists
8  if there exist $u, v$ with $t(u) \neq t(v) \neq 0$ then
9      ERROR: Graph is not connected
10 WRITE-F
```

```
Procedure Merge-Cycle
input  : Ordered directed cycle $C = (v_1, \dots, v_k)$ of length $k$
1  NEW-NODES
2  CHOOSE-NODES
3  WRITE
4  MERGE
5  UPDATE
6  for every edge $e \in C$ do
7      delete $e$ from $E_{\text{int}}$
```

The output stream is a successor function, i.e. e_1, $\delta(e_1)$, e_2, $\delta(e_2)$, ... For $a, b, c \in V$ with (a, b); $(b, c) \in \vec{E}$ the triple (a, b, c) represents the successor function $(a, b) \to \delta((a, b)) = (b, c)$. So, edge (b, c) is the successor of edge (a, b). The output stream is *not* necessarily an ordered trail!

The main result is the following theorem [13 SPP].

Theorem 6 (Glazik, Schiemann, Srivastav, 2017). *There exists an one-pass W-Streaming algorithm with* $\mathrm{O}(n \log n)$ *RAM that outputs an Euler tour on the input graph G (if G contains an Euler tour).*

We sketch the proof. Let δ be a successor function. Equivalence classes: $e \in E$: $[e]_\delta = \{f \in E;\ \underbrace{e \equiv_\delta f}_{\exists k: \delta^k(e) = f}\ \}$ We identify the successor function with equivalence classes on $\vec{E}$.

Lemma 1 (Algebraic Representation [13 SPP], Lemma 1). *Let* δ *be a bijective successor function on a directed graph* $\vec{G} = (V, \vec{E})$*. Then* $\equiv_\delta$ *is an equivalence relation on* $\vec{E}$.

Lemma 2. *Let $\vec{G} = (V, \vec{E})$ be a directed graph with bijective successor function δ and the related equivalence relation $\equiv_\delta$. Then we have:*

(i) Let $e \in \vec{E}$ and $k_1, k_2 \in \mathbb{N}$ with $k_1 \neq k_2$ and $\delta^{k_1}(e) = \delta^{k_2}(e)$. Then $|k_1 - k_2| \geq |[e]_\delta|$.
(ii) For any $e \in \vec{E}$ we have $\delta^{|[e]_\delta|}(e) = e$.

Proof. (i): $F_s : \vec{E} \to \vec{E}$, $s \in \mathbb{N}$, $F_s(e') = \delta^{s(k_1-k_2)}(e')$.

- $\delta^{k_2}(e)$ fixpoint of F_s.
- $M := \{\delta^\ell(e); k_2 \leq \ell < k_1\}$, $|M| \leq k_1 - k_2$
- $[e]_\delta \subseteq M$ by fixpoint property of F_s

The assumption $k_1 - k_2 < |[e]_\delta|$ implies $|M| < |[e]_\delta| \leq |M| \to$ *contradiction*
(ii): $r := |[e]_\delta|$. Lets assume for a moment that $\delta^r(e_0) \neq e_0$ for some e_0.

- $M := \{\delta^\ell(e_0); 1 \leq \ell \leq r\} \subseteq [e_0]_\delta$

Case 1: $e_0 \in M$. Then

$$\delta^0(e_0) = e_0 = \delta^\ell(e_0) \text{ for some } \ell < r.$$

By (i): $\ell - 0 \geq r \to$ *contradiction*

Case 2: $e \notin M$. Then $|M| < |[e]_\delta|$. By the pigeonhole principle, there exist $1 \leq k_1, k_2 \leq |[e]_\delta|$ with $\delta^{k_1}(e) = \delta^{k_2}(e)$ in contradiction to (i).

Further, a structured theorem is needed. For an edge $e = (v, w)$ let $e_{(1)} := v$, $e_{(2)} := w$.

Theorem 7 (Successor function generates Euler tour [13 SPP], **Theorem 3).** *Let $= \vec{G}(V, E)$ be a directed graph with bijective successor function δ such that $e \equiv_\delta e'$ for all $e, e' \in \vec{E}$. Then δ is the successor function of an Euler tour for G.*

Let δ^0 be the successor function of an edge disjoint cycle decomposition of G. The algorithm computes a sequence of successor functions $\delta_0^* = \delta^0, \delta_1^*, \ldots, \delta_N^* := \delta^*$

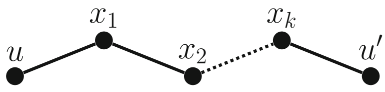
Theorem 8. *If G is Eulerian, δ^* determines an Euler tour on G.*

The following lemma is the backbone of the proof and requires substantial work.

Lemma 3 ([13 SPP], **Lemma 9).** *Let $k \in \{0, \ldots, N\}$. Then, δ_k^* is bijective and for any $(u, v), (u', v') \in R^*(E)$, we have*

(i) If $(u, v), (u', v')$ are processed edges, then $(u, v) \equiv_{\delta_k^*} (u', v') \Leftrightarrow t_k(u) = t_k(u')$.
(ii) If (u, v) is a processed edge, then $t_k(u) = t_k(v)$.
(iii) If $t_k(u) = 0$, then $(u, v) \equiv_{\delta_k^*} (u', v') \Leftrightarrow (u, v) \equiv_\delta (u', v')$.

Proof (Proof of Theorem 8). We show: If δ^* is bijective and $e \equiv_{\delta^*} e'$ for all e, e', then δ^* is an Euler tour by Theorem 3. Then, by Lemma 3, $\delta^* = \delta_N^*$ is bijective. For the second property let $e, e' \in E$, $e = (u, v)$ and $e' = (u', v')$. We show $e \equiv_{\delta^*} e'$. Now, there exists an u–u'–path P in G because G is Eulerian. Let $P = u\, x_1\, x_2 \ldots x_k\, u'$ such a path.



By Lemma 3 (ii), label t_N propagates through P:

$$\begin{aligned} & t_N(u) = t_N(x_1) = t_N(x_2) = \cdots = t_N(x_k) = t_N(u') \\ \Rightarrow \quad & t_N(u) = t_N(u') \\ \underset{\text{Lemma 3(i)}}{\Rightarrow} \quad & e \equiv_{\delta_N^*} e' \end{aligned}$$

In future work we may investigate other routing problems and applications for streaming algorithms using Euler tours.

References

1. Brown, C.T., Howe, A., Zhang, Q., Pyrkosz, A.B., Brom, T.H.: A reference-free algorithm for computational normalization of shotgun sequencing data, pp. 1–18. ArXiv e-prints (2012). https://arxiv.org/abs/1203.4802
2. Bulterman, R.W., van der Sommen, F.W., Zwaan, G., Verhoeff, T., van Gasteren, A.J.M., Feijen, W.H.J.: On computing a longest path in a tree. Inf. Process. Lett. **81**(2), 93–96 (2002). https://doi.org/10.1016/S0020-0190(01)00198-3
3. Compeau Phillip, E.C., Pevzner Pavel, A., Tesler, G.: How to apply de Bruijn graphs to genome assembly. Nat. Biotechnol. **29**(11), 987–991 (2011). https://doi.org/10.1038/nbt.2023
4. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. J. Algorithms **55**(1), 58–75 (2005). https://doi.org/10.1016/j.jalgor.2003.12.001
5. Dementiev, R., Kettner, L., Sanders, P.: STXXL: standard template library for XXL data sets. Softw. Pract. Exp. **38**(6), 589–637 (2008). https://doi.org/10.1002/spe.844
6. Demetrescu, C., Escoffier, B., Moruz, G., Ribichini, A.: Adapting parallel algorithms to the w-stream model, with applications to graph problems. Theor. Comput. Sci. **411**(44–46), 3994–4004 (2010). https://doi.org/10.1016/j.tcs.2010.08.030
7. Demetrescu, C., Finocchi, I., Ribichini, A.: Trading off space for passes in graph streaming problems. ACM Trans. Algorithms **6**(1), 6:1-6:17 (2009). https://doi.org/10.1145/1644015.1644021
8. Deorowicz, S., Debudaj-Grabysz, A., Grabowski, S.: Disk-based k-mer counting on a PC. BMC Bioinform. **14**, 160 (2013). https://doi.org/10.1186/1471-2105-14-160
9. Deorowicz, S., Kokot, M., Grabowski, S., Debudaj-Grabysz, A.: KMC 2: fast and resource-frugal k-mer counting. Bioinformatics **31**(10), 1569–1576 (2015). https://doi.org/10.1093/bioinformatics/btv022
10. Dietzfelbinger, M., Hagerup, T., Katajainen, J., Penttonen, M.: A reliable randomized algorithm for the closest-pair problem. J. Algorithms **25**(1), 19–51 (1997). https://doi.org/10.1006/jagm.1997.0873
11. Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., Zhang, J.: On graph problems in a semi-streaming model. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 531–543. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27836-8_46
12. Gao, T., et al.: Bloomfish: a highly scalable distributed k-mer counting framework. In: IEEE ICPADS 2017, pp. 170–179. IEEE Computer Society (2017). https://doi.org/10.1109/ICPADS.2017.00033

13 SPP. Glazik, C., Schiemann, J., Srivastav, A.: Finding Euler tours in one pass in the W-streaming model with O(n log(n)) RAM. CoRR abs/1710.04091 (2017). Theory of Computing Systems 2022, 23 p. Springer. https://doi.org/10.1007/s00224-022-10077-w

14. Handelsman, J., Rondon, M.R., Brady, S.F., Clardy, J., Goodman, R.M.: Molecular biological access to the chemistry of unknown soil microbes: a new frontier for natural products. Chem. Biol. **5**(10), R245-9 (1998). https://doi.org/10.1016/s1074-5521(98)90108-9
15. Kelley, D.R., Schatz, M.C., Salzberg, S.L.: Quake: quality-aware detection and correction of sequencing errors. Genome Biol. **11**(11), 1–13 (2010). https://doi.org/10.1186/gb-2010-11-11-r116
16 SPP. Kliemann, L., Schielke, C., Srivastav, A.: A streaming algorithm for the undirected longest path problem. In: ESA, pp. 56:1–56:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.ESA.2016.56
17. Kokot, M., Dlugosz, M., Deorowicz, S.: KMC 3: counting and manipulating k-mer statistics. Bioinformatics **33**(17), 2759–2761 (2017). https://doi.org/10.1093/bioinformatics/btx304
18 SPP. von Looz, M., Staudt, C.L., Meyerhenke, H., Prutkin, R.: Fast generation of dynamic complex networks with underlying hyperbolic geometry. CoRR abs/1501.03545 (2015). http://arxiv.org/abs/1501.03545
19. Marçais, G., Kingsford, C.: A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. Bioinformatics **27**(6), 764–770 (2011). https://doi.org/10.1093/bioinformatics/btr011
20. Nehls, C.: Effizientes sortier-basiertes Zählen von k-meren im externen Speicher. Mathematisches Seminar, Universität zu Kiel, Masterarbeit (2018)
21. Pandey, P., Bender, M.A., Johnson, R., Patro, R.: Squeakr: an exact and approximate k-mer counting system. Bioinformatics **34**(4), 568–575 (2018). https://doi.org/10.1093/bioinformatics/btx636
22. Pevzner, P.A., Tang, H., Waterman, M.S.: A new approach to fragment assembly in DNA sequencing. In: RECOMB, pp. 256–267. ACM (2001). https://doi.org/10.1145/369133.369230
23. Pohl, I.: A method for finding Hamilton paths and knight's tours. Commun. ACM **10**(7), 446–449 (1967). https://doi.org/10.1145/363427.363463
24. Pohl, I., Stockmeyer, L.: Pohl-Warnsdorf - revisited. In: Proceedings of the ISC 2004 (2004). https://users.soe.ucsc.edu/~pohl/Papers/Pohl_Stockmeyer_full.pdf
25. Pongrácz, L.L.: A greedy approximation algorithm for the longest path problem in undirected graphs. CoRR abs/1209.2503 (2012). withdrawn
26. Rizk, G., Lavenier, D., Chikhi, R.: DSK: k-mer counting with very low memory usage. Bioinformatics **29**(5), 652–653 (2013). https://doi.org/10.1093/bioinformatics/btt020
27. Roy, R.S., Bhattacharya, D., Schliep, A.: Turtle: identifying frequent k-mers with cache-efficient algorithms. Bioinformatics **30**(14), 1950–1957 (2014). https://doi.org/10.1093/bioinformatics/btu132
28. Sun, X., Woodruff, D.P.: Tight bounds for graph problems in insertion streams. In: APPROX-RANDOM, pp. 435–448. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015). https://doi.org/10.4230/LIPIcs.APPROX-RANDOM.2015.435
29 SPP. Wedemeyer, A., Kliemann, L., Srivastav, A., Schielke, C., Reusch, T.B., Rosenstiel, P.: An improved filtering algorithm for big read datasets and its application to single-cell assembly. BMC Bioinform. **18**(1), 324 (2017). https://doi.org/10.1186/s12859-017-1724-7
30. Wölfel, P.: Über die Komplexität der Multiplikation in eingeschränkten Branchingprogrammmodellen. Ph.D. thesis, Technical University of Dortmund, Germany (2003). http://hdl.handle.net/2003/2539

Scalable Text Index Construction

Timo Bingmann[1], Patrick Dinklage[2], Johannes Fischer[2], Florian Kurpicz[2](✉), Enno Ohlebusch[3], and Peter Sanders[1]

[1] Karlsruher Institut für Technologie, Karlsruhe, Germany
tb@panthema.net, sanders@kit.edu
[2] Technische Universität Dortmund, Dortmund, Germany
{patrick.dinklage,florian.kurpicz}@tu-dortmund.de,
johannes.fischer@cs.tu-dortmund.de
[3] Universität Ulm, Ulm, Germany
enno.ohlebusch@uni-ulm.de

Abstract. We survey recent advances in scalable text index construction with a focus on practical algorithms in distributed, shared, and external memory.

Keywords: Text indices · Suffix array · Suffix tree · Wavelet tree · Burrows-Wheeler transform · FM-index · Distributed memory · Shared memory · External memory

1 Introduction

Texts occur in many different domains, ranging from natural language texts over source code to DNA and protein sequences, and their amount is ever-increasing. The field of algorithm and data structure research on strings is often referred to as *Stringology*. One important aspect within this line of research is the efficient construction of text indices. A text index is a data structure that provides additional information for a given text to speed up answering different types of queries, e.g., pattern matching queries that ask if (or how often, or where) a pattern occurs in the text. We focus on full-text indices for possibly unstructured texts, which allow the user to query for arbitrary patterns (this excludes, e.g., inverted indices). Real-world applications of text indices can be found, for example, in computational biology where text indices are a crucial part of the software for DNA alignment [134]. However, the amount of textual data is increasing significantly faster than the computational capacity of ordinary computers. For example, in 2008 the 1000 Genomes Project (1KGP) was launched to collect and sequence the genomes of thousands of people, whereas, in 2020, the 1+Million Genomes Initiative (1+MG) started to collect at least one million genomes, making this collection 1000 times larger. Therefore, scalable construction algorithms that can handle the massively growing amount of text are necessary.

In this survey, we discuss the current state of the art in scalable text index construction. We focus on distributed, external, and shared memory construction algorithms for different text indices and their applications. While there already exist surveys focussing on particular indices (e.g., suffix arrays [28,172] or wavelet trees [149,160,63 SPP]), or models of computation (e.g., external memory [23,56]), this chapter tries to give

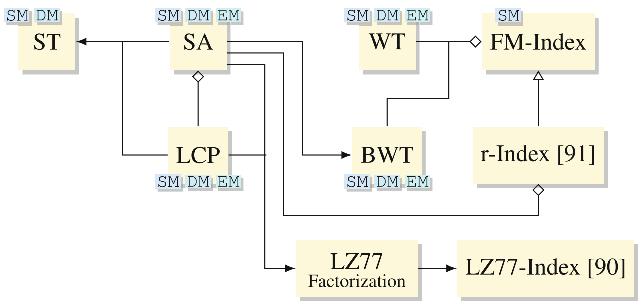


Fig. 1. Relations of text indices. In this article, we consider text indices that have scalable construction algorithms. The labels `SM`, `DM`, `EM` mark whether such construction algorithms in shared, distributed, and external memory exist. Note that the LZ77 factorization itself is a text compression and not an index. We use arrows (→) to denote indices that are used (in practice) to compute the targeted text index or if they are a special case (–▷) of the targeted index. Diamonds (–◇) are used to denote indices that are part of the targeted text index.

a more unified view. To this end, we point out common techniques that are used in different models of computation or in the computation of different text indices.

This survey is structured as follows. First, in Sect. 2, we introduce models of computation and give an overview of (string) sorting algorithms and further building blocks that are required as basic tools for text index construction. The main body of work can be found in Sect. 3. Here, we discuss the scalable construction of different text indices. We start with the suffix array (*SA*), one of the most well-researched text indices, and the longest common prefix (*LCP*) array, which often accompanies the SA. Next, we take a look wavelet trees (*WT*) and the Burrows-Wheeler Transform (*BWT*), which both are important parts of the FM-index, a compressed text index frequently used in practice. Then, we discuss algorithms for the suffix tree (*ST*) and space efficient representations thereof. See Fig. 1 for an overview of the text indices and their relations. While most of the discussed work solely focuses on the construction of the text indices, we also show approaches to *answer* queries on text indices in distributed memory. Finally, in Sect. 4, we show real-world applications of text indices in bioinformatics and text compression before we address future challenges in Sect. 5.

2 Preliminaries

Let $T = T[0] \dots T[n-2]\$$ be a text of length n over an alphabet $\Sigma = [0, \sigma)$, where we assume that T is terminated with an end-of-file or sentinel symbol \$ with $\$ \notin \Sigma$ and $\$ < \alpha$ for all $\alpha \in \Sigma$. A text over an alphabet of size $\sigma = 2$ is called *bit vector*. Usually, bit vectors do not contain a sentinel. We call $T[i..j) = T[i] \dots T[j-1]$ a *substring* of T for $i, j \in [0, n]$. The substrings $T[0..i)$ and $T[j..n)$ are called *prefix* and *suffix* for $i, j \in [0, n]$.

2.1 Models of Computation

In this section, we introduce models of computation that are relevant for the rest of this chapter and give pointers to software libraries that are commonly used to implement algorithms in those models. The starting point is the sequential *random access machine* (RAM) model [182], where we have a single *processing element* (PE) that contains multiple registers to perform operations on data and a main memory, which can be accessed in constant time. However, real-world systems are often more complex and require more sophisticated models.

One of these models is the *external memory* (EM) model [4]. Here, we have an internal memory of size M words and an external memory of unlimited size that is much slower to access randomly. To compensate for this, transfer between EM and RAM happens in blocks of B consecutive words. Such a transfer is called *I/O operation* (I/O for short). The cost of external memory algorithms is then described by the number of required I/Os, e.g., scanning through N elements requires $\Theta\left(\frac{N}{B}\right)$ I/Os, and sorting N elements requires $\mathrm{sort}(N) := \Theta\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ I/Os. The software libraries STXXL [57] and TPIE [9] implement the most commonly used external memory algorithms and data structures. A (practical) relaxation of the model is the *semi-external* model, where we allow random access to either the input or output, but not both. The Succinct Data Structure Library (SDSL) [94] provides implementations of semi-external construction algorithms for various data structures.

We also consider two parallel machine models, where by p we always denote the number of available PEs. The first is the *parallel random access machine* (PRAM), where all PEs have access to the same (shared) memory. There are various PRAM variants differentiating between which types of concurrent memory reads/writes are allowed; for practical algorithms on a multi-core processor one should only use exclusive writes, implying that the *Concurrent Read Exclusive Write* (CREW) model is best for analyzing algorithms. In the analysis, the *work* and *depth* are of interest. The former is the total number of operations performed, and the latter is the longest sequence of sequential dependencies in the algorithm. When implementing shared memory algorithms, Cilk [38] (now deprecated), OpenMP [53], Intel's TBB [174], Microsoft's Parallel Patterns Library (PPL), or built-in concurrency features of the programming language, e.g., `thread` in C++11, are often used to express parallelism. The Multi-Core Standard Template Library (MCSTL) [188] provides parallel algorithms and can be used as the *parallel mode* of the GNU C++ Standard Library. Recently, ParlayLib [36] was introduced as a library containing efficient implementations of the parallel algorithms in the C++ Standard Library.

The *distributed memory* model is our second parallel machine model. Here, communication between different PEs is conducted by sending messages over a network, and PEs have only local memory. Often, the cost of such a message is given as a startup cost plus a cost that depends on the size of the message. This is also reflected in the *bulk-synchronous parallel* model [200], where algorithms are divided into a sequence of supersteps consisting of three phases: local work, communication, and synchronization. The cost of an algorithm is then the sum of the costs of all supersteps. In practice, there are two flavors of frameworks for developing distributed algorithms: low-level interfaces provided by the *message passing interface* (MPI)[1] with its open-source implementations Open MPI [89] and MPICH [98], and frameworks providing a more

[1] MPI standard: https://www.mpi-forum.org/docs (last accessed 2020-07-14).

high-level functionality, e.g., Apache Flink [5], Apache Hadoop (based on MapReduce [54]), Apache Spark [210], and Thrill [29 SPP].

2.2 Building Blocks

Sorting. Sorting is a fundamental and well-studied topic in computer science, and the many results fill entire volumes [129,146] of related work. Hence, we will only review recent results for sorting integers in this section, which can be used in various of the following text indexing algorithms. In applications, sorting is most often still performed using classic sequential algorithms [107,159], despite existing more cache- or instruction-efficient variants [12,65,180,205] and well-developed modern parallel algorithms for shared-memory machines such as IPS^{4}o [17], or the sorters in the MCSTL [188], Intel's TBB [174], the PBBS [186], ParlayLib [36], or Microsoft's PPL. Another method of accelerating sorting is by vectorizing comparisons or operations using SIMD instructions [35,41,87,108,110,207,209].

For sorting integers, there is also the option of using radix sort algorithms, which have to be implemented carefully for modern CPUs [123,152,173]. Many parallel radix sorts for shared-memory machines are also available [138,165,192,203], and are most prominent on GPUs [101,109,154,181,194].

Sorting of data on external memory is a classic subject [4,58], and implementations are available in specialized libraries like TPIE [8] or STXXL [57].

An entirely different challenge is sorting on highly-scalable distributed shared-nothing machines, where load balancing, communication, and data redistribution have to be devised carefully, as PEs do not share memory. Most distributed memory sorting algorithms are based on either Quicksort [1,13,16,133,178,196] or sample sort [13,15,37,60,96,106,193,14 SPP].

Sorting is often used as a black box for text indexing algorithms, but depending on the model, machine, or scenario, large performance gains are possible by picking a better sorting implementation.

String Sorting. Sorting strings is an interesting special case of sorting, especially for text indexing algorithms, and most classical sorting algorithms have been adapted to multi-component objects or multi-key data [26,33,123,152,162,189]. Early parallel algorithms were formulated in the PRAM model and are based on merging of tries [102,113]. For external memory, theoretical algorithms were proposed, distinguishing short and long strings [7], or using hashing [70]. Many well-developed cache-efficient sequential and shared-memory parallel string sorting algorithms [28,33,30 SPP] are available in the TLX C++ library[2]. The fastest sequential ones are engineered variants of radix sort with very little memory overhead, and the fastest shared-memory parallel one is a string-aware sample sort implementation. These implementations also support outputting the lengths of the longest common prefixes (LCPs) of lexicographically adjacent strings at next to zero extra cost.

[2] TLX website: https://panthema.net/tlx/ (last accessed 2020-10-18).

While in principle the shared-memory parallel algorithms could be adapted to shared-nothing distributed supercomputers, they neglect that *communication volume* is the limiting factor for the scalability of algorithms to large systems [6, 39]. The first distributed string sorting algorithm we developed was a straight-forward adaptation of merge sort for use in a distributed suffix array construction algorithm [78 SPP]. This first version still considered strings as unbreakable objects.

Bingmann et al. therefore developed genuine distributed string sorting algorithms based on multi-way merge sort [34 SPP], which break up the strings into characters. The strings on each PE are first sorted locally. The PEs then collectively execute a distributed partitioning algorithm which yields p ranges of equal size with respect to the entire data. Each range is spread across the p machines in p fragments, and in the next step, each PE sends its misplaced $p-1$ fragments to the corresponding target machine. Finally, each PE merges the received partition fragments. The appeal of multi-way merging for communication-efficient sorting is that the local sorting exposes common prefixes of the local input strings. The *Distributed String Merge Sort* (MS) exploits this by only communicating the length of the common prefix with the previous string followed by the remaining characters. Here, the LCP values also allow us to use the multiway LCP-merging technique previously developed by Bingmann et al. [30 SPP] in such a way that characters are only inspected once.

The second algorithm, *Distributed Prefix-Doubling String Merge Sort* (PDMS), further improves communication efficiency by only communicating characters that may be needed to establish the global ordering of the data (the distinguishing prefix). The algorithm also has optimal local work for a comparison-based string sorting algorithm. The key idea is to apply the communication-efficient duplicate detection algorithm by Sansers et al. [179] to geometrically growing prefixes of each string. Once a prefix has no duplicate anymore, we know that it is sufficient to transmit only this prefix. The same idea was also used to make *any* PRAM algorithm LCP-aware [68 SPP].

An experimental evaluation of MS and PDMS (which are implemented in MPI) on up to 1280 cores shows that these algorithm are often more than five times faster than previous non-string-aware algorithms. In the future, we hope that these algorithms will find their way into general purpose distributed toolkits such as Apache Spark [210] or Thrill [29 SPP].

Further Building Blocks. The *prefix sum* (w.r.t. a binary associative operator $\oplus$) of n elements $A[0], \ldots, A[n-1]$ is an array B of n elements with $B[i] = \bigoplus_{k=0}^{i} A[k]$ for $i \in [0, n)$. In the PRAM model, the prefix sum of n elements can be computed in $\mathcal{O}(\lg n)$ depth and $\mathcal{O}(n)$ work [112, p. 47]. Due to their ubiquity, algorithms for prefix sums are part of frameworks used in different parallel models, e.g., distributed [29 SPP] and shared memory [188].

Rank and *select* data structures for a bit vector of length n allow us to compute the number of set (or unset) bits up to position $i \in [0, n)$ (rank), and the position of the j-th set (or unset) bit for $j \in [1, n]$ (select), respectively. They are an important ingredient of wavelet trees (see Sect. 3.2). To the best of our knowledge, the only parallel construction algorithms for rank and select data structures are described by Shun [185] and require $\mathcal{O}(\lg n)$ depth and $\mathcal{O}(n/\lg n)$ work if the n bits are packed into $\lceil n/\lg n \rceil$ words.

	0	1	2	3	4	5	6	7	8	9	10	11
T	`m`	`i`	`s`	`s`	`i`	`s`	`s`	`i`	`p`	`p`	`i`	`$`
SA	11	10	7	4	1	0	9	8	6	3	5	2
LCP	0	0	1	1	4	0	0	1	0	2	1	3

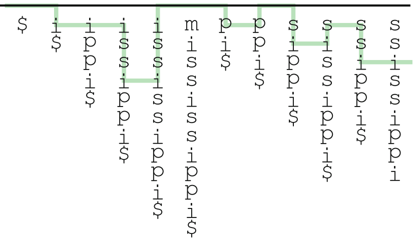


Fig. 2. Suffix array and longest common prefix array (see Sect. 3.1) for the text $T =$ `mississippi$`. Below, we also show the suffixes in lexicographical order, i.e., the suffixes represented in the suffix array. There, we also visualize the longest common prefixes of two lexicographically consecutive suffixes in green (🟢). (Color figure online)

In practice, only sequential construction has been considered, e.g., [46,147,211]. However, the construction of the data structures proposed by Zhou et al. [211] heavily relies on prefix sums and could thus easily be parallelized.

We can generalize binary rank and select queries for a text T. Then, the function $rank_\alpha(T,i)$ counts, for some character $\alpha \in \Sigma$ and a text position $i \in [0,n)$, the number of occurrences of α in $T[0..i]$, whereas $select_\alpha(T,k)$, for some $k > 0$, finds the position of the k-th occurrence of α in T. Generalized rank/select queries can be answered efficiently using wavelet trees, which reduce them to $\mathcal{O}(\lg\sigma)$ binary rank/select queries (see Sect. 3.2).

3 Text Indices

A text index provides additional information for a text to speed up answering different types of queries. In the following, we give an overview of different construction algorithms for text indices in the models that we describe in Sect. 2.1.

3.1 Scalable Suffix Array Construction

One of the best-researched text indices is the *suffix array* (*SA*), which has been introduced by Manber and Myers [150] and independently by Gonnet et al. [95] as the PAT array. The *SA* of a text T of length n is a permutation of $[0,n)$ such that $T[SA[i],n) < T[SA[j],n)$ for all $0 \le i < j < n$, i.e., it lists all suffixes lexicographically. See Fig. 2 for an example. Suffix arrays are a space efficient replacement of *suffix trees* (*ST*) (see Sect. 3.3). To obtain the same functionality as the *ST*s, *SA*s are often accompanied by additional arrays containing further information. Since suffix array construction algorithms sort all suffixes of a text, we use the term *suffix sorting* synonymously with suffix array construction.

When both the text and the *SA* fit into memory, the *SA* can be computed in linear time using the *difference cover* algorithm [124]. The idea is to sample suffixes and sort the samples. Using the sorted samples, we can lexicographically compare two suffixes in constant time. First, we compute SA^{12} containing all suffixes starting at positions that are not a multiple of three, i.e., suffixes starting at positions that are congruent to 1 and 2 modulo 3. To this end, we interpret three characters as one (increasing the alphabet size) and recursively call this algorithm until all characters are unique. Then, the SA^0 of all other suffixes is computed using the already computed SA^{12}. To obtain the final *SA*, SA^0 and SA^{12} are merged. The algorithm described above is called *DC3*. It can be generalized to other difference covers modulo $X > 3$; then we refer to it as *DCX*. The DCX algorithm can easily be adapted to several models of computation where it also is asymptotically optimal [124]. However, it often impractical due to substantial constant factor overheads, while *induced sorting* algorithms (Sect. 3.1) are superior, at least in the sequential computations. But the latter are hard to parallelize. Closing this gap between theory and practice is an interesting open problem for algorithm engineering. Note that all but one [20] sequential linear time suffix sorting algorithms rely on recursion. The *SA* can be constructed sequentially with only constant space overhead while retaining a linear running time [97, 141]. For more information on sequential suffix sorting, we point to two extensive surveys [28, 172] and a practical evaluation [19 SPP].

We now give an overview of suffix sorting algorithms in external memory, in shared memory (briefly touching also GPUs), and in distributed memory. Later, we take a look at the LCP array, one of the arrays often supplementing the *SA*.

External Memory. Crauser and Ferragina [52] and Dementiev et al. [56] present EM prefix doubling algorithms with discarding. The idea of *prefix doubling* [150] is to sort all suffixes based on the *h-order* $\leq_h$, defined by $T[i,n) \leq_h T[j,n) \Leftrightarrow T[i,i+h) \leq T[j,j+h)$ ($=_h$ and $<_h$ are defined analogously). The *h-rank* of a suffix is the number of suffixes that are strictly smaller w.r.t. the *h*-order. Now, during the *k*-th iteration, we compute the 2^k-ranks using the 2^{k-1}-ranks: for all suffixes $T[i,n)$, we use the ranks of $T[i,i+2^{k-1})$ and $T[i+2^{k-1},i+2^k)$, which are known from the previous iteration. We stop when ranks are unique; then, each rank is the position of that suffix in the *SA*. In practice, we can *discard* those *h*-ranks that are unique and not needed to compute other ranks any more, which can speed up the sorting, as it reduces the number of elements that we have to sort. For texts with small alphabets, prefix doubling algorithms are in practice often sped up by *alphabet reduction* in combination with *word packing*, e.g., [56, 81, 32 SPP, 78 SPP]. Here, an alphabet of size σ is first mapped to $[0,\sigma')$ such that $\sigma' \leq \sigma$ each character of the new alphabet occurs at least once in the text and they retain their original order. Then, each character is augmented such that it not only stores $T[i]$, but also the following $\lfloor b/\lg\sigma' \rfloor$ characters for some suitable bit-width b. This makes sense, for example, when there are unused bits already reserved in the binary representation of the characters, as with DNA ($\sigma' = 4$) stored in bytes ($b = 4$). This allows prefix doubling algorithms to skip the first $\lfloor \lg(\lfloor b/\lg\sigma' \rfloor) \rfloor$ iterations. Dementiev et al. [56] also generalize prefix doubling to α-tupling, i.e., considering α^k-ranks during the $(k+1)$-th iteration and present experimental results for their implementations. Here, EM DC3 is superior to all prefix doubling/quadrupling ($\alpha = 2$ and $\alpha = 4$) algorithms w.r.t. running time and I/Os. They also show that for small alphabets, DCX can yield further improvements when using difference covers of size 31.

Induced sorting (see [144] for a detailed overview) is another prominent approach for EM suffix sorting. It is also used in the fastest sequential main memory suffix sorting algorithms [19 SPP] that are called SAIS [164] and DivSufSort[3]. This technique has also been generalized to compute the *SA* of collections of strings [145]. The general idea of all EM induced sorting suffix sorting algorithms is to: (1) classify all suffixes into two classes, which can be done in a single scan of the text, (2) sort at most $n/2$ special suffixes, which are suffixes from one of the classes that are (in text order) next to a suffix from the other class, and (3) induce the lexicographical order of all other suffixes using an EM priority queue. The two most prominently used classification schemes are by Itoh and Tanaka [111] and Nong et al. [164]. All following external memory algorithms make use of the latter classification scheme.

Bingmann et al. [31] propose *eSAIS* following the ideas described above. Additionally, eSAIS can also be used to compute the *LCP* array, which we define later in this section. Another EM induced sorting algorithm *DSAIS* is presented by Nong et al. [163]. However, this algorithm assumes that $n = \mathcal{O}(M^2/B)$, which limits the scalability, as the input size is still bounded by the size of the main memory (it is also not faster in practice than eSAIS [122]). An improved version *DSAIS+* by Wu et al. [206] is reported to be faster than eSAIS and also requires around half the disk space. Another EM induced sorting algorithm, called *fSAIS*, is presented by Kärkkäinen et al. [122]. The fSAIS algorithm introduces multiple improvements compared with eSAIS and DSAIS. First, it uses the classification by Nong et al. [164] but switches the classes when it comes to determining the special class, which resolves some corner cases, because now the last suffix $T[n-1..n)$ cannot be in the special class. Then, a stable priority queue is used, making timestamps to keep track of the order of the induced suffices unnecessary (compared to eSAIS) and thus reducing the I/O volume. Finally, to avoid random access on the text, a simplified *blockwise preinducing* [163] is used, i.e., the text is split into fixed sized blocks and the characters in each block are ordered in the same way they are accessed during the inducing phase. In addition to fewer random access, this makes it unnecessary to store the text positions from which the suffixes is induced. All these improvements halve the I/O volume of the algorithm compared to eSAIS. Han et al. [103] recently presented *nSAIS*, which reduces the I/O volume and required disk space even further.

Another idea for EM suffix sorting is to split the text into consecutive blocks such that the *SA* of the block can be computed in main memory. These *partial SA*s (plus additional information that helps later on) are then merged to obtain the final *SA* [117]. This approach can be parallelized [121] in EM.

Shared Memory and GPGPU. On a PRAM, we are only aware of induced sorting algorithms. Labeit et al. [132] present a parallel implementation of DivSufSort. Lao et

[3] Original implementation without publication: https://github.com/y-256/libdivsufsort (last accessed 2020-10-18). Fischer and Kurpicz give a detailed description of the algorithm and extend it to also compute the *LCP* array [77 SPP].

al. present a parallel version of SAIS [136] and SACAK [137], the latter being a simplified version of SAIS. Both are faster on repetitive texts than the parallel DivSufSort. An improved parallel SACAK algorithm, by Xie et al. [208], is the fastest algorithm on most inputs (in their evaluation, the parallel DivSufSort is only faster on two of the non-repetitive inputs).

Finally, we also want to mention *SA* construction using graphics cards (general purpose computation on graphics processing unit, GPGPU). Due to the limited amount of memory available on graphics cards, these algorithms do not scale well. The dominant techniques used in GPGPUs are prefix doubling: either heavily relying on prefix sums [195] or using radix sort [169, 202]. DCX algorithms have been presented by Deo and Keely [59] and Wang et al. [202] but are outperformed in practice by the prefix doubling approaches. The latter also present a DCX-prefix-doubling hybrid, which is the fastest GPGPU suffix sorting algorithm.

Distributed Memory. In distributed memory, suffix sorting becomes harder than in RAM, as we have to communicate to obtain access to text that is not locally available on a PE; we want to avoid random access on data that is not local. There exist distributed suffix sorting algorithms that are based on merge-sort [128], quicksort [161], and radix sort [2, 88]. The DCX algorithm has also been practically evaluated in distributed memory [131, 155, 32 SPP][4].

In practice, variants of prefix doubling are most often used, with different implementations of how the new ranks are computed. Kitajima and Navarro [127] presented an early distributed version of Manber and Myers's [150] prefix doubling algorithm, but it requires a lot of bookkeeping. Flick and Aluru's distributed prefix doubling algorithm [81] makes use of the inverse *SA* that is partly computed based on the currently considered h-ranks. A further practical improvement is that the algorithm switches to a different strategy for refining the ranks for small groups of suffixes with the same rank; this reduces communication even further. In addition, this algorithm is the only distributed algorithms that supports the computation of the *LCP* array. Two distributed prefix doubling algorithm are presented by Bingmann et al. [32 SPP]. Those algorithms have been implemented in the Thrill framework [29 SPP], which results in some restrictions regarding the access to the distributed data. The first algorithm makes use of a *window* of size 2^k (in the k-th iteration) to obtain the required rank, whereas the second one is a prefix doubling with *discarding* algorithm. This idea was later revisited and implemented using MPI [78 SPP]. Here, the prefix doubling algorithm and distributed string sorting (see Sect. 2.2) are used as building blocks for a distributed *induced sorting* suffix sorting algorithm, which is the most memory efficient distributed suffix sorting algorithms currently available, but only works efficiently for small alphabets due to a σ^2-factor in space and the number of synchronization steps.

Longest Common Prefix Array. The *SA* is often accompanied by different arrays containing useful information to speed up different types of queries. One of the most

[4] DC3/7/13 implementation without publication is available at https://github.com/bingmann/pDCX (last accessed 2020-09-25).

important ones is the *longest common prefix* (*LCP*) array. It contains the lengths of the longest common prefixes of lexicographically consecutive suffixes. More formally, $LCP[0] = 0$ and $LCP[i] = \max\{\ell \geq 0\colon\, T[SA[i], SA[i]+\ell) = T[SA[i-1], SA[i-1]+\ell)\}$, see Fig. 2. The *LCP* array can be computed sequentially in linear time [125].

There exist *LCP* array construction algorithms based on prefix doubling in distributed memory [81]. In external memory, the *LCP* array can be constructed while executing eSAIS [31]. Alternatively, it can be computed after the computation of the *SA* [115,116]. This EM computation can also be parallelized [118,119]). In GPGPUs, there exists a parallel version of Kasai et al.'s [125] algorithm [59]. We refer to [183] for a extensive evaluation of different shared memory *LCP* array construction algorithms. The *LCP* array construction has also been generalized to collections of strings [67,145].

3.2 Compressed Full-Text Index

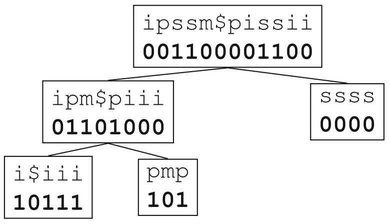
In the following, we consider a space-efficient alternative to the *SA*, the FM-index. We first look at the construction of its two main building blocks, the Burrows-Wheeler transform and the wavelet tree, and then how it can be combined to finally obtain the FM-index.

Burrows-Wheeler Transform. The *Burrows-Wheeler transform* (*BWT*) [42] of a text T of length n is defined by $BWT[i] = T[SA[i] - 1 \mod n]$. A different, more verbatim definition of the *BWT* is that we sort the strings $S_0 = T[0] \dots T[n-1], S_1 = T[1] \dots T[n-1]T[0], \dots, S_{n-1} = T[n-1]T[0] \dots T[n-2]$ (the *shifts* of T) lexicographically. Then *BWT* is the last character of each of the shifts, when the shifts are read in lexicographic order. We call this the *naive* approach. See Fig. 3a for an example of the *BWT*. The first definition of the *BWT* can be translated to a simple construction algorithm based on the *SA*—for which we have seen many construction algorithms in different models of computation in Sect. 3.1. However, there are many algorithms that do not require the computation of the *SA*. In RAM, the best main memory algorithm can compute the *BWT* in time $\mathcal{O}(n \lg \sigma / \sqrt{\lg n})$ for alphabets of size $\sigma \leq \sqrt{\lg n}$ [126].

On a PRAM, Hayashi and Taura [104] present a construction algorithm that is based on the divide-and-conquer paradigm. They first recursively split the text into consecutive slices (until the size of a slice falls below a threshold). After that, *partial BWT*s are computed for the slices. These partial *BWT*s are then merged in parallel. To speed up merging, additional information, like *SA* samples and *WT*s, is used. Liu et al. [143] present an algorithm that does not merge the partial *BWT*s directly, but only computes partial *SA*s and merges those. However, unlike Hayashi and Taura, they use a single dedicated PE to merge the partial *SA*s, which are computed by all other PEs. Again, additional information, like the *LCP* array (see Sect. 3.1), is used. The *BWT* is then obtained using the final *SA*. Fuentes-Sepúlveda et al. [86] present a parallel version of [157] that considers consecutive slices of size $\Delta = \lceil \lg_\sigma n \rceil$ as meta-symbols. The *SA* of the concatenation of S_1 and S_2 (of size $2n/\Delta$) is used to compute a partial *BWT*. Then, all other shifts S_i ($\Delta - 2$ many) are merged (each in parallel) with S_1. Additional information obtained by the merging is used to update the partial *BWT*.

	0	1	2	3	4	5	6	7	8	9	10	11
T	m	i	s	s	i	s	s	i	p	p	i	$
SA	11	10	7	4	1	0	9	8	6	3	5	2
BWT	i	p	s	s	m	$	p	i	s	s	i	i

(a) Burrows-Wheeler transform.



(b) Wavelet Tree.

α	code
$	000
i	001
m	010
p	011
s	100

(c) Binary representation.

Fig. 3. Burrows-Wheeler transform of the text $T =$ mississippi$ in (a). In (b), we show the wavelet tree (assuming $\sigma = 8$) of the Burrows-Wheeler transform depicted in (a). The binary representation of the characters is given in (c). Together, the Burrows-Wheeler transform and the wavelet tree are the FM-index, which we briefly describe in Sect. 3.2.

Ohlebusch et al. [167] consider the *reverse BWT* (BWT^{rev}), i.e., the BWT of the reverse text $T^{\text{rev}} = T[n-1]T[n-2]\dots T[0]$ that is of interest for short read mapping (cf. Sect. 4). The sequential version of the algorithm makes use of the wavelet tree of the BWT of the text, the SA, and the text itself. This leads to independent intervals in BWT^{rev} that can easily be computed in parallel. Gilchrist and Cuhadar [93] show that for many applications (cf. Sect. 4.2), the BWT is only required for slices of the text. The BWT construction for independent slices of the text is easy to parallelize.

Menon et al. [153] give a distributed BWT construction algorithm based on MapReduce. Another distributed algorithm based on merging is presented by Wang et al. [201]. This algorithm is tuned for large collection of DNA reads and first partitions the text with respect to a common prefix, and then computing the BWT for partitions with a common prefix—similar to the domain decomposition for wavelet trees (cf. Sect. 3.2). Ferragina et al. [72] present an EM version of [104] that is based on merging. Also in EM, *prefix free parsing* [40, 130] is used, which is a technique similar to the one used for the asymptotically best sequential BWT construction algorithm [126]. The naive BWT construction has also been parallelized with FPGAs by Trinidad et al. [198]. Here, the disadvantage is that we actually have to store all shifts S_i. This approach is also considered on GPGPUs by Patel et al. [170].

As with SA and the LCP array, the BWT has also been generalized for a collection of strings and there exist external memory algorithms for its construction [67, 145].

Wavelet Trees. For our compressed full-text index, we need to answer generalized rank and select queries (see Sect. 2.2) on the BWT efficiently. The *wavelet tree* (WT), introduced by Grossi et al. [99], is a binary tree data structure that allows answering both queries in time $\mathcal{O}(\lg \sigma)$ and can be stored in $n\lceil \lg \sigma \rceil(1+o(1))$ bits of memory. Each node of the tree represents an interval $[a,b] \subseteq \Sigma$ and is labeled by a bit vector that contains one bit for each text position i, in text order, where $T[i] \in [a,b]$. The bit is set iff $T[i] > \lfloor (a+b)/2 \rfloor$. The root node represents the entire alphabet $[a,b] = \Sigma$ and is therefore labeled by n bits, corresponding to the entire text T. A node has two children if $|[a,b]| \geq 2$. Then, recursively, the left child represents the interval $[a, \lfloor (a+b)/2 \rfloor]$, and the right child represents $[\lfloor (a+b)/2 \rfloor + 1, b]$. Finally, the leaves represent intervals of size one or two. Because the alphabet is split in two halves at every node, the tree has height $\lceil \lg \sigma \rceil$. Figure 3b shows an example.

Instead of comparing a character to the interval's middle to determine its bit in a node, it is more common to look at the $\lceil \lg \sigma \rceil$ bits of the characters' binary representations, starting with the most significant bit. Each bit tells whether to go left (zero) or right (one), i.e., characters encode a path down the WT starting from the root. In that regard, different codes can be used. A prominent example for using a code other than binary is the *Huffman-shaped* WT, which is constructed based on the characters' canonical Huffman codes. The bit vectors labelling the nodes then require only as much space as the Huffman-compressed text.

Apart from text indexing, the WT has applications in more areas, as described in various surveys on the topic [73, 100, 149, 160]. An alternative representation of the WT—the wavelet *matrix* (WM)—introduced by Claude et al. [47], is a more efficient choice when dealing with large alphabets. It only requires negligible extra space compared to the WT and can be used to answer the same queries in the same asymptotic time. However, when answering queries, fewer constant-time binary rank queries are needed on the bit vectors than in the WT, making it faster in practice. The similarities and differences between WT and WM are studied in more in detail by Dinklage [61]. The remainder of this section focuses on algorithms to construct the WT in the computational models introduced in Sect. 2.1. We will refer to *levels* of the WT, where level ℓ describes the set of nodes with depth ℓ.

We first consider *sequential* construction algorithms. There are various improvements to naïve algorithms to construct the WT: Claude et al. [48] and Tischler [197] give the most space-efficient algorithms using only $\mathcal{O}(\lg n)$ bits, but do not provide a competitive implementation. Da Fonseca and da Silva [84] give an *online* construction algorithm, i.e., one where no prior knowledge of the input alphabet is required, that runs in time $\mathcal{O}(n \lg \sigma)$ and uses $n\lceil \lg \sigma \rceil + o(n \lg \sigma)$ bits of space. The fastest known algorithms in theory require time $\mathcal{O}(n \lg \sigma / \sqrt{\lg n})$ and were given by Babenko et al. [18] and Munro et al. [158]. The latter was implemented by Kaneta [114], proving that the use of modern CPU instructions can reflect theoretical improvements also in practice. Kaneta's results are competitive with the currently known fastest and most space-efficient algorithm to construct the WT, which has been developed by Fischer et al. [79 SPP]: it is based on *prefix counting* and, except for the topmost level, constructs the WT bottom-up as described in the following. In a first scan of T, we compute the histogram of T, i.e., the frequencies of all characters, as well as the topmost level of the

WT, which consists of the characters' most significant bits in the same order in which they occur in T. For each remaining level $\ell \in [2, \lceil \lg \sigma \rceil)$, starting with the bottommost level, we first compute the histogram of T. This is done by combining the frequencies of every pair in the previous histogram: because a node combines the two intervals of the alphabet represented by its children, the total frequency of its represented characters is the sum of the respective frequencies of its children. The histogram for level ℓ allows us to easily compute the positions of the first bit for every node on level ℓ. In one scan of T, we can then compute the bits for all nodes on level ℓ and directly write them to the correct positions. The algorithm requires total time $\mathcal{O}(n \lg \sigma)$ and $\sigma \lceil \lg n \rceil$ bits of space in addition to the input and output. The same technique can be used to construct the Huffman-shaped WT, where it also yields the best practical results in terms of speed and space usage.

We now regard the *parallel* WT construction in the *shared memory* model. Labeit et al. [132] gave a recursive algorithm based on the *parallel split* operation. Here, the available PEs process T in parallel to compute the bits for the root node. These bits are then used to perform a parallel split of T for the left and right child, which are recursively processed in parallel. The number of PEs used to process each child is proportional to the sizes of the children. Two further techniques for parallel WT construction stand out: *domain decomposition* and an algorithm based on *sorting*. The use of domain decomposition for WT construction has first been proposed by Sepúlveda et al. [85]. The input T is partitioned such that every PE receives a slice of size n/p. and computes the entire WT for its slice using any sequential algorithm, e.g., prefix counting. In a subsequent step, these WT are merged into the WT for T, which can be done efficiently by concatenating the bit vectors contained in the corresponding nodes. Because an arbitrary sequential construction algorithm can be used locally, domain decomposition can be tuned to have a very low memory footprint. The algorithm based on sorting, first proposed by Shun [184], constructs the WT top-down, level by level, and makes use of stable integer sorters, which are well studied for all practically relevant computational models. The bits of the topmost level can be computed in an initial parallel scan of T, similar to the (sequential) prefix counting algorithm. Then, before proceeding to some level $\ell > 1$, the text is reordered by stably sorting the characters according to their ℓ-bit prefixes, which puts them in the correct positions to compute that level's bit vector in a parallel scan of the reordered text. To that end, the algorithm only requires $\lceil \lg \sigma \rceil$ parallel scans of T. For both algorithms, Shun presents techniques that allow for different trade-offs between work and time [185]. The best known implementations were given by Fischer et al. [79 SPP], concluding that domain decomposition is the fastest approach in practice, also for constructing the Huffman-shaped WT.

The parallel construction in *distributed memory* has been studied by Dinklage et al. [64 SPP], confirming the practical relevance of domain decomposition, which yields the fastest running times and best memory efficiency in practice. An important measure for distributed memory algorithms is the communication volume. During the distributed domain decomposition, only the merging phase requires communication between the PEs. They also adapted Shun's parallel sorting algorithm [184] to distributed memory and achieved nearly as good running times, albeit requiring more communication. Because the individually constructed levels need not be partitioned into nodes, the sorting algorithm has furthermore been found to be better suited than domain decomposition for constructing the WT for large alphabets.

Finally, we look at WT construction in *external memory*. Ellert and Kurpicz [69 SPP] present sequential and parallel external memory algorithms. The sequential algorithm is based on sorting and works similar to the corresponding parallel algorithm. Using only a constant amount of main memory, it requires two scans of T for each level of the WT. They also provide various semi-external algorithms with similar properties, all of which outperform the semi-external WT construction algorithms from the Succinct Data Structure Library (SDSL) [94]. Finally, their parallel algorithm makes use of domain decomposition to distribute work on the available PEs, each PE using a sequential in-memory algorithm (e.g., prefix counting) to construct a partial WT. Because the p parts of T may not fit into main memory, each PE furthermore partitions its part into segments of size k such that a segment and its WT does fit in main memory. They then process their part segment by segment. The algorithm requires four scans over T for each level, plus σ random I/O operations for each segment. Naturally, because of the necessary synchronizations with external memory, the algorithm only scales well up to a limited number of PEs. Yet, the parallelization achieves a notable speedup in practice.

FM-Index. The FM-index [74] combines the BWT and (Huffman shaped) WTs to a compressed full-text index. It is widely used, in particular in most DNA read aligners [134] and in Bioinformatics in general (cf. Sect. 4.1).

To *locate* a pattern using the FM-index, a *backward search* is performed. Using the C array (for each $\alpha \in \Sigma$, $C[\alpha]$ is the overall number of occurrences of characters in BWT that are strictly smaller than α, i.e., the rank of α in Σ) and the WT of the BWT to answer $rank_\alpha(i)$ (on the BWT, cf. Fig. 3) it is possible to search backwards for a pattern in T [74]: Given an ω-interval $[i, j]$ (i.e., ω is a prefix of $T[SA[k]..n)$ if and only if $i \leq k \leq j$) and $\alpha \in \Sigma$, the procedure called $backwardSearch(\alpha, [i, j])$ returns the $\alpha\omega$-interval $[lb, rb]$, where $lb = C[\alpha] + rank_\alpha[i] + 1$ and $rb = C[\alpha] + rank_\alpha[j+1]$. If $lb > rb$, the pattern does not occur in T.

Note that any combination of BWT and WT construction algorithms can be combined to compute the FM-index (in any model of computation). Still, there exist dedicated practical PRAM FM-index construction algorithms by Labeit et al. [132] and Lio et al. [143]. The former combines their parallel SA (see Sect. 3.1) and WT (see Sect. 3.2) construction algorithms to compute an FM-index (in parallel), whereas the latter provides a parallel algorithm that computes both the BWT and the FM-index.

3.3 Suffix Trees

A suffix tree (ST) for a string of length n is a compact trie storing all the suffixes of T, i.e., the concatenation of the edge labels on the path from the root to leaf i exactly spells out the suffix $T[i..n)$; see Fig. 4a for an example. Weiner [204] showed that it can be constructed in linear time provided that the underlying alphabet has constant size. Farach-Colton et al. [71] gave the first suffix tree construction algorithm that is optimal for all alphabets. It has linear run-time for alphabets consisting of integers in a polynomial range. The ST is one of the most powerful data structures in string processing, with applications in fields like bioinformatics or information retrieval, e.g., [21,45].

Abouelhoda et al. [3] showed that there is a one-to-one correspondence between the set of all lcp-intervals and the set of all internal nodes of the *ST* of T. Let us define the concept of lcp-intervals (see Fig. 4a). An interval $[i, j]$ in the *LCP* array—for simplicity, we now assume that $LCP[0] = -1 = LCP[n]$—is called an *lcp-interval of lcp-value* ℓ if (1) $LCP[i] < \ell$, (2) $LCP[k] \geq \ell$ for all k with $i < k \leq j$, (3) $LCP[k] = \ell$ for at least one k with $i < k \leq j$, and (4) $LCP[j+1] < \ell$. Every index k ($i < k \leq j$) with $LCP[k] = \ell$ is called *ℓ-index* or *lcp-index*. A leaf in the *ST* corresponds to a *singleton interval* $[k,k]$. The parent interval of an lcp-interval $[i, j]$ (or a singleton interval) is the smallest lcp-interval that contains $[i, j]$ but does not coincide with $[i, j]$.

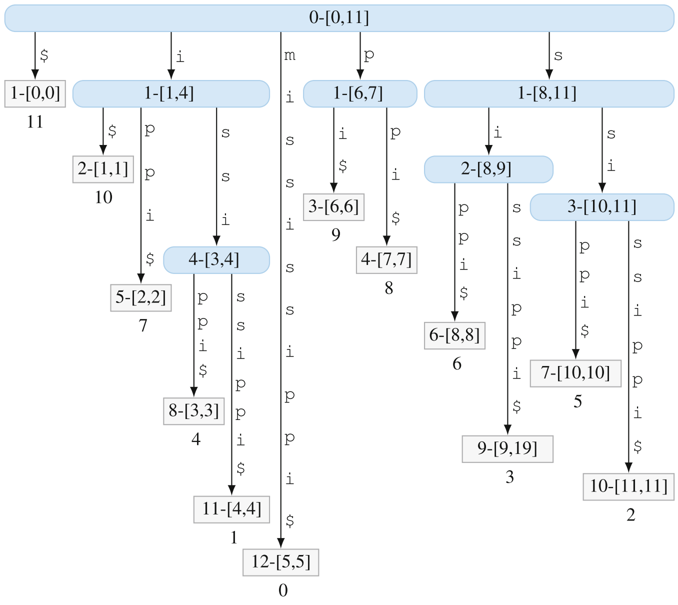
The drawback of *ST*s is their huge space consumption: even carefully engineered implementations require 8–20 bytes per input character. It is possible to save a lot of space by representing the *ST* topology by a sequence of balanced parentheses. The sequence BPS, for instance, can be constructed by a depth first search traversal of the (uncompressed) *ST* as follows. At each node v (starting at the root), write an opening parenthesis, recursively process the child nodes of v, and write a closing parenthesis afterwards (see Fig. 4b). Since the *ST* has n leaves and up to $n-1$ internal nodes, the BPS needs up to $4n-2$ bits. Based on the BPS, all navigational operations on the *ST* can be supported with data structures that require only $o(n)$ bits [177].

The BPS can be constructed in parallel on a shared memory architecture in the CRCW PRAM model with the help of the *LCP* array as follows; see [22] for details, where also the necessary adjustments for the CREW model are explained. Create two arrays C_o and C_c of size n, enumerate all lcp-intervals in parallel, and increment $C_o[i]$ and $C_c[j]$ for each lcp-interval $[i, j]$. After that, compute the prefix sum *PS* of $sum[i+1] = C_o[i] + C_c[i]$, and write $C_o[i]$ opening followed by $C_c[i]$ closing parenthesis at position $PS[i]$ into the bitvector BPS (in parallel). It is possible to enumerate all lcp-intervals (in parallel) with the help of the arrays *PSV* (previous smaller value) and *NSV* (next smaller value), which are defined as follows:

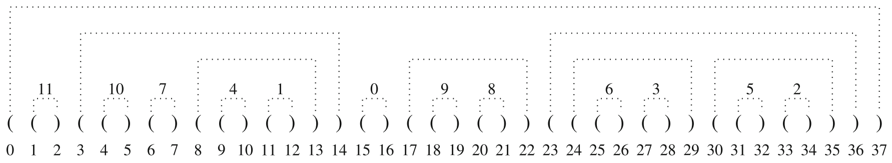
$$PSV[i] = \max\{j \mid 0 \leq j < i \text{ and } LCP[j] < LCP[i]\}$$
$$NSV[i] = \min\{j \mid i < j \leq n \text{ and } LCP[j] < LCP[i]\}$$

Table 1 shows an example (an entry $\perp$ means that the value is undefined). The key observation is that for any index i with $0 < i < n$ and $LCP[i] = \ell$ the interval $[PSV[i], NSV[i]-1]$ is an lcp-interval of lcp-value ℓ and i is one of its lcp-indices; for a proof see, e.g., [166, Lemma 4.3.8]. A problem of this approach is that an lcp-interval with multiple ℓ-indices will occur more than once in the enumeration. To overcome this problem, such an interval is reported if and only if i is the first (left-most) ℓ-index of the interval. To this end, previous smaller values (*PSV*) are replaced with previous smaller or equal values (*PSEV*), where the array *PSEV* is defined by $PSEV[i] = \max\{j \mid 0 \leq j < i \text{ and } LCP[j] \leq LCP[i]\}$. Then $[PSV[i], NSV[i]-1]$ appears in the enumeration if and only if $LCP[i] \neq LCP[PSEV[i]]$.

The problem of computing previous smaller and next smaller values, also known as the all-nearest-smaller-value problem (ANSV), was already solved by Berkman et al. [27] with $\mathcal{O}(n)$ work and in $\mathcal{O}(\log\log n)$ time using $\mathcal{O}(n/\log\log n)$ processors on a CRCW PRAM.



(**a**) Suffix tree.



(**b**) BPS of the suffix tree.

Fig. 4. In (a), the suffix tree for $T =$ `mississippi$` (an annotation ℓ-$[i, j]$ within a node shows the corresponding lcp-interval, i.e., ℓ is the string-depth of the node and $[i, j]$ is the corresponding interval). The number below the leafs is the starting position of the corresponding suffix. For the corresponding *SA* and *LCP* array see Fig. 2. The BPS of the suffix tree is shown in (b). Matching parentheses are connected by dotted lines. A leaf in the suffix tree is represented by an opening parenthesis that is immediately followed by a closing parenthesis; its leaf number (leaf i represents suffix $T[i..n]$) is depicted above the two parentheses. The last row shows the positions of the parentheses in the BPS.

In the following, we will focus on distributed memory. He and Huang [105] presented a bulk-synchronous parallel adaption of the algorithm invented by Berkman et al. [27]. Flick and Aluru [82] improved their work in various directions by introducing a generalized version of the ANSV problem. They showed how to handle duplicate values, generalized the communication structure, and provided novel proofs. Based on the improvements on the ANSV problem, they presented a parallel *ST* construction algorithm using the suffix- and *LCP* array that runs in $\mathcal{O}(n/p+p)$ time, which is work optimal for $p = \mathcal{O}(\sqrt{n})$. In a first phase, they represent the *ST* as an array *E*

Table 1. The *LCP* array with $LCP[0] = -1 = LCP[n]$ for $T =$ `mississippi$` (cf. *LCP* array in Fig. 2) and the corresponding arrays *NSV*, *PSV*, *PSEV*, and *PFE*.

	0	1	2	3	4	5	6	7	8	9	10	11	12
LCP	−1	0	1	1	4	0	0	1	0	2	1	3	−1
NSV	⊥	12	5	5	5	12	12	8	12	10	12	12	⊥
PSV	⊥	0	1	1	3	0	0	6	0	8	8	10	⊥
PSEV	⊥	0	1	2	3	1	5	6	6	8	8	10	⊥
PFE	⊥	⊥	1	2	2	1	1	1	1	1	1	10	⊥

of edges $(i, parent(i))$. This approach requires a unique representative index for each node v in the *ST*. Since v corresponds to an lcp-interval, one can choose the first (leftmost) lcp-index of that lcp-interval as a representative. Moreover, Lemma 1 shows that the representative of the parent interval can be computed with the help of "previous-furthest-equal" values, defined for all i with $1 < i < n$ as follows:

$$PFE[i] = \min\{j \mid PSV[k] < j < i \text{ and } LCP[j] = LCP[k], \text{ where } k = PSEV[LCP[i]]\}$$

Lemma 1. *Recall that, for any index i with $1 < i < n$, the interval $[lb, rb]$, where $lb = PSV[i]$ and $rb = NSV[i] - 1$, is an lcp-interval and i is an lcp-index of $[lb, rb]$. In the following, let $m = PFE[i]$. If $LCP[m] = LCP[i]$, then m is the representative lcp-index and i is a different lcp-index of $[lb, rb]$. From now on we assume $LCP[m] < LCP[i]$. In this case, we have $LCP[m] = LCP[PSV[i]]$. If $LCP[PSV[i]] < LCP[NSV[i]]$, then $NSV[i]$ is the representative lcp-index of the parent interval of $[lb, rb]$; see [166, Lemma 4.3.9]. Otherwise, $PSV[i]$ is an lcp-index of the parent interval of $[lb, rb]$ and m is the representative lcp-index of that parent.*

Flick and Aluru's algorithm assumes that all inputs are distributed equally across processors with n/p elements per process. It computes *PFE* and *NSV* in $\mathcal{O}(n/p + p)$ time. Since the processor for the range $[\frac{n}{p}j, \frac{n}{p}(j+1) - 1]$ has the corresponding portions of *LCP*, *PFE*, and *NSV* in local memory, it can compute edges $(i, parent(i))$ in its range based on Lemma 1. The parents of leaf nodes in its range can be computed similarly; see [82] for details. In the second phase of their algorithm, Flick and Aluru show how edges can be inverted (and analyse the communication complexity). This is because for pattern matching applications, instead of having parent pointers, each internal node should point to its children. Other algorithms for distributed *ST* include [44,50,212].

3.4 Query Answering

Up to this point, we only have considered the construction of different full-text indices. Since all full-text indices that we have looked at have their origin in RAM, they can easily be used there by allocating the incoming queries in a round robin fashion to the PEs. However, in external or distributed memory, the obstacle is that neither the whole text nor the whole index can be accessed in a random access manner, as in the construction algorithms. In this section, we take a look at different approaches to answer queries in such a setting.

Clifford [49] show how to use a suffix tree in distributed memory to answer different types of queries. They build the suffix tree using Ukkonen's algorithm [199]. For this purpose, the whole text is required at each PE, limiting the scalability of this approach significantly.

Mäkinen et al. [148] use the *compressed* suffix array (CSA) [176] in distributed and external memory. The CSA requires roughly the same space as the *compressed* text but also does not need the text to answer queries (unlike the *SA*); it is a *self-index*. In main memory, queries of length m can be answered in $\mathcal{O}(m \lg n)$ time. They improve query times by sampling ℓ-length strings instead of characters and encoding the supporting data structures using Elias delta encoding in combination with lookup tables. This allow for constant time access to the supporting data structures (not queries). In EM, their approach can search for a pattern of length m in $\mathcal{O}(m \lg_B n)$ I/Os (which can be reduced to $\mathcal{O}((m \lg n)/B)$ if $\mathcal{O}(n)$ bits can be stored in main memory). In distributed memory, m supersteps are required to answer such a query. During each superstep only a constant number of words have to be communicated and $\mathcal{O}(\lg n)$ local work is required.

Arroyuelo et al. [10] compare different layouts of the *SA* for pattern matching in distributed memory. When each PE holds a consecutive slice of the *SA*, we have the *global* layout. In addition to the *SA*, pruned suffixes are stored to speed-up querying at the local PE. To speed up queries, a trie for the suffixes at the beginning and end of each slice is built at each PE in order to distribute queries to the PE that can answer it locally. Next, in the *local* layout, each PE holds a consecutive slice of the text and builds a *SA* only for this local slice. Here, each PE must answer the query locally and return the result, requiring only a constant number of supersteps but significant local work (as all PEs always have to search for the query). The *multiplex* layout is an intermingled global layout, where the i-th entry of the global *SA* is stored at PE $i \mod p$ in consecutive fashion, i.e., the i-th and $(i+p)$-th entry are stored consecutively at the same PE. Corresponding pruned suffixes are stored as in the global layout. The multiplex layout (and in some cases the global layout) is the most efficient one in their experiments. They also propose two additional layouts that, however, perform not as well in practice.

The global layout is extended by Fischer et al. [80 SPP]. Instead of answering the query directly on the *SA*, a Patricia trie [156] is constructed for each local slice. To this end, the *LCP* array (see Sect. 3.1) is required. Furthermore, a global trie is used to distribute the query to the corresponding PE. These two tries together allow queries to be answered with a constant number of supersteps.

Flick and Aluru [83] further improve the above two-level designs by developing the *distributed enhanced SA* (DESA). One improvement of DESA is to eliminate the explicitly stored tree structure of the two-level indices. Also, the DESA does not partition the text into consecutive slices of the same size (where queries may have to be answered on multiple PEs) but partitions the text into more fine grained intervals, such that all intervals can be processed on a single PE. This approach currently scales best in practice. The local search is an adapted version of Fischer and Heun's [75] query algorithm for enhanced *SA*s [3]. Hence, they only need the *SA*, *LCP* array, *range minimum queries* (RMQs, returning the positions of the smallest element in a given range), and some additional information. To help load balancing queries, the top level trie is dynamically (based in the input) constructed such that each bottom level index, corresponding to a leaf in the top level trie, covers intervals of size n/cp for a constant c. To this end, the ANSV problem (cf. Sect. 3.3) is solved. This approach is significantly better than a static top level lookup table and overall the best in practice.

4 Applications

All previously described text indices have more applications than (exact) pattern matching. They can also be used to answer approximate queries, i.e., when allowing differences between the pattern and the matched positions. Pockrandt [171] shows how to transform those queries into exact queries. This is also used in practice in the SeqAn library [175], which contains efficient algorithms and data structures for the analysis of biological data (and strings in general). Furthermore, they can be used to compute succinct de Bruijn graphs, all pairs suffix-prefix overlaps, and maximal repeats [67]. In the following, we take a more detailed look into two fields where text indices are of great importance—*Bioinformatics* (Sect. 4.1) and *lossless compression* (Sect. 4.2).

4.1 Bioinformatics

The most successful application of index structures in bioinformatics is backward search based on an FM-index [74] (e.g., in form of the WT of the BWT of the input string [99], cf. Sect. 3.2). For information on k-mer-based tools, we refer to the recent survey by Marchet et al. [151].

The most important application of backward search in bioinformatics is read mapping. Ultra-high-throughput next-generation sequencing technologies (NGS) have been commercially available since 2005. In NGS, DNA is fragmented into small pieces, of which the first few bases are sequenced, yielding several millions of short "reads", each 30 to 400 base pairs ("DNA characters") long. The read mapping task is now to align these reads to a reference genome, i.e., to the known, nearly complete chromosomal DNA sequences of the organism in question (which may be up to several billion base pairs long); see [43] for an overview article.

Short read mappers like Bowtie [135] or BWA [140] must be able to deal with (sequencing) errors. Inexact matching is either based on recursive algorithms that use backtracking or on the seed-and-extend strategy (exact matches are used as seeds and the shared seeds are then extended into longer, inexact alignments). The same approach has also been successfully applied in genome assembly [187] (sequence assembly refers to aligning and merging reads in order to reconstruct the original sequence). Here, the fastest implementations only utilize a few threads [24]. Those read mappers usually do not use parallel construction algorithms, as the reference sequences are short, allowing a space-efficient sequential algorithm to compute the index in less than an hour.

Alignments of longer sequences (ranging from long read mapping to whole genome alignment) are also obtained by exact matching and the seed-and-extend method. One of the earliest tools in comparative genomics is based on suffix trees (and later on suffix arrays) [55], but there are also tools using the BWT [142]. The major principle of comparative genomics is that common features of two organisms will often be encoded within the DNA that is evolutionarily conserved between them. Therefore, comparative genomic approaches start with making some form of alignment of genome sequences. Then, they look for orthologous sequences (sequences that share a common ancestry) in the aligned genomes and check to what extent those sequences are conserved. Nowadays, one tries to take multiple genomes simultaneously into account; see [51] for an overview of pangenomics. When it comes to the alignment of longer sequences, scaling algorithms are used, e.g., multithreaded semi-external prefix-doubling algorithms [190] or building multiple partial indices (in parallel) and merging them [191]. Compressed suffix trees and FM-indices have been used in indexing variation graphs [92] and for graphical pangenome analysis [21]. In particular, the balanced parentheses sequence BPS from Sect. 3.3 was used for indexing variation graphs [190] (using the algorithm described in [168]). Using a dynamic FM-index, sequences can be inserted in batches, which can easily be parallelized [139].

4.2 Compression

Text indices have been successfully applied to text compression, most notably to compressors based on the *BWT* (see Sect. 3.2) and on different variants of the Lempel-Ziv parsing of the text. Intuitively, this link between indexing and compression seems plausible, as in both cases one tries to 'group' similar substrings; in the former for listing occurrences, in the latter for exploiting the repetitiveness to somehow save space. We only consider compressors that operate over the *full* text (*not* restricted to small sliding windows/blocks); this is important for highly repetitive texts such as DNA collections of individuals from the same species.

Lempel-Ziv in External Memory. The LZ77-factorization [213] of a text T is defined as follows: suppose $T[0..i)$ has already been parsed into LZ77-phrases. Then the next LZ77-phrase is the longest prefix of $T[i..n)$ that has an occurrence in T starting strictly before i (but possibly ending in $T[i..n)$), or a single character if $T[i]$ does not occur before. Given a text index on T, this prefix can be located by iteratively querying for $T[i..i+1)$, $T[i..i+2)$, $\ldots$, as long as an occurrence starting before i exists. In main memory, Fischer et al. [76] have the most space efficient implementation of this idea using compressed variants of the suffix tree, needing only $(1+\varepsilon)n\log n + \mathscr{O}(n)$ bits of space and running in $\mathscr{O}(n/\varepsilon)$ time. The difficulty in EM is, of course, that such repeated querying causes too many I/Os. Kärkkäinen et al. [120] avoid this in two ways: in their EM-LPF algorithm, they first compute the array of *longest previous factors* in EM, from which the LZ77 factorization is easily obtained in total sort(n) I/Os. Their second algorithms, EM-LZScan, divides T into blocks of size $\Theta(M)$ and then computes *matching statistics* [166, Sect. 5.5.4] of the current block w.r.t. the prefix of T up to the current block. EM-LZScan needs $\mathscr{O}\left(\frac{n^2\log\sigma}{BM\log n}\right)$ I/Os in theory, but is significantly faster in practice than EM-LPF for highly repetitive texts. A different approach was taken by Dinklage et al. [62 SPP], who show that the flexibility of allowing factor occurrences also be to the *right* of their starting position (so-called *bidirectional parsings*) leads to a much better throughput than EM-LZScan, while achieving similar compression rates. Their algorithm plcpcomp has been successfully applied to texts of size 128 GiB on a machine with just 16 GiB of RAM. Considering *de*compression, Belazzougui et al. [25] show the I/O-complexity to be $\text{sort}(n/\log_\sigma n)$ I/Os and also give a practical implementation; however, this algorithm cannot be applied to the bidirectional variant, which is much slower at decompression. Other variants of LZ exist, but have so far not been successfully applied to large datasets, although promising approaches exist for LZ78 that might lead to semi-external solutions [11].

Parallel Burrows-Wheeler-Based Compression. In Sect. 3.2, we already mentioned the relevant literature for computing the *BWT* L. This output can be postprocessed to compute a compressed version of T, as characters following a similar preceding context are grouped in L. The postprocessing consists of computing the move-to-front numbers when processing L from left to right, followed by a Huffman encoding of the resulting numbers. On the PRAM, Edwards and Vishkin [66] show how to perform those latter steps in $\mathscr{O}(\log n)$ parallel time and $\mathscr{O}(n)$ work, and report good speedups on FPGA-hardware over popular tools such as bzip2, although only using moderately-sized inputs. They also show how to decompress the resulting file within the same complexities. At their core, the algorithms are reduced to the building blocks prefix sums (cf. Sect. 2.2) and list ranking. Geared more towards practice, Patel et al. [170] have similar ideas and show GPGPU implementations; however, they use mergesort for computing the *BWT* and report this as their main bottleneck.

We are not aware of any algorithms in external or distributed memory implementing the full *BWT* compression pipeline, despite that algorithms for computing the *BWT* exist in these models of computation (see Sect. 3.2).

5 Conclusion and Future Work

Advanced text index data structures such as suffix trees, suffix arrays and wavelet trees are key to handling large data sets in a range of important applications. A combination of parallel, external, and compressed implementations can approach the requirements for handling the exploding amounts of available data.

In this short survey, we have discussed a number of techniques for building and using such data structures. Our impression is that memory hierarchies and compression by themselves are fairly well understood by now. A range of parallelization approaches are known but they suffer from a tradeoff between asymptotic scalability and efficiency. In particular, the most efficient sequential and external techniques are inherently sequential. Hence, a number of important open problems remain. These involve highly scalable techniques with good constant factors (e.g., for constructing suffix arrays and *LCP* arrays with linear work) as well as integration of parallelism, memory hierarchies, compression and applications. Another interesting research direction is to engineer recent text indices for highly repetitive data [91] for handling large texts. In principle, big data frameworks such as Thrill [29 SPP] can handle parallelization and memory hierarchies automatically but the question remains whether the involved overheads are acceptable.

References

1. Abali, B., Özgüner, F., Bataineh, A.: Balanced parallel sort on hypercube multiprocessors. IEEE Trans. Parallel Distrib. Syst. **4**(5), 572–581 (1993). https://doi.org/10.1109/71.224220
2. Abdelhadi, A., Kandil, A.H., Abouelhoda, M.: Cloud-based parallel suffix array construction based on MPI. In: MECBME, pp. 334–337 (2014). https://doi.org/10.1109/MECBME.2014.6783271
3. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms **2**(1), 53–86 (2004). https://doi.org/10.1016/S1570-8667(03)00065-0
4. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM **31**(9), 1116–1127 (1988). https://doi.org/10.1145/48529.48535
5. Alexandrov, A., et al.: The stratosphere platform for big data analytics. VLDB J. **23**(6), 939–964 (2014). https://doi.org/10.1007/s00778-014-0357-y
6. Amarasinghe, S., et al.: Exascale software study: software challenges in extreme scale systems. DARPA IPTO, Air Force Research Labs, Technical report, pp. 1–153 (2009)
7. Arge, L., Ferragina, P., Grossi, R., Vitter, J.S.: On sorting strings in external memory (extended abstract). In: STOC, pp. 540–548. ACM (1997). https://doi.org/10.1145/258533.258647
8. Arge, L., Procopiuc, O., Scott Vitter, J.: Implementing I/O-efficient data structures using TPIE. In: Möhring, R., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 88–100. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45749-6_12
9. Arge, L., Rav, M., Svendsen, S.C., Truelsen, J.: External memory pipelining made easy with TPIE. In: BigData, pp. 319–324. IEEE Computer Society (2017). https://doi.org/10.1109/BigData.2017.8257940
10. Arroyuelo, D., Bonacic, C., Costa, V.G., Marín, M., Navarro, G.: Distributed text search using suffix arrays. Parallel Comput. **40**(9), 471–495 (2014). https://doi.org/10.1016/j.parco.2014.06.007
11. Arroyuelo, D., Cánovas, R., Navarro, G., Raman, R.: LZ78 compression in low main memory space. In: Fici, G., Sciortino, M., Venturini, R. (eds.) SPIRE 2017. LNCS, vol. 10508, pp. 38–50. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67428-5_4
12. Aumüller, M., Dietzfelbinger, M.: Optimal partitioning for dual-pivot quicksort. ACM Trans. Algorithms **12**(2), 18:1–18:36 (2016). https://doi.org/10.1145/2743020
13. Axtmann, M.: Robust Scalable Sorting. Ph.D. thesis, Karlsruhe Institute of Technology, Germany (2021). https://doi.org/10.5445/IR/1000136621

14 SPP. Axtmann, M., Bingmann, T., Sanders, P., Schulz, C.: Practical massively parallel sorting. In: SPAA, pp. 13–23. ACM (2015). https://doi.org/10.1145/2755573.2755595

15. Axtmann, M., Sanders, P.: Robust massively parallel sorting. In: ALENEX, pp. 83–97. SIAM (2017). https://doi.org/10.1137/1.9781611974768.7
16. Axtmann, M., Wiebigke, A., Sanders, P.: Lightweight MPI communicators with applications to perfectly balanced quicksort. In: IPDPS, pp. 254–265. IEEE Computer Society (2018). https://doi.org/10.1109/IPDPS.2018.00035
17. Axtmann, M., Witt, S., Ferizovic, D., Sanders, P.: Engineering in-place (shared-memory) sorting algorithms. ACM Trans. Parallel Comput. **9**(1), 2:1–2:62 (2022). https://doi.org/10.1145/3505286
18. Babenko, M.A., Gawrychowski, P., Kociumaka, T., Starikovskaya, T.: Wavelet trees meet suffix trees. In: SODA, pp. 572–591. SIAM (2015). https://doi.org/10.1137/1.9781611973730.39

19 SPP. Bahne, J., et al.: SACABench: benchmarking suffix array construction. In: Brisaboa, N.R., Puglisi, S.J. (eds.) SPIRE 2019. LNCS, vol. 11811, pp. 407–416. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32686-9_29

20. Baier, U.: linear-time suffix sorting - a new approach for suffix array construction. In: CPM, pp. 23:1–23:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.CPM.2016.23

21. Baier, U., Beller, T., Ohlebusch, E.: Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler transform. Bioinformatics **32**(4), 497–504 (2016). https://doi.org/10.1093/bioinformatics/btv603

22. Baier, U., Beller, T., Ohlebusch, E.: Space-efficient parallel construction of succinct representations of suffix tree topologies. ACM J. Exp. Algorithmics **22** (2017). https://doi.org/10.1145/3035540

23. Barsky, M., Stege, U., Thomo, A.: A survey of practical algorithms for suffix tree construction in external memory. Softw. Pract. Exp. **40**(11), 965–988 (2010). https://doi.org/10.1002/spe.960

24. Bauer, M.J., Cox, A.J., Rosone, G.: Lightweight algorithms for constructing and inverting the BWT of string collections. Theor. Comput. Sci. **483**, 134–148 (2013). https://doi.org/10.1016/j.tcs.2012.02.002

25. Belazzougui, D., Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Lempel-Ziv decoding in external memory. In: Goldberg, A.V., Kulikov, A.S. (eds.) SEA 2016. LNCS, vol. 9685, pp. 63–74. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-38851-9_5

26. Bentley, J.L., Sedgewick, R.: Fast algorithms for sorting and searching strings. In: SODA, pp. 360–369. ACM/SIAM (1997)

27. Berkman, O., Schieber, B., Vishkin, U.: Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. J. Algorithms **14**(3), 344–370 (1993). https://doi.org/10.1006/jagm.1993.1018

28. Bingmann, T.: Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools. Ph.D. thesis, Karlsruhe Institute of Technology, Germany (2018). https://doi.org/10.5445/IR/1000085031

29 SPP. Bingmann, T., et al.: Thrill: high-performance algorithmic distributed batch data processing with C++. In: BigData, pp. 172–183. IEEE Computer Society (2016). https://doi.org/10.1109/BigData.2016.7840603

30 SPP. Bingmann, T., Eberle, A., Sanders, P.: Engineering parallel string sorting. Algorithmica **77**(1), 235–286 (2017). https://doi.org/10.1007/s00453-015-0071-1

31. Bingmann, T., Fischer, J., Osipov, V.: Inducing suffix and LCP arrays in external memory. ACM J. Exp. Algorithmics **21**(1), 2.3:1–2.3:27 (2016). https://doi.org/10.1145/2975593

32 SPP. Bingmann, T., Gog, S., Kurpicz, F.: Scalable construction of text indexes with thrill. In: BigData, pp. 634–643. IEEE (2018). https://doi.org/10.1109/BigData.2018.8622171

33. Bingmann, T., Sanders, P.: Parallel string sample sort. In: Bodlaender, H.L., Italiano, G.F. (eds.) ESA 2013. LNCS, vol. 8125, pp. 169–180. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40450-4_15

34 SPP. Bingmann, T., Sanders, P., Schimek, M.: Communication-efficient string sorting. In: IPDPS, pp. 137–147. IEEE (2020). https://doi.org/10.1109/IPDPS47924.2020.00024

35. Blacher, M., Giesen, J., Sanders, P., Wassenberg, J.: Vectorized and performance-portable quicksort. CoRR abs/2205.05982 (2022)

36. Blelloch, G.E., Anderson, D., Dhulipala, L.: Parlaylib - A toolkit for parallel algorithms on shared-memory multicore machines. In: SPAA, pp. 507–509. ACM (2020). https://doi.org/10.1145/3350755.3400254

37. Blelloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J., Zagha, M.: A comparison of sorting algorithms for the connection machine CM-2. Commun. ACM **39**(12es), 273–297 (1996)
38. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. J. Parallel Distributed Comput. **37**(1), 55–69 (1996). https://doi.org/10.1006/jpdc.1996.0107
39. Borkar, S.: Exascale computing - A fact or a fiction? In: IPDPS, p. 3. IEEE Computer Society (2013). https://doi.org/10.1109/IPDPS.2013.121
40. Boucher, C., Gagie, T., Kuhnle, A., Langmead, B., Manzini, G., Mun, T.: Prefix-free parsing for building big BWTs. Algorithms Mol. Biol. **14**(1), 13:1–13:15 (2019). https://doi.org/10.1186/s13015-019-0148-5
41. Bramas, B.: A novel hybrid quicksort algorithm vectorized using AVX-512 on intel skylake. Int. J. Adv. Comput. Sci. Appl. **8**(10) (2017). arXiv:1704.08579
42. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm, Technical report (1994)
43. Canzar, S., Salzberg, S.L.: Short read mapping: an algorithmic tour. Proc. IEEE **105**(3), 436–458 (2017). https://doi.org/10.1109/JPROC.2015.2455551
44. Chen, C., Schmidt, B.: Constructing large suffix trees on a computational grid. J. Parallel Distributed Comput. **66**(12), 1512–1523 (2006). https://doi.org/10.1016/j.jpdc.2006.08.004
45. Chim, H., Deng, X.: A new suffix tree similarity measure for document clustering. In: WWW, pp. 121–130. ACM (2007). https://doi.org/10.1145/1242572.1242590
46. Claude, F., Navarro, G.: Practical rank/select queries over arbitrary sequences. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 176–187. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89097-3_18
47. Claude, F., Navarro, G., Pereira, A.O.: The wavelet matrix: an efficient wavelet tree for large alphabets. Inf. Syst. **47**, 15–32 (2015). https://doi.org/10.1016/j.is.2014.06.002
48. Claude, F., Nicholson, P.K., Seco, D.: Space efficient wavelet tree construction. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 185–196. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24583-1_19
49. Clifford, R.: Distributed suffix trees. J. Discrete Algorithms **3**(2–4), 176–197 (2005). https://doi.org/10.1016/j.jda.2004.08.004
50. Clifford, R., Sergot, M.: Distributed and paged suffix trees for large genetic databases. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 70–82. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44888-8_6
51. Dutilh, B.E.: Consortium: Computational pan-genomics: status, promises and challenges. Briefings Bioinform. **19**(1), 118–135 (2018). https://doi.org/10.1093/bib/bbw089
52. Crauser, A., Ferragina, P.: A theoretical and experimental study on the construction of suffix arrays in external memory. Algorithmica **32**(1), 1–35 (2002). https://doi.org/10.1007/s00453-001-0051-5
53. Dagum, L., Leonardo, R.: OpenMP: an industry-standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998). https://doi.org/10.1109/99.660313
54. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008). https://doi.org/10.1145/1327452.1327492
55. Delcher, A.L., Kasif, S., Fleischmann, R.D., Peterson, J., White, O., Salzberg, S.L.: Alignment of whole genomes. Nucleic Acids Res. **27**(11), 2369–2376 (1999). https://doi.org/10.1093/nar/27.11.2369

56. Dementiev, R., Kärkkäinen, J., Mehnert, J., Sanders, P.: Better external memory suffix array construction. ACM J. Exp. Algorithmics **12**, 3.4:1–3.4:24 (2008). https://doi.org/10.1145/1227161.1402296
57. Dementiev, R., Kettner, L., Sanders, P.: STXXL: standard template library for XXL data sets. Softw. Pract. Exp. **38**(6), 589–637 (2008). https://doi.org/10.1002/spe.844
58. Dementiev, R., Sanders, P.: Asynchronous parallel disk sorting. In: SPAA, pp. 138–148. ACM (2003). https://doi.org/10.1145/777412.777435
59. Deo, M., Keely, S.: Parallel suffix array and least common prefix for the GPU. In: PPOPP, pp. 197–206. ACM (2013). https://doi.org/10.1145/2442516.2442536
60. DeWitt, D.J., Naughton, J.F., Schneider, D.A.: Parallel sorting on a shared-nothing architecture using probabilistic splitting. In: PDIS, pp. 280–291. IEEE Computer Society (1991). https://doi.org/10.1109/PDIS.1991.183115
61. Dinklage, P.: Translating between wavelet tree and wavelet matrix construction. In: Stringology, pp. 126–135. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science (2019)

62 SPP. Dinklage, P., Ellert, J., Fischer, J., Köppl, D., Penschuck, M.: Bidirectional text compression in external memory. In: ESA, pp. 41:1–41:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ESA.2019.41

63 SPP. Dinklage, P., Ellert, J., Fischer, J., Kurpicz, F., Löbel, M.: Practical wavelet tree construction. ACM J. Exp. Algorithmics **26** (2021). https://doi.org/10.1145/3457197

64 SPP. Dinklage, P., Fischer, J., Kurpicz, F.: Constructing the wavelet tree and wavelet matrix in distributed memory. In: ALENEX, pp. 214–228. SIAM (2020). https://doi.org/10.1137/1.9781611976007.17

65. Edelkamp, S., Weiß, A.: Blockquicksort: avoiding branch mispredictions in quicksort. ACM J. Exp. Algorithmics **24**(1), 1.4:1–1.4:22 (2019). https://doi.org/10.1145/3274660
66. Edwards, J.A., Vishkin, U.: Parallel algorithms for burrows-wheeler compression and decompression. Theor. Comput. Sci. **525**, 10–22 (2014). https://doi.org/10.1016/j.tcs.2013.10.009
67. Egidi, L., Louza, F.A., Manzini, G., Telles, G.P.: External memory BWT and LCP computation for sequence collections with applications. Algorithms Mol. Biol. **14**(1), 6:1–6:15 (2019). https://doi.org/10.1186/s13015-019-0140-0

68 SPP. Ellert, J., Fischer, J., Sitchinava, N.: LCP-aware parallel string sorting. In: Malawski, M., Rzadca, K. (eds.) Euro-Par 2020. LNCS, vol. 12247, pp. 329–342. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57675-2_21

69 SPP. Ellert, J., Kurpicz, F.: Parallel external memory wavelet tree and wavelet matrix construction. In: Brisaboa, N.R., Puglisi, S.J. (eds.) SPIRE 2019. LNCS, vol. 11811, pp. 392–406. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32686-9_28

70. Fagerberg, R., Pagh, A., Pagh, R.: External string sorting: faster and cache-oblivious. In: Durand, B., Thomas, W. (eds.) STACS 2006. LNCS, vol. 3884, pp. 68–79. Springer, Heidelberg (2006). https://doi.org/10.1007/11672142_4
71. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. J. ACM **47**(6), 987–1011 (2000). https://doi.org/10.1145/355541.355547
72. Ferragina, P., Gagie, T., Manzini, G.: Lightweight data indexing and compression in external memory. Algorithmica **63**(3), 707–730 (2012). https://doi.org/10.1007/s00453-011-9535-0
73. Ferragina, P., Giancarlo, R., Manzini, G.: The myriad virtues of wavelet trees. Inf. Comput. **207**(8), 849–866 (2009). https://doi.org/10.1016/j.ic.2008.12.010

74. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: FOCS, pp. 390–398. IEEE Computer Society (2000). https://doi.org/10.1109/SFCS.2000.892127
75. Fischer, J., Heun, V.: A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In: Chen, B., Paterson, M., Zhang, G. (eds.) ESCAPE 2007. LNCS, vol. 4614, pp. 459–470. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74450-4_41
76. Fischer, J., I, T., Köppl, D., Sadakane, K.: Lempel-Ziv factorization powered by space efficient suffix trees. Algorithmica **80**(7), 2048–2081 (2018). https://doi.org/10.1007/s00453-017-0333-1

77 SPP. Fischer, J., Kurpicz, F.: Dismantling divsufsort. In: Stringology, pp. 62–76. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague (2017)

78 SPP. Fischer, J., Kurpicz, F.: Lightweight distributed suffix array construction. In: ALENEX, pp. 27–38. SIAM (2019). https://doi.org/10.1137/1.9781611975499.3

79 SPP. Fischer, J., Kurpicz, F., Löbel, M.: Simple, fast and lightweight parallel wavelet tree construction. In: ALENEX, pp. 9–20. SIAM (2018). https://doi.org/10.1137/1.9781611975055.2

80 SPP. Fischer, J., Kurpicz, F., Sanders, P.: Engineering a distributed full-text index. In: ALENEX, pp. 120–134. SIAM (2017). https://doi.org/10.1137/1.9781611974768.10

81. Flick, P., Aluru, S.: Parallel distributed memory construction of suffix and longest common prefix arrays. In: SC, pp. 16:1–16:10. ACM (2015). https://doi.org/10.1145/2807591.2807609
82. Flick, P., Aluru, S.: Parallel construction of suffix trees and the all-nearest-smaller-values problem. In: IPDPS, pp. 12–21. IEEE Computer Society (2017). https://doi.org/10.1109/IPDPS.2017.62
83. Flick, P., Aluru, S.: Distributed enhanced suffix arrays: efficient algorithms for construction and querying. In: SC, pp. 72:1–72:17. ACM (2019). https://doi.org/10.1145/3295500.3356211
84. da Fonseca, P.G.S., da Silva, I.B.F.: Online construction of wavelet trees. In: SEA, pp. 16:1–16:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.SEA.2017.16
85. Fuentes-Sepúlveda, J., Elejalde, E., Ferres, L., Seco, D.: Parallel construction of wavelet trees on multicore architectures. Knowl. Inf. Syst. **51**(3), 1043–1066 (2017). https://doi.org/10.1007/s10115-016-1000-6
86. Fuentes-Sepúlveda, J., Navarro, G., Nekrich, Y.: Parallel computation of the burrows wheeler transform in compact space. Theor. Comput. Sci. **812**, 123–136 (2020). https://doi.org/10.1016/j.tcs.2019.09.030
87. Furtak, T., Amaral, J.N., Niewiadomski, R.: Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms. In: SPAA, pp. 348–357. ACM (2007). https://doi.org/10.1145/1248377.1248436
88. Futamura, N., Aluru, S., Kurtz, S.: Parallel suffix sorting. In: Electrical Engineering and Computer Science, vol. 64 (2001)
89. Gabriel, E., et al.: Open MPI: goals, concept, and design of a next generation MPI implementation. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds.) EuroPVM/MPI 2004. LNCS, vol. 3241, pp. 97–104. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30218-6_19
90. Gagie, T., Gawrychowski, P., Kärkkäinen, J., Nekrich, Y., Puglisi, S.J.: LZ77-based self-indexing with faster pattern matching. In: Pardo, A., Viola, A. (eds.) LATIN 2014. LNCS, vol. 8392, pp. 731–742. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54423-1_63

91. Gagie, T., Navarro, G., Prezza, N.: Fully functional suffix trees and optimal text searching in BWT-runs bounded space. J. ACM **67**(1), 2:1–2:54 (2020). https://doi.org/10.1145/3375890
92. Garrison, E., et al.: Variation graph toolkit improves read mapping by representing genetic variation in the reference. Nat. Biotechnol. **36**(9), 875–879 (2018). https://doi.org/10.1038/nbt.4227
93. Gilchrist, J., Cuhadar, A.: Parallel lossless data compression based on the burrows-wheeler transform. In: AINA, pp. 877–884. IEEE Computer Society (2007). https://doi.org/10.1109/AINA.2007.109
94. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: plug and play with succinct data structures. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 326–337. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07959-2_28
95. Gonnet, G.H., Baeza-Yates, R.A., Snider, T.: New indices for text: Pat trees and pat arrays. In: Information Retrieval: Data Structures & Algorithms, pp. 66–82. Prentice-Hall (1992)
96. Goodrich, M.T.: Communication-efficient parallel sorting. SIAM J. Comput. **29**(2), 416–432 (1999). https://doi.org/10.1137/S0097539795294141
97. Goto, K.: Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets. In: Stringology, pp. 111–125. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science (2019)
98. Gropp, W., Lusk, E.L., Doss, N.E., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Comput. **22**(6), 789–828 (1996). https://doi.org/10.1016/0167-8191(96)00024-5
99. ch14DBLP:confspssodaspsGrossiGV03 Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: SODA, pp. 841–850. ACM/SIAM (2003)
100. Grossi, R., Vitter, J.S., Xu, B.: Wavelet trees: from theory to practice. In: CCP, pp. 210–221. IEEE Computer Society (2011). https://doi.org/10.1109/CCP.2011.16
101. Ha, L.K., Krüger, J.H., Silva, C.T.: Fast four-way parallel radix sorting on GPUs. Comput. Graph. Forum **28**(8), 2368–2378 (2009). https://doi.org/10.1111/j.1467-8659.2009.01542.x
102. Hagerup, T.: Optimal parallel string algorithms: sorting, merging and computing the minimum. In: STOC, pp. 382–391. ACM (1994). https://doi.org/10.1145/195058.195202
103. Han, L.B., Wu, Y., Nong, G.: Succinct suffix sorting in external memory. Inf. Process. Manag. **58**(1), 102378 (2021). https://doi.org/10.1016/j.ipm.2020.102378
104. Hayashi, S., Taura, K.: Parallel and memory-efficient burrows-wheeler transform. In: BigData, pp. 43–50. IEEE Computer Society (2013). https://doi.org/10.1109/BigData.2013.6691757
105. He, X., Huang, C.: Communication efficient BSP algorithm for all nearest smaller values problem. J. Parallel Distributed Comput. **61**(10), 1425–1438 (2001). https://doi.org/10.1006/jpdc.2001.1741
106. Helman, D.R., Bader, D.A., JáJá, J.: A randomized parallel sorting algorithm with an experimental study. J. Parallel Distributed Comput. **52**(1), 1–23 (1998). https://doi.org/10.1006/jpdc.1998.1462
107. Hoare, C.A.R.: Quicksort. Comput. J. **5**(1), 10–15 (1962). https://doi.org/10.1093/comjnl/5.1.10
108. Hou, K., Wang, H., Feng, W.: A framework for the automatic vectorization of parallel sort on x86-based processors. IEEE Trans. Parallel Distrib. Syst. **29**(5), 958–972 (2018). https://doi.org/10.1109/TPDS.2018.2789903

109. Huang, B., Gao, J., Li, X.: An empirically optimized radix sort for GPU. In: ISPA, pp. 234–241. IEEE Computer Society (2009). https://doi.org/10.1109/ISPA.2009.89
110. Inoue, H., Moriyama, T., Komatsu, H., Nakatani, T.: A high-performance sorting algorithm for multicore single-instruction multiple-data processors. Softw. Pract. Exp. **42**(6), 753–777 (2012). https://doi.org/10.1002/spe.1102
111. Itoh, H., Tanaka, H.: An efficient method for in memory construction of suffix arrays. In: SPIRE/CRIWG, pp. 81–88. IEEE (1999). https://doi.org/10.1109/SPIRE.1999.796581
112. JáJá, J.: An Introduction to Parallel Algorithms. Addison-Wesley, Boston (1992)
113. JáJá, J., Ryu, K.W., Vishkin, U.: Sorting strings and constructing digital search trees in parallel. Theor. Comput. Sci. **154**(2), 225–245 (1996). https://doi.org/10.1016/0304-3975(94)00263-0
114. Kaneta, Y.: Fast wavelet tree construction in practice. In: Gagie, T., Moffat, A., Navarro, G., Cuadros-Vargas, E. (eds.) SPIRE 2018. LNCS, vol. 11147, pp. 218–232. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00479-8_18
115. Kärkkäinen, J., Kempa, D.: Faster external memory LCP array construction. In: ESA.,pp. 61:1–61:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.ESA.2016.61
116. Kärkkäinen, J., Kempa, D.: LCP array construction in external memory. ACM J. Exp. Algorithmics **21**(1), 1.7:1–1.7:22 (2016). https://doi.org/10.1145/2851491
117. Kärkkäinen, J., Kempa, D.: Engineering a lightweight external memory suffix array construction algorithm. Math. Comput. Sci. **11**(2), 137–149 (2017). https://doi.org/10.1007/s11786-016-0281-1
118. Kärkkäinen, J., Kempa, D.: Engineering external memory LCP array construction: Parallel, in-place and large alphabet. In: SEA, pp. 17:1–17:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.SEA.2017.17
119. Kärkkäinen, J., Kempa, D.: Better external memory LCP array construction. ACM J. Exp. Algorithmics **24**(1), 1.3:1–1.3:27 (2019). https://doi.org/10.1145/3297723
120. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Lempel-ziv parsing in external memory. In: DCC, pp. 153–162. IEEE (2014). https://doi.org/10.1109/DCC.2014.78
121. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Parallel external memory suffix sorting. In: Cicalese, F., Porat, E., Vaccaro, U. (eds.) CPM 2015. LNCS, vol. 9133, pp. 329–342. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19929-0_28
122. Kärkkäinen, J., Kempa, D., Puglisi, S.J., Zhukova, B.: Engineering external memory induced suffix sorting. In: ALENEX, pp. 98–108. SIAM (2017). https://doi.org/10.1137/1.9781611974768.8
123. Kärkkäinen, J., Rantala, T.: Engineering radix sort for strings. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 3–14. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89097-3_3
124. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM **53**(6), 918–936 (2006). https://doi.org/10.1145/1217856.1217858
125. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A. (ed.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-48194-X_17
126. Kempa, D., Kociumaka, T.: String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In: STOC, pp. 756–767. ACM (2019). https://doi.org/10.1145/3313276.3316368
127. Kitajima, J.P., Navarro, G.: A fast distributed suffix array generation algorithm. In: SPIRE/CRIWG, pp. 97–105. IEEE (1999). https://doi.org/10.1109/SPIRE.1999.796583

128. Kitajima, J.P., Ribeiro-Neto, B., Ziviani, N.: Network and memory analysis in distributed parallel generation of pat arrays. In: BSCA, pp. 192–202 (1996)
129. Knuth, D.E.: The Art of Computer Programming, Volume III: Sorting and Searching. Addison-Wesley (1973)
130. Kuhnle, A., Mun, T., Boucher, C., Gagie, T., Langmead, B., Manzini, G.: Efficient construction of a complete index for pan-genomics read alignment. In: Cowen, L.J. (ed.) RECOMB 2019. LNCS, vol. 11467, pp. 158–173. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17083-7_10
131. Kulla, F., Sanders, P.: Scalable parallel suffix array construction. Parallel Comput. **33**(9), 605–612 (2007). https://doi.org/10.1016/j.parco.2007.06.004
132. Labeit, J., Shun, J., Blelloch, G.E.: Parallel lightweight wavelet tree, suffix array and FM-index construction. J. Discrete Algorithms **43**, 2–17 (2017). https://doi.org/10.1016/j.jda.2017.04.001
133. Lan, Y., Mohamed, M.A.: Parallel quicksort in hypercubes. In: SAC, pp. 740–746. ACM (1992). https://doi.org/10.1145/130069.130085
134. Langmead, B., Salzberg, S.L.: Fast gapped-read alignment with bowtie 2. Nat. Meth. **9**(4), 357 (2012). https://doi.org/10.1038/nmeth.1923
135. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.L.: Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol. **10**(3), R25 (2009). https://doi.org/10.1186/gb-2009-10-3-r25
136. Lao, B., Nong, G., Chan, W.H., Pan, Y.: Fast induced sorting suffixes on a multicore machine. J. Supercomput. **74**(7), 3468–3485 (2018). https://doi.org/10.1007/s11227-018-2395-5
137. Lao, B., Nong, G., Chan, W.H., Xie, J.Y.: Fast in-place suffix sorting on a multicore computer. IEEE Trans. Comput. **67**(12), 1737–1749 (2018). https://doi.org/10.1109/TC.2018.2842050
138. Lee, S., Jeon, M., Kim, D., Sohn, A.: Partitioned parallel radix sort. J. Parallel Distributed Comput. **62**(4), 656–668 (2002). https://doi.org/10.1006/jpdc.2001.1808
139. Li, H.: Fast construction of FM-index for long sequence reads. Bioinform. **30**(22), 3274–3275 (2014). https://doi.org/10.1093/bioinformatics/btu541
140. Li, H., Durbin, R.: Fast and accurate short read alignment with burrows-wheeler transform. Bioinform. **25**(14), 1754–1760 (2009). https://doi.org/10.1093/bioinformatics/btp324
141. Li, Z., Li, J., Huo, H.: Optimal in-place suffix sorting. In: Gagie, T., Moffat, A., Navarro, G., Cuadros-Vargas, E. (eds.) SPIRE 2018. LNCS, vol. 11147, pp. 268–284. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00479-8_22
142. Lippert, R.: Space-efficient whole genome comparisons with Burrows Wheeler transforms. J. Comput. Biol. **12**(4), 407–415 (2005). https://doi.org/10.1089/cmb.2005.12.407
143. Liu, Y., Hankeln, T., Schmidt, B.: Parallel and space-efficient construction of burrows-wheeler transform and suffix array for big genome data. IEEE/ACM Trans. Comput. Biol. Bioinform. **13**(3), 592–598 (2016). https://doi.org/10.1109/TCBB.2015.2430314
144. Louza, F.A., Gog, S., Telles, G.P.: Induced suffix sorting. In: Construction of Fundamental Data Structures for Strings. SCS, pp. 23–40. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-55108-7_3
145. Louza, F.A., Telles, G.P., Hoffmann, S., de Aguiar Ciferri, C.D.: Generalized enhanced suffix array construction in external memory. Algorithms Mol. Biol. **12**(1), 26:1–26:16 (2017). https://doi.org/10.1186/s13015-017-0117-9
146. Mahmoud, H.M.: Sorting: A Distribution Theory. John Wiley, Hoboken (2000)
147. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theor. Comput. Sci. **387**(3), 332–347 (2007). https://doi.org/10.1016/j.tcs.2007.07.013

148. Mäkinen, V., Navarro, G., Sadakane, K.: Advantages of Backward Searching — Efficient Secondary Memory and Distributed Implementation of Compressed Suffix Arrays. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 681–692. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30551-4_59
149. Makris, C.: Wavelet trees: a survey. Comput. Sci. Inf. Syst. **9**(2), 585–625 (2012). https://doi.org/10.2298/CSIS110606004M
150. Manber, U., Myers, E.W.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993). https://doi.org/10.1137/0222058
151. Marchet, C., Boucher, C., Puglisi, S.J., Medvedev, P., Salson, M., Chikhi, R.: Data structures based on k-mers for querying large collections of sequencing datasets. bioRxiv (2020). https://doi.org/10.1101/866756
152. McIlroy, P.M., Bostic, K., McIlroy, M.D.: Engineering radix sort. Comput. Syst. **6**(1), 5–27 (1993)
153. Menon, R.K., Bhat, G.P., Schatz, M.C.: Rapid parallel genome indexing with mapreduce. In: MapReduce, pp. 51–58 (2011). https://doi.org/10.1145/1996092.1996104
154. Merrill, D., Grimshaw, A.S.: High performance and scalable radix sorting: a case study of implementing dynamic parallelism for GPU computing. Parallel Process. Lett. **21**(2), 245–272 (2011). https://doi.org/10.1142/S0129626411000187
155. Metwally, A.A., Kandil, A.H., Abouelhoda, M.: Distributed suffix array construction algorithms: Comparison of two algorithms. In: CIBEC, pp. 27–30. IEEE (2016). https://doi.org/10.1109/CIBEC.2016.7836092
156. Morrison, D.R.: PATRICIA - practical algorithm to retrieve information coded in alphanumeric. J. ACM **15**(4), 514–534 (1968). https://doi.org/10.1145/321479.321481
157. Munro, J.I., Navarro, G., Nekrich, Y.: Space-efficient construction of compressed indexes in deterministic linear time. In: SODA, pp. 408–424. SIAM (2017). https://doi.org/10.1137/1.9781611974782.26
158. Munro, J.I., Nekrich, Y., Vitter, J.S.: Fast construction of wavelet trees. Theor. Comput. Sci. **638**, 91–97 (2016). https://doi.org/10.1016/j.tcs.2015.11.011
159. Musser, D.R.: Introspective sorting and selection algorithms. Softw. Pract. Exp. **27**(8), 983–993 (1997). https://doi.org/10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-%23
160. Navarro, G.: Wavelet trees for all. J. Discrete Algorithms **25**, 2–20 (2014). https://doi.org/10.1016/j.jda.2013.07.004
161. Navarro, G., Kitajima, J.P., Ribeiro-Neto, B.A., Ziviani, N.: Distributed generation of suffix arrays. In: Apostolico, A., Hein, J. (eds.) CPM 1997. LNCS, vol. 1264, pp. 102–115. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63220-4_54
162. Ng, W., Kakehi, K.: Merging string sequences by longest common prefixes. IPSJ Digital Courier **4**, 69–78 (2008)
163. Nong, G., Chan, W.H., Hu, S.Q., Wu, Y.: Induced sorting suffixes in external memory. ACM Trans. Inf. Syst. **33**(3), 12:1–12:15 (2015). https://doi.org/10.1145/2699665
164. Nong, G., Zhang, S., Chan, W.H.: Two efficient algorithms for linear time suffix array construction. IEEE Trans. Comput. **60**(10), 1471–1484 (2011). https://doi.org/10.1109/TC.2010.188
165. Obeya, O., Kahssay, E., Fan, E., Shun, J.: Theoretically-efficient and practical parallel in-place radix sorting. In: SPAA, pp. 213–224. ACM (2019). https://doi.org/10.1145/3323165.3323198
166. Ohlebusch, E.: Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction. Oldenbusch Verlag (2013)
167. Ohlebusch, E., Beller, T., Abouelhoda, M.I.: Computing the burrows-wheeler transform of a string and its reverse in parallel. J. Discrete Algorithms **25**, 21–33 (2014). https://doi.org/10.1016/j.jda.2013.06.002

168. Ohlebusch, E., Gog, S., Kügel, A.: Computing matching statistics and maximal exact matches on compressed full-text indexes. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 347–358. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16321-0_36
169. Osipov, V.: Parallel suffix array construction for shared memory architectures. In: Calderón-Benavides, L., González-Caro, C., Chávez, E., Ziviani, N. (eds.) SPIRE 2012. LNCS, vol. 7608, pp. 379–384. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34109-0_40
170. Patel, R.A., Zhang, Y., Mak, J., Davidson, A., Owens, J.D.: Parallel lossless data compression on the GPU. In: InPar, pp. 1–9. IEEE (2012). https://doi.org/10.1109/InPar.2012.6339599
171. Pockrandt, C.: Approximate String Matching: Improving Data Structures and Algorithms. Ph.D. thesis, Free University of Berlin, Dahlem, Germany (2019). https://doi.org/10.17169/refubium-2185
172. Puglisi, S.J., Smyth, W.F., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Comput. Surv. **39**(2), 4 (2007). https://doi.org/10.1145/1242471.1242472
173. Rahman, N., Raman, R.: Adapting radix sort to the memory hierarchy. ACM J. Exp. Algorithmics **6**, 7 (2001). https://doi.org/10.1145/945394.945401
174. Reinders, J.: Intel Threading Building Blocks - Outfitting C++ for Multi-core Processor Parallelism. O'Reilly, Newton (2007)
175. Reinert, K., et al.: The seqan c++ template library for efficient sequence analysis: a resource for programmers. J. Biotechnol. **261**, 157–168 (2017). https://doi.org/10.1016/j.jbiotec.2017.07.017
176. Sadakane, K.: Compressed text databases with efficient query algorithms based on the compressed suffix array. In: Goos, G., Hartmanis, J., van Leeuwen, J., Lee, D.T., Teng, S.-H. (eds.) ISAAC 2000. LNCS, vol. 1969, pp. 410–421. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-40996-3_35
177. Sadakane, K.: Succinct representations of lcp information and improvements in the compressed suffix arrays. In: SODA, pp. 225–232. ACM/SIAM (2002)
178. Sanders, P., Hansch, T.: Efficient massively parallel quicksort. In: Bilardi, G., Ferreira, A., Lüling, R., Rolim, J. (eds.) IRREGULAR 1997. LNCS, vol. 1253, pp. 13–24. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63138-0_2
179. Sanders, P., Schlag, S., Müller, I.: Communication efficient algorithms for fundamental big data problems. In: BigData, pp. 15–23. IEEE (2013). https://doi.org/10.1109/BigData.2013.6691549
180. Sanders, P., Winkel, S.: Super scalar sample sort. In: Albers, S., Radzik, T. (eds.) ESA 2004. LNCS, vol. 3221, pp. 784–796. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30140-0_69
181. Satish, N., Harris, M.J., Garland, M.: Designing efficient sorting algorithms for many-core GPUs. In: IPDPS, pp. 1–10. IEEE (2009). https://doi.org/10.1109/IPDPS.2009.5161005
182. Shepherdson, J.C., Sturgis, H.E.: Computability of recursive functions. J. ACM **10**(2), 217–255 (1963). https://doi.org/10.1145/321160.321170
183. Shun, J.: Fast parallel computation of longest common prefixes. In: SC, pp. 387–398. IEEE (2014). https://doi.org/10.1109/SC.2014.37
184. Shun, J.: Parallel wavelet tree construction. In: DCC, pp. 63–72. IEEE (2015). https://doi.org/10.1109/DCC.2015.7
185. Shun, J.: Improved parallel construction of wavelet trees and rank/select structures. Inf. Comput. **273**, 104516 (2020). https://doi.org/10.1016/j.ic.2020.104516
186. Shun, J., et al.: Brief announcement: the problem based benchmark suite. In: SPAA, pp. 68–70. ACM (2012). https://doi.org/10.1145/2312005.2312018

187. Simpson, J.T., Durbin, R.: Efficient construction of an assembly string graph using the FM-index. Bioinformatics **26**(12), 367–373 (2010). https://doi.org/10.1093/bioinformatics/btq217
188. Singler, J., Sanders, P., Putze, F.: MCSTL: the multi-core standard template library. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 682–694. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74466-5_72
189. Sinha, R., Zobel, J.: Efficient trie-based sorting of large sets of strings. In: ACSC, pp. 11–18. Australian Computer Society (2003)
190. Sirén, J.: Indexing variation graphs. In: ALENEX, pp. 13–27. SIAM (2017). https://doi.org/10.1137/1.9781611974768.2
191. Sirén, J., Garrison, E., Novak, A.M., Paten, B., Durbin, R.: Haplotype-aware graph indexes. Bioinformatics **36**(2), 400–407 (2020). https://doi.org/10.1093/bioinformatics/btz575
192. Sohn, A., Kodama, Y.: Load balanced parallel radix sort. In: ICS, pp. 305–312. ACM (1998). https://doi.org/10.1145/277830.277903
193. Solomonik, E., Kalé, L.V.: Highly scalable parallel sorting. In: IPDPS, pp. 1–12. IEEE (2010). https://doi.org/10.1109/IPDPS.2010.5470406
194. Stehle, E., Jacobsen, H.: A memory bandwidth-efficient hybrid radix sort on GPUs. In: SIGMOD Conference, pp. 417–432. ACM (2017). https://doi.org/10.1145/3035918.3064043
195. Sun, W., Ma, Z.: Parallel lexicographic names construction with CUDA. In: ICPADS, pp. 913–918. IEEE Computer Society (2009). https://doi.org/10.1109/ICPADS.2009.31
196. Sundar, H., Malhotra, D., Biros, G.: Hyksort: a new variant of hypercube quicksort on distributed memory architectures. In: ICS, pp. 293–302. ACM (2013). https://doi.org/10.1145/2464996.2465442
197. Tischler, G.: On wavelet tree construction. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 208–218. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21458-5_19
198. Trinidad, J.F.M., Cumplido-Parra, R., Uribe, C.F.: An FPGA-based parallel sorting architecture for the burrows wheeler transform. In: ReConFig. IEEE Computer Society (2005). https://doi.org/10.1109/RECONFIG.2005.9
199. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**(3), 249–260 (1995). https://doi.org/10.1007/BF01206331
200. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990). https://doi.org/10.1145/79173.79181
201. Wang, H., et al.: BWTCP: a parallel method for constructing BWT in large collection of genomic reads. In: Kunkel, J.M., Ludwig, T. (eds.) ISC High Performance 2015. LNCS, vol. 9137, pp. 171–178. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20119-1_13
202. Wang, L., Baxter, S., Owens, J.D.: Fast parallel skew and prefix-doubling suffix array construction on the GPU. Concurr. Comput. Pract. Exp. **28**(12), 3466–3484 (2016). https://doi.org/10.1002/cpe.3867
203. Wassenberg, J., Sanders, P.: Engineering a multi-core radix sort. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011. LNCS, vol. 6853, pp. 160–169. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23397-5_16
204. Weiner, P.: Linear pattern matching algorithms. In: SWAT (FOCS), pp. 1–11. IEEE (1973). https://doi.org/10.1109/SWAT.1973.13

205. Wild, S., Nebel, M.E., Neininger, R.: Average case and distributional analysis of dual-pivot quicksort. ACM Trans. Algorithms **11**(3), 22:1–22:42 (2015). https://doi.org/10.1145/2629340
206. Wu, Y., Lao, B., Ma, X., Nong, G.: An improved algorithm for building suffix array in external memory. In: Shen, H., Sang, Y. (eds.) PAAP 2019. CCIS, vol. 1163, pp. 320–330. Springer, Singapore (2020). https://doi.org/10.1007/978-981-15-2767-8_29
207. Xiaochen, T., Rocki, K., Suda, R.: Register level sort algorithm on multi-core SIMD processors. In: IA3@SC, pp. 9:1–9:8. ACM (2013). https://doi.org/10.1145/2535753.2535762
208. Xie, J.Y., Lao, B., Nong, G.: In-place suffix sorting on a multicore computer with better design. In: Shen, H., Sang, Y. (eds.) PAAP 2019. CCIS, vol. 1163, pp. 331–342. Springer, Singapore (2020). https://doi.org/10.1007/978-981-15-2767-8_30
209. Yin, Z., et al.: Efficient parallel sort on AVX-512-based multi-core and many-core architectures. In: HPCC/SmartCity/DSS, pp. 168–176. IEEE (2019). https://doi.org/10.1109/HPCC/SmartCity/DSS.2019.00038
210. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: HotCloud. USENIX Association (2010)
211. Zhou, D., Andersen, D.G., Kaminsky, M.: Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In: Bonifaci, V., Demetrescu, C., Marchetti-Spaccamela, A. (eds.) SEA 2013. LNCS, vol. 7933, pp. 151–163. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38527-8_15
212. Zhu, G., Guo, C., Lu, L., Huang, Z., Yuan, C., Gu, R., Huang, Y.: DGST: efficient and scalable suffix tree construction on distributed data-parallel platforms. Parallel Comput. **87**, 87–102 (2019). https://doi.org/10.1016/j.parco.2019.06.002
213. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977). https://doi.org/10.1109/TIT.1977.1055714

Author Index

Abu-Khzam, Faisal N. 97
Albers, Susanne 196
Angriman, Eugenio 3

Bingmann, Timo 252
Brandes, Ulrik 134

Castenow, Jannik 179
Chinazzo, André 39, 57

De Schryver, Christian 39
Dinklage, Patrick 252
Disser, Yann 152
Droschinsky, Andre 76

Feldkord, Björn 179
Fischer, Johannes 252

Giesen, Joachim 213
van der Grinten, Alexander 3

Hamann, Michael 3, 134
Hanselle, Jonas 179
Häuser, Luise 134
Hofheinz, Dennis 169
Humbeck, Lina 76

Kiltz, Eike 169
Klimm, Max 152
Knollmann, Till 179
Koch, Oliver 76
Kriege, Nils M. 76
Kuehne, Lars 213
Kurpicz, Florian 252

Lamm, Sebastian 97
Laue, Sören 213

Malatyali, Manuel 179
Meyer auf der Heide, Friedhelm 179
Meyer, Ulrich 21
Meyerhenke, Henning 3
Mnich, Matthias 97
Mutzel, Petra 76

Noe, Alexander 97

Ohlebusch, Enno 252

Penschuck, Manuel 3, 21

Sanders, Peter 252
Schäfer, Till 76
Schielke, Christian 229
Schiemann, Jan 229
Schryver, Christian De 57
Schulz, Christian 97
Srivastav, Anand 229
Strash, Darren 97

Wagner, Dorothea 134
Wedemeyer, Axel 229
Wehn, Norbert 39, 57

Zweig, Katharina 39, 57