Enabling Scalability: Graph Hierarchies and Fault Tolerance

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik des
Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation
von
Lars Demian Hespe

Tag der mündlichen Prüfung: 21. Juli 2023

1. Referent:	Prof. Dr. Peter Sanders Karlsruher Institut für Technologie Deutschland
2. Referent:	Prof. Dr. Henning Meyerhenke Humboldt-Universität zu Berlin Deutschland

To my family.

Abstract

In this dissertation, we explore approaches to two techniques for building scalable algorithms. First, we look at different graph problems. We show how to exploit the input graph's inherent hierarchy for scalable graph algorithms. The second technique takes a step back from concrete algorithmic problems. Here, we consider the case of node failures in large distributed systems and present techniques to quickly recover from these.

In the first part of the dissertation, we investigate how hierarchies in graphs can be used to scale algorithms to large inputs. We develop algorithms for three graph problems based on two approaches to build hierarchies. The first approach reduces instance sizes for NP-hard problems by applying so-called *reduction rules*. These rules can be applied in polynomial time. They either find parts of the input that can be solved in polynomial time, or they identify structures that can be contracted (reduced) into smaller structures without loss of information for the specific problem. After solving the reduced instance using an exponential-time algorithm, these previously contracted structures can be uncontracted to obtain an exact solution for the original input. In addition to a simple preprocessing procedure, reduction rules can also be used in *branch-and-reduce* algorithms where they are successively applied after each branching step to build a hierarchy of problem kernels of increasing computational hardness. We develop reduction-based algorithms for the classical NP-hard problems *Maximum Independent Set* and *Maximum Cut*. The second approach is used for *route planning* in road networks where we build a hierarchy of road segments based on their importance for long distance shortest paths. By only considering important road segments when we are far away from the source and destination, we can substantially speed up shortest path queries.

In the second part of this dissertation, we take a step back from concrete graph problems and look at more general problems in high performance computing (HPC). Here, due to the ever increasing size and complexity of HPC clusters, we expect hardware and software failures to become more common in massively parallel computations. We present two techniques for applications to recover from failures and resume computation. Both techniques are based on in-memory storage of redundant information and a data distribution that enables fast recovery. The first technique can be used for general purpose distributed processing frameworks: We identify data that is redundantly available on multiple machines and only introduce additional work for the remaining data that is only available on one machine. The second technique is a checkpointing library engineered for fast recovery using a

data distribution method that achieves balanced communication loads. Both our techniques have in common that they work in settings where computation after a failure is continued with less machines than before. This is in contrast to many previous approaches that—in particular for checkpointing—focus on systems that keep spare resources available to replace failed machines.

Overall, we present different techniques that enable scalable algorithms. While some of these techniques are specific to graph problems, we also present tools for fault tolerant algorithms and applications in a distributed setting. To show that those can be helpful in many different domains, we evaluate them for graph problems and other applications like phylogenetic tree inference.

Acknowledgements

First and foremost, I would like to thank Peter Sanders for giving me the opportunity to work in his research group as well as providing guidance when needed and letting me pursue my own research interests. I would also like to thank Henning Meyerhenke for agreeing to review this dissertation.

Furthermore I would like to thank my co-authors Jonathan Dees, Damir Ferizovic, Lorenz Hübschle-Schneider, Lukas Hübner, Alexey Kozlov, Sebastian Lamm, Matthias Mnich, Peter Sanders, Christian Schorr, Dominik Schreiber, Christian Schulz, Alexandros Stamatakis, Darren Strash, and Martin Weidner for all the fruitful cooperations as well as my students Damir Ferizovic, Tom George, Lukas Hübner, Charel Mercatoris, and Christian Schorr.

I really enjoyed working with all my colleagues from over the years. The work-related (or not) discussions, the immense combined knowledge, the virtual coffee breaks while we were all stuck at home, the nights out, and the group retreats made this a wonderful place to work. Thank you, Yaroslav Akhremtsev, Michael Axtmann, Tomáš Balyo, Timo Bingmann, Daniel Funke, Simon Gog, Lars Gottesbüren, Tobias Heuer, Lukas Hübner, Lorenz Hübschle-Schneider, Markus Iser, Florian Kurpicz, Sebastian Lamm, Moritz Laupichler, Hans-Peter Lehmann, Nikolai Maas, Tobias Maier, Matthias Schimek, Sebastian Schlag, Dominik Schreiber, Christian Schulz, Daniel Seemaier, Jochen Speck, Tim Niklas Uhl, Marvin Williams, and Sascha Witt.

I thank Shengjia Feng, Florian Kurpicz, Moritz Laupichler, Matthias Schimek, Dominik Schreiber, Daniel Seemaier, and Tim Niklas Uhl for proof-reading parts of this dissertation and their helpful comments and suggestions.

Finally, I would like to thank my family and Shengjia Feng for their love and support throughout the years as well as my friends for keeping me sane.

Table of Contents

Chapter 1

1 Introduction and Overview

In this chapter, we motivate the work presented in the rest of this dissertation and summarize the contributions of the following chapters. We develop new algorithms that enable scaling to large problem instances based on two approaches: Exploiting hierarchy in graph problem instances and tolerating compute node failures in distributed applications that scale to large cluster computers.

Graphs are a common tool to model real-world or digital data. Examples include road networks, friendship relations, web links, VLSI design, or image segmentation [DGJ09; LK14; RA15; DGS18]. With the size of today's networks in the billions we need algorithms that can scale to large graphs. For example, in 2011, the Facebook graph contained 721 million vertices and 68.7 billion edges [Uga+11]. On real-world instances even presumably fast algorithms like Dijkstra's algorithm for shortest paths [Dij59] can take seconds [Bas+16] which becomes a problem when it has to be run thousands of times in route planning services. We present algorithms that aim to scale to large graphs by building or exploiting graph hierarchies. These help to identify parts of the graph that need the most attention or the most computational resources. By focusing work on these important parts and spending less resources on the unimportant parts, we develop practically fast algorithms.

When scaling to increasingly large inputs, at some point a single machine is not sufficient for solving many problems. Either because of the limited computational power or because the input would simply not fit into the memory of a single machine. At this point, we move to distributed systems. In the scientific world, these are often large *High Performance Computing* (HPC) clusters consisting of tens to hundreds of thousands of compute cores, e.g., [Ste23; Lei23]. With the increasing number of processors in HPC clusters, the probability that some processors fail during a computation rises. Handling such failures constitutes a major challenge for future exascale systems [SDM10]. For example ORNL's Jaguar Titan Cray XK7 system had on average 2.33 failures/day between August 2008 and 2010 [Gam+14]. In upcoming systems, we expect a hardware failure to occur every 30 to 60 minutes [Cap+14; DHR15; Sni+14]. We develop techniques to recover from such failures based on replication and purpose-built data distributions that cause little overhead during normal operation and fast recovery after a failure.

1.1 Contributions

Hierarchical Graph Algorithms (Part I)

In Part I, we investigate hierarchies in graph problems using two underlying approaches: Problem size reduction and identification of important roads in road networks.

For the first approach, we investigate algorithms that reduce the problem size by either solving the "easy" parts of an instance beforehand, or by "compressing" subgraphs into smaller ones to be solved later. These reductions can then be used in so-called *branch-and-reduce* algorithms which work similarly to branch-and-bound algorithms with an additional reduction step after each branching step.

For the second approach, we identify important roads in a road network and use that information to build auxiliary data structures that speed up shortest path queries.

For both approaches, we present algorithms that either improve on or match the performance of the current state-of-the-art on meaningful and standard benchmark sets.

Maximum Independent Sets (Chapter 3). We present two results for the Maximum Independent Set (MIS) Problem. First, we show an engineered algorithm based on both branch-and-reduce and branch-and-bound that won the 2019 PACE Implementation Challenge[1] by 13% more instances solved than the second place. Second, we develop new branching rules for branch-and-reduce algorithms that specifically target branching points that either enable the reduction step to further decrease the problem size or separate the graph into two or more components that can be solved independently. Our experiments show that this enables us to solve the instances of the aforementioned PACE challenge 30% faster than with the standard branching strategy.

Maximum Cuts (Chapter 4). We present the first practical reduction algorithm for the Maximum Cut Problem. While previous theoretical *kernelization*[2] algorithms for Maximum Cut exist, they were not evaluated in practice before. We implement and generalize the theoretical reduction rules such that they can be applied with less restrictions and are faster to compute. We show that these new rules are able to substantially reduce the problem size for specific graph classes. We also show that the size reduction leads to up to 900 times shorter running times to fully solve our benchmark instances than using a state-of-the-art solver on the original graph.

Route Planning in Road Networks (Chapter 5). Some of the most successful algorithms for shortest path calculations in road networks compute a hierarchy of

[1] Parameterized Algorithms and Computational Experiments Challenge: a yearly international competition on changing problems from the world of parameterized algorithms. `https://pacechallenge.org/2019/`

[2] The term used for problem size reduction in *fixed parameter tractable* (FPT) algorithms that have more strict requirements for the reduction rules

the graph in a preprocessing step after which shortest path queries can be answered multiple orders of magnitude faster than in algorithms that do not preprocess the graph. Previously, new algorithms were able to achieve faster query times by adding more levels of hierarchy. This culminated in *Contraction Hierarchies* where each *vertex* in the graph represents its own level in the hierarchy. We present *Edge Hierarchies* that go even further and put each *edge* on its own level. We show that Edge Hierarchies can achieve comparable query times to Contraction Hierarchies on continental-sized road networks and under specific metrics can even outperform them.

Data Recovery for Fault-Tolerant MPI Applications (Part II)

When scaling algorithms to the largest compute clusters we have today (or the even larger ones we expect to have in the future), we have to expect some compute nodes to fail during the execution of a program. Thus, in order to finish such a computation, we have to make the underlying algorithm fault tolerant. While most existing solutions assume that spare resources are available to replace failed ones, our work focuses on continuing work on the remaining compute nodes as well as fast recovery. In Part II, we present an approach to fault tolerance in general purpose parallel processing frameworks and a generic library for the rapid recovery of data after a failure. Of course, this also relates to graph algorithms similar to those presented in Part I. In fact, frameworks like the ones we study in Section 7 are often used to parallelize graph algorithms. Either directly, as shown in our own experimental evaluation, or through graph processing frameworks built on top of them [Xin+13].

Fault Tolerance for Distributed Processing Frameworks (Chapter 7). One way of developing parallel algorithms is the use of parallel processing frameworks like MapReduce [DG08], Spark [Zah+12], or Thrill [Bin+16]. By expressing an algorithm in terms of their restricted set of operations, these frameworks can parallelize the execution without any explicit parallelization efforts by the application programmer. This can also include transparently tolerating node failures without any intervention by the application programmer or system administrator. Many existing implementations—especially for MapReduce—use local hard disks and distributed filesystems for this or, in the case of MPI-based implementations, do not support fault tolerance at all. We present an approach for tolerating failures in parallel processing frameworks without the need for (local or remote) disks by storing messages sent between the participating nodes in memory. This only has a small impact on fault-free execution times of at most 4% for most benchmark problems and enables fast recovery after a failure.

Fast General Purpose Data Recovery (Chapter 8). One main concern of fault tolerant applications is recovering the data lost after a failure. Most existing approaches focus on the fast creation of checkpoints for rapidly changing data but in turn show relatively slow recovery times. In some cases, however, this data is static

or only changes rarely. We present a library for these cases where fast recovery times are needed. This will become increasingly important when failures become more frequent in even larger systems. We store all required data in-memory and optimize the data distribution such that a large fraction of the nodes can participate in the data recovery procedure. We also use our library in a widely used bioinformatics application where we are able to reduce recovery times by often more than an order of magnitude.

Part I

Hierarchical Graph Algorithms

Chapter 2

2 Introduction

Today's graph networks can have up to billions of edges. For example, the Facebook graph contained 721 million vertices and 68.7 billion edges in 2011 [Uga+11]. In order to process large graph we need scalable graph algorithms: Even presumably fast algorithms like Dijkstra's algorithm for shortest paths [Dij59] can take seconds [Bas+16] on real-world inputs. For more computationally complex problems, a solution cannot be found within hours. In the following chapters we present results for three different graph problems: Maximum independent sets, maximum cuts, and shortest paths. All these problems have in common that they scale to large graphs by using or building some kind of hierarchy in the graph.

For the maximum independent set and maximum cut problems we employ problem size reductions where we take an input graph and remove all the "easy" parts. These reduction techniques are often used in so-called *branch-and-reduce* algorithms where (in addition to traditional branching) a hierarchy of smaller and smaller graphs is built using reductions.

For shortest path problem, we build a hierarchy of the edges of a road network. The hierarchy is given by the importance of a road for long trips. Here, highways are usually preferred over gravel roads. Small streets must still be considered though, especially at the beginning and end of a route.

References and Attribution. This part is based on the conference papers [Hes+20; HLS21a; Fer+20] and [HS19a]. In all of these papers the author of this thesis is (one of) the main author(s). Further information on contributions are provided in the chapters covering these papers. Large segments of this part were copied verbatim from the conference papers or the corresponding technical reports [Hes+19b; HLS21b; Fer+19; HS19b].

2.1 Preliminaries

Here, we give some general definitions and notations used for all our graph algorithms. We provide more specific definitions and notations used only for *some* problems in the respective chapters.

A *graph* is a tuple $G = (V, E)$ with a set of n *vertices* $V = \{0, \dots, n-1\}$ and a set of m *edges* E. To specify the vertex and edge sets of a specific graph G, we use $V(G)$ and $E(G)$, respectively. In the case of *undirected* graphs, edges are subsets

of V of size 2, i. e., $E \subseteq \{\{u, v\} \mid u, v \in V\}$. In the case of *directed* graphs, edges are 2-tuples of vertices, i. e., $E \subseteq V \times V$. We assume that G is *simple*, i. e., it has no self loops or multi-edges. In the rest of this section, any definition made for directed graphs also applies to the undirected case. Definitions using the undirected notation are only used for undirected graphs in this dissertation.

In case of *(edge) weighted* graphs, we additionally use an edge weight function $\omega : E \to \mathbb{R}_{>0}$ (or some other set of numbers). We then sometimes write $G = (V, E, \omega)$. For *unweighted* graphs, the weight for each edge is 1.

The *complement* $\overline{G}$ of a graph G is the graph where every edge becomes a non-edge and every non-edge becomes an edge, i.e., $V(\overline{G}) = V(G)$ and $E(\overline{G}) = (V(G) \times V(G)) \setminus E(G)$.

The *(open) neighborhood* of a vertex $v \in V$ is denoted by $N(v) = \{u \mid \{v, u\} \in E\}$. Furthermore, we denote the *closed neighborhood* of a vertex by $N[v] = N(v) \cup \{v\}$. We define the open and closed neighborhood of a set of vertices $U \subseteq V$ as $N(U) = \bigcup_{u \in U} N(v) \setminus U$ and $N[U] = N(U) \cup U$, respectively. The *degree* of a vertex $v \in V$ is the size of its neighborhood $d(v) = |N(v)|$ and $\Delta = \max_{v \in V} \{d(v)\}$. For a vertex $v \in V$, we further define the *two-neighborhood* $N^2(v) = N(N(v)) \setminus N[v]$.

For a subset of vertices $V_S \subseteq V$, the *(vertex-)induced subgraph* $G[V_S] = (V_S, E_S)$ is given by restricting the edges of G to vertices of V_S, i. e., $E_S = \{\{u, v\} \in E \mid u, v \in V_S\}$. Likewise, for a subset of edges $E_S \subseteq E$, the *edge-induced subgraph* $G[E_S] = (V_S, E_S)$ is given by restricting the vertices of G to the endpoints of edges in E_S, i. e., $V_S = \{u \in V \mid \{u, v\} \in E_S\}$. For a subset of vertices $U \subset V$, we further define $G - U$ as the induced subgraph $G[V \setminus U]$.

A *path* is a sequence of vertices $(v_0, \ldots, v_n)$ such that $(v_i, v_{i+1}) \in E$ for $0 \leq i < n$. The *length* of a path is the sum of its edge weights. A path with $n + 1$ vertices is called an n-path because it contains n edges. A path with $v_0 = v_n$ is called a *cycle* of G. A *cordless* cycle is a cycle with no edges between the vertices that are not part of the cycle, i.e., there are no edges (v_i, v_j) with $j \neq i + 1$. The length of a shortest path with source vertex s and target vertex t is also called the *distance* between s and t, or $\mathrm{dist}(s, t)$.

A subgraph of G induced by a maximal subset of vertices that are connected by a path is called a *connected component*. Furthermore, a graph that only contains one connected component is called *connected*. Likewise, a graph with more than one connected component is called *disconnected*. A subset $S \subset V$ of a connected graph G is called a *vertex separator* if the removal of S from G makes the graph disconnected. A (sub-)graph is *biconnected* if it remains connected after removing any single vertex, i.e., there is no vertex separator of size 1. A *biconnected component* is a maximal subgraph that is biconnected.

Chapter 3

3 Maximum Independent Sets

References and Attribution. This chapter is based on the conference papers [Hes+20] and [HLS21a]. Together with Sebastian Lamm, the author of this thesis is one of the main authors of these paper with editing done by Christian Schulz and Darren Strash for [Hes+20] and Christian Schorr for [HLS21a]. Further information on contributions are provided in the sections covering these papers. Large parts of this chapter were copied verbatim from the conference papers or the corresponding technical reports [Hes+19b; HLS21b].

3.1 Introduction

An *independent set* of a graph $G = (V, E)$ is a set of vertices $I \subseteq V$ of G such that no two vertices in this set are adjacent. The problem of finding such an independent set of maximum cardinality, the *maximum independent set problem*, is a fundamental NP-hard problem [GJS74]. Its applications (either directly or through its complementary problems) cover a wide variety of fields including computer graphics [San+08], network analysis [Put+15], route planning [Kie+10] and computational biology [BW06; Che+08]. In computer graphics for instance, large independent sets can be used to optimize the traversal of mesh edges in a triangle mesh. Further applications stem from its complementary problems minimum vertex cover and maximum clique.

One of the best known techniques for finding maximum independent sets, both in theory [XN17; CKX10] and practice [AI16], are *data reduction algorithms*. These algorithms apply a set of reduction rules to decrease the size of an instance while maintaining the ability to compute an optimal solution afterwards. A recently successful type of data reduction algorithm are so-called *branch-and-reduce algorithms* [AI16], which exhaustively apply a set of reduction rules to compute an irreducible graph. If no further rule can be applied, the algorithm branches into (at least) two smaller subproblems, which are then solved recursively. To make them more efficient in practice, these algorithms also make use of problem-specific upper and lower bounds to quickly prune the search space.

Complementary to independent sets are vertex covers and cliques. Many techniques have been proposed for solving these problems, and papers in the literature usually focus on one of these problems in particular. However, all of these problems are equivalent: a minimum vertex cover C in G is the complement of a maximum

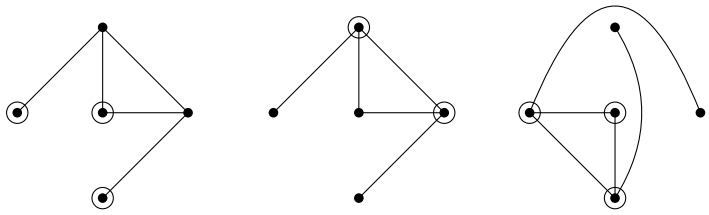


Figure 3.1: A graph G with a Maximum Independent Set (MIS), a Minimum Vertex Cover (MVC), and a Maximum Clique (MC) of the complement graph $\overline{G}$.

independent set $V \setminus C$ in G, which is a maximum clique $V \setminus C$ in $\overline{G}$. Thus, an algorithm that solves one of these problems can be used to solve the others.

3.2 Preliminaries

For general definitions and notations for graphs see Section 2.1. Here we give some more definitions specific to maximum independent sets and its related problems. Note that in this chapter, we consider *undirected*, *unweighted* graphs.

An *independent set* of a graph is a subset of vertices $I \subseteq V$ such that no two vertices of I are adjacent. A *maximum independent set* (MIS) is an independent set of maximum cardinality. Closely related to independent set are vertex covers and cliques. A *vertex cover* is a set of vertices $C \subseteq V$ such that for each edge $\{u, v\} \in E$ either u or v is contained in C. The complement of a (maximum) independent set of a graph G is a *(minimum) vertex cover* (MVC) of G. A *clique* is a subset of vertices $K \subseteq V$ such that all vertices of K are adjacent to each other, i. e., $\forall u, v \in K : \{u, v\} \in E$. Finally, a (maximum) independent set of a graph G is a *(maximum) clique* (MC) in the complement graph $\overline{G}$. Figure 3.1 shows an example graph with the three sets described in this section.

3.3 Related Work

Research results in the area can be found through work on the minimum vertex cover problem and its complementary maximum clique and independent set problems, and can often be categorized depending on the angle of attack. For exact exponential (theoretical) algorithms, the maximum independent set problem is canonically studied, for parameterized algorithms, the minimum vertex cover problem is studied, and the maximum clique problem is normally solved exactly in practice (though there are recent exceptions). However, these problems are only *trivially* different—techniques for solving one problem require only subtle modifications to solve the other two.

Theoretical Exponential-time Algorithms. The maximum independent set problem is most often considered when designing exact (exponential-time) algorithms, and much research has be devoted to reducing the base of the exponential running time. A primary technique is to develop rules to modify the graph, removing or contracting subgraphs that can be solved simply, which reduces the graph to a smaller instance. These rules are referred to as *data reduction rules* (often simplified to *reduction rules* or *reductions*). Reduction rules have been used to reduce the running time of the brute force $O(n^2 2^n)$ algorithm to the $O(2^{n/3})$ time algorithm of Tarjan and Trojanowski [TT77], and to the current best polynomial space algorithm with running time of $O^*(1.1996^n)$ by Xiao and Nagamochi [XN17].

The reduction rules used for these algorithms are often staggeringly simple, including *pendant vertex removal*, *vertex folding* [CKJ01] and *twin* reductions [XN13], which eliminate nearly all vertices of degree three or less from the graph. These algorithms apply reductions during recursion, only branching when the graph can no longer be reduced [FK10], and are referred to as *branch-and-reduce* algorithms. Further techniques used to accelerate these algorithms include *branching rules* [KLR09; FGK09] which eliminate unnecessary branches from the search tree, as well as faster exponential-time algorithms for graphs of small maximum degree [XN17].

Parameterized Algorithms. For parameterized algorithms, we now turn to the minimum vertex cover problem. The most efficient algorithms for computing a minimum vertex cover in both theory and practice repeatedly apply data reduction rules to obtain a (hopefully) much smaller problem instance. If this smaller instance has size bounded by a function of some parameter, it's called a *kernel*, and producing a polynomially-sized kernel gives a fixed-parameter tractable Algorithm in the chosen parameter. Reductions are surprisingly effective for the minimum vertex cover problem. In particular, letting k be the size of a minimum vertex cover, the well-known crown reduction rule produces a kernel of size $3k$ [CFJ05] and the LP-relaxation reduction due to Nemhauser and Trotter [NT75], produces a kernel of size $2k$ [CKJ01]. Chen et al. [CKX10] developed the current best parameterized algorithm for minimum vertex cover, giving a branch-and-reduce algorithm with running time $O(1.2738^k + kn)$ and polynomial space. For more information on the history of vertex cover kernelization, see the recent survey by Fellows et al. [Fel+18].

Exact Algorithms in Practice. The most efficient maximum clique solvers use a branch-and-bound search with advanced vertex reordering strategies and pruning (typically using approximation algorithms for graph coloring, MaxSAT [LFX13] or constraint satisfaction). The long-standing canonical algorithms for finding the maximum clique are the MCS algorithm by Tomita et al. [Tom+10] and the bit-parallel algorithms of San Segundo et al. [Seg+13; SRJ11]. Recently, Li et al. [LJM17] introduced the MoMC algorithm, which uses incremental MaxSAT logic to achieve speed ups of up to 1 000 over MCS. Experiments by Batsyn et al. [Bat+14] show that MCS can be sped up significantly by giving an initial solution found through local search. However, even with these state-of-the-art algorithms, graphs on thousands of vertices remain intractable. For example, a difficult graph on 4 000

vertices required 39 wall-clock hours in a highly-parallel MapReduce cluster, and is estimated to require over a year of sequential computation [XGA13]. Recent clique solvers for sparse graphs investigate applying simple data reduction rules, using an initial clique given by some inexact method [VBB15; SLP16; Cha19]. However, these techniques rarely work on dense graphs, such as the complement graphs that we consider here. A thorough discussion of many results in clique finding can be found in the survey of Wu and Hao [WH15].

Data reductions have been successfully applied in practice to solve many problems that are intractable with general algorithms. Butenko et al. [But+02; But+09] were the first to show that simple reductions could be used to compute exact maximum independent sets on graphs with hundreds vertices for graphs derived from error-correcting codes. Their algorithm works by first applying *isolated clique removal* reductions, then solving the remaining graph with a branch-and-bound algorithm. Later, Butenko and Trukhanov [BT07] introduced the *critical independent set* reduction, which was able to solve graphs produced by the Sanchis graph generator. Larson [Lar07] later proposed an algorithm to find a *maximum* critical independent set, but in experiments it proved to be slow in practice [Str16]. Iwata et al. [IOY14] then showed how to remove a large collection of vertices from a maximum matching all at once.

For the minimum vertex cover problem, it has long been known that two such simple reductions, called *pendant vertex removal* and *vertex folding*, are particularly effective in practice. However, two seminal experimental works explored the efficacy of further reductions. Abu-Khzam et al. [Abu+07] showed that *crown reductions* are as effective (and sometimes faster) in practice than performing the LP relaxation reduction (which, as they show in the paper, removes crowns) on graphs. We briefly note that critical independent sets, together with their neighborhoods, are in fact crowns, and thus in some ways the work of Butenko and Trukhanov [BT07] replicates that by Abu-Khzam et al. [Abu+07], though their experiments are run on different graphs.

Later, Akiba and Iwata [AI16] showed that an extensive collection of advanced data reduction rules (together with branching rules and lower bounds for pruning search) are also highly effective in practice. Their algorithm finds exact minimum vertex covers on a corpus of large social networks with hundreds of thousands of vertices or more in mere seconds. More details on the reduction rules follow in Section 3.3.1.

We briefly note that we considered other reduction techniques that emphasize fast computation at the cost of a larger (irreducible) graph [CLZ17; Str16; HSS18]; however, we did not find them as effective as Akiba and Iwata [AI16] for exactly solving difficult instances. This is somewhat expected, however, since these techniques are optimized to produce fast high-quality solutions when combined with inexact methods such as local search.

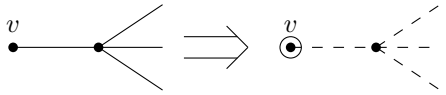


Figure 3.2: Pendant vertex reduction. Vertex v has degree one, so it is added to the MIS and removed from the graph along with its neighbor.

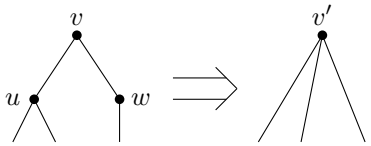


Figure 3.3: Vertex folding reduction. Vertex v has degree two and its neighbors u and w are not connected. Vertices v, u, and w are contracted into a new vertex v'.

3.3.1 Reduction Rules

As both contributions presented in this chapter rely on the use of data reduction rules, we now give a brief description of the reductions used in our contributions—some of them will be explained in more detail later on. Each reduction allows us to choose vertices that are either in some maximum independent set, or for which we can locally choose a vertex in a maximum independent set after solving the remaining graph by following simple rules. If a maximum independent set is found in the reduced graph, then each reduction may be undone, producing a maximum independent set in the original graph. Refer to Akiba and Iwata [AI16] for a more thorough discussion, including implementation details. Our implementation of the reductions is an adaptation of Akiba and Iwata's original code.

Pendant vertices: Any vertex v of degree one, called a *pendant*, it is in some maximum independent set, therefore v and its neighbor u can be removed from G. See Figure 3.2 for an example.

Vertex folding: For a vertex v with degree 2 whose neighbors u and w are not adjacent, either v is in some maximum independent set, or both u and w are in some maximum independent set. Therefore, we can contract u, v, and w to a single vertex v'. If v' is in the computed maximum independent set of the reduced graph, then u and w are in a maximum independent set of the original graph. Otherwise, v is in a maximum independent set. Figure 3.3 shows an example application of vertex folding.

Linear Programming Relaxation: First introduced by Nemhauser and Trotter [NT75] for the vertex packing problem, they present a linear programming relaxation with a half-integral solution (i.e., using only values 0, 1/2, and 1) which can be solved

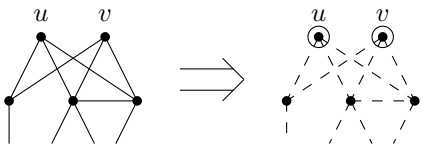


Figure 3.4: Twin reduction, case 1. Vertices u and v have degree three and the same neighborhood with an edge connecting two of their neighbors. Vertices u and v are added to the MIS and removed from the graph along with their neighborhood.

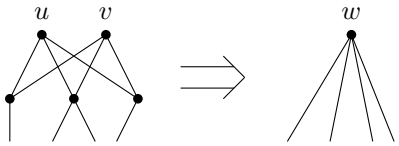


Figure 3.5: Twin reduction, case 2. Vertices u and v have degree three and the same neighborhood with no edges within their neighborhood. Vertices u, v, and their neighborhood are contracted into a new vertex w.

using bipartite matching. Their relaxation may be formulated as follows: maximize $\sum_{v \in V} x_v$, such that for each edge $(u, v) \in E$, $x_u + x_v \leq 1$ and for each vertex $v \in V$, $x_v \geq 0$. There is a maximum independent set containing all vertices with value 1, and are therefore added to the solution and removed from the graph together with their neighbors. We use the further improvement from Iwata et al. [IOY14], which computes a solution whose half-integral part is minimal.

Unconfined [XN13]: Though there are several definitions of an *unconfined* vertex in the literature, we use the simple one from Akiba and Iwata [AI16]. A vertex v is *unconfined* when determined by the following simple algorithm. First, initialize $S = \{v\}$. Then find a $u \in N(S)$ such that $|N(u) \cap S| = 1$ and $|N(u) \setminus N[S]|$ is minimized. If there is no such vertex, then v is confined. If $N(u) \setminus N[S] = \varnothing$, then v is unconfined. If $N(u) \setminus N[S]$ is a single vertex w, then add w to S and repeat the algorithm. Otherwise, v is confined. Unconfined vertices can be removed from the graph, since there always exists a maximum independent set that does not contain unconfined vertices.

Twin [XN13]: Let u and v be vertices of degree 3 with $N(u) = N(v)$. If $G[N(u)]$ has edges, then add u and v to the maximum independent set and remove u, v, $N(u)$, $N(v)$ from G. Otherwise, $N(u)$ may belong to some maximum independent set. We still remove u, v, $N(u)$, $N(v)$ from G, and add a new gadget vertex w to G with edges to u's two-neighborhood (vertices at a distance 2 from u). If w is in the computed maximum independent set, then u's (and v's) neighbors are in some maximum independent set, otherwise u and v are in a maximum independent set.

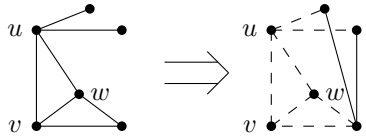


Figure 3.6: Funnel reduction. Vertices u and v are neighbors and $N(v) \setminus \{u\}$ induces a clique. We remove u, v, and $N(v) \cap N(u) = \{w\}$ from the graph and connect all of u's neighbors to all of v's neighbors.

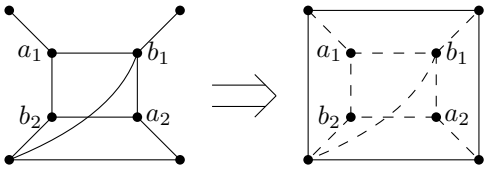


Figure 3.7: Desk reduction. Vertices a_1, b_1, a_2, b_2 induce a cordless 4-cycle with a minimum degree of three and the total number of neighbors of $A = \{a_1, a_2\}$ and $B = \{b_1, b_2\}$ is two. Also, A and B do not share any neighbors except each others. This is reduced by removing the entire 4-cycle and adding edges from each vertex adjacent to an a-vertex to each vertex adjacent to a b-vertex.

Figures 3.4 and 3.5 show example applications for both cases of the twin reduction.

Alternative: Two sets of vertices A and B are said to be *alternatives* if $|A| = |B| \geq 1$ and there exists an maximum independent set I such that $I \cap (A \cup B)$ is either A or B. Then we remove A and B and $C = N(A) \cap N(B)$ from G and add edges from each $a \in N(A) \setminus C$ to each $b \in N(B) \setminus C$. Then we add either A or B to I, depending on which neighborhood has vertices in I. Two structures are detected as alternatives. First, if v and u are neighbors and $N(v) \setminus \{u\}$ induces a complete graph, then $\{u\}$ and $\{v\}$ are alternatives (a *funnel*). Next, if there is a cordless 4-cycle $a_1 b_1 a_2 b_2$ where each vertex has at least degree 3. Then sets $A = \{a_1, a_2\}$ and $B = \{b_1, b_2\}$ are alternatives (called a *desk*) when $|N(A) \setminus B| \leq 2$, $|N(A) \setminus B| \leq 2$, and $N(A) \cap N(B) = \varnothing$. Figures 3.6 and 3.7 show examples of the funnel and desk reduction, respectively.

3.4 WeGotYouCovered: The Winning PACE 2019 Solver

Abstract. *We present the winning solver of the PACE 2019 Implementation Challenge, Vertex Cover Track. Our algorithm uses a portfolio of techniques, including an aggressive kernelization strategy, local search, branch-and-reduce, and a state-of-the-art branch-and-bound solver. Of particular interest is that several of our techniques were* not *from the literature on the vertex over problem: they were originally published to solve the (complementary) maximum independent set and maximum clique problems.*

Aside from illustrating our solver's performance in the PACE 2019 Implementation Challenge, our experiments provide several key insights not yet seen before in the literature. First, kernelization can boost the performance of branch-and-bound clique solvers enough to outperform branch-and-reduce solvers. Second, local search can significantly boost the performance of branch-and-reduce solvers. And finally, somewhat surprisingly, kernelization can sometimes make branch-and-bound algorithms perform worse *than running branch-and-bound alone.*

References and Attribution. This section is based on the conference paper [Hes+20]. Together with Sebastian Lamm, the author of this thesis is one of the main authors of this paper with editing done by Christian Schulz and Darren Strash. The author made major contributions to the algorithms used as well as the time slicing approach. The implementation and experiments were done by the author in close cooperation with Sebastian Lamm. Large parts of this section were copied verbatim from the conference paper or the corresponding technical report [Hes+19b].

Note that this section is written in terms of the minimum vertex cover problem as this was the problem to be solved in the PACE 2019 challenge.

3.4.1 Introduction

To win the Vertex Cover track of the PACE 2019 Implementation Challenge, we deployed a portfolio of solvers, using techniques from the literature on maximum independent sets, minimum vertex covers, and maximum cliques. These include data reduction rules and branch-and-reduce for the minimum vertex cover problem [AI16], iterated local search for the maximum independent set problem [ARW12], and a state-of-the-art branch-and-bound maximum clique solver [LJM17].

Our Results. In this chapter, we describe our techniques and solver in detail and analyze the results of our experiments on the data sets provided by the challenge. Not only do our experiments illustrate the power of the techniques spanning the literature, they also provide several new insights not yet seen before. In particular, kernelization followed by branch-and-bound can outperform branch-and-reduce

solvers; seeding branch-and-reduce by an initial solution from local search can significantly boost its performance; and, somewhat surprisingly, kernelization is sometimes counterproductive: branch-and-bound algorithms can perform significantly worse on the kernel than on the original input graph.

Organization. In Section 3.4.2 we outline each of the techniques that we use, and describe in Section 3.4.3 how we combine all of the techniques in our final solver that scored the most points in the PACE 2019 Implementation Challenge. Lastly, in Section 3.4.4 we perform an experimental evaluation to show the impact of the components used on the final number of instances solved during the challenge.

3.4.2 Techniques

We now describe the techniques that we use in our solver.

3.4.2.a) Kernelization

We use the full collection of data reduction rules explained in Section 3.3.1 whose efficacy was studied by Akiba and Iwata [AI16]. To compute a kernel, Akiba and Iwata [AI16] 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. Following their study we order the reductions as follows: degree-one vertex (i.e., pendant) removal, unconfined vertex removal [XN13], a well-known linear-programming relaxation [IOY14; NT75] (which, consequently, removes crowns [Abu+07]), vertex folding [CKJ01], and twin, funnel, and desk reductions [XN13].

3.4.2.b) Branch-and-Reduce

Branch-and-reduce is a paradigm that intermixes data reduction rules and branching. We use the algorithm of Akiba and Iwata, which exhaustively applies their full suite of reduction rules before branching, and includes a number of advanced branching rules as well as lower bounds to prune search.

Branching. When branching, a vertex of maximum degree is chosen for inclusion into the vertex cover. Mirrors and satellites are detected when branching, in order to eliminate branching on certain vertices. A *mirror* of a vertex v is a vertex $u \in N^2(v)$ such that $N(v) \setminus N(u)$ is a clique or empty. Fomin et al. [FGK09] show that either the mirrors of v or $N(v)$ is in a minimum vertex cover, and we can therefore branch on all mirrors at once. This branching prevents branching on mirrors individually and decreases the size of the remaining graph (and thus the depth of the search tree). A *satellite* of a vertex v is a vertex $u \in N^2(v)$ such that there exists a vertex $w \in N(v)$ such that $N(w) \setminus N[v] = \{u\}$. If a vertex v has no

mirrors, then either v is in a minimum vertex cover or the neighbors of v's satellites are in a minimum vertex cover. Akiba and Iwata [AI16] further introduce *packing* branching, maintaining linear inequalities for each vertex included or excluded from the current vertex cover (called *packing constraints*) throughout recursion; when a constraint is violated, further branching can be eliminated.

Lower Bounds. We briefly remark that Akiba and Iwata [AI16] implement lower bounds to prune the search space. Their lower bounds are based on clique cover, the LP relaxation, and cycle covers (see their paper for further details). The final lower bound used for pruning is the maximum of these three.

3.4.2.c) Branch-and-Bound

Experiments by Strash [Str16] show that the full power of branch-and-reduce is only needed *very rarely* in real-world instances; kernelization followed by a standard branch-and-bound solver is sufficient for many real-world instances. Furthermore, branch-and-reduce does not work well on many synthetic benchmark instances, where data reduction rules are ineffective [AI16], and instead add significant overhead to branch-and-bound. We use a state-of-the-art branch-and-bound maximum clique solver (MoMC) by Li et al. [LJM17], which uses incremental MaxSAT reasoning to prune search, and a combination of static and dynamic vertex ordering to select the vertex for branching. We run the clique solver on the complement graph, giving a maximum independent set from which we derive a minimum vertex cover. In preliminary experiments, we found that a kernel can sometimes be harder for the solver than the original input; therefore, we run the algorithm on both the kernel and on the original graph.

3.4.2.d) Iterated Local Search

Batsyn et al. [Bat+14] showed that if branch-and-bound search is primed with a high-quality solution from local search, then instances can be solved up to thousands of times faster. We use the iterated local search algorithm by Andrade et al. [ARW12] to prime the *branch-and-reduce* solver with a high-quality initial solution. To the best of our knowledge, this has not been tried before. Iterated local search was originally implemented for the maximum independent set problem, and is based on the notion of (j, k)-swaps. A (j, k)-swap removes j nodes from the current solution and inserts k nodes. The authors present a fast linear-time implementation that, given a maximal independent set, can find a $(1, 2)$-swap or prove that none exists. Their algorithm applies $(1, 2)$-swaps until reaching a local maximum, then perturbs the solution and repeats. We implemented the algorithm to find a high-quality solution on *the kernel*. Calling local search on the kernel has been shown to produce a high-quality solution much faster than without kernelization [CLZ17; Dah+16].

3.4.3 Putting it all Together

Our algorithm first runs a preprocessing phase, followed by 4 phases of solvers.

Phase 1. (Preprocessing) Our algorithm starts by computing a kernel of the graph using the reductions by Akiba and Iwata [AI16]. From there we use iterated local search to produce a high-quality solution S_{init} on the (hopefully smaller) kernel.

Phase 2. (Branch-and-Reduce, short) We prime a branch-and-reduce solver with the initial solution S_{init} and run it with a short time limit.

Phase 3. (Branch-and-Bound, short) If Phase 2 is unsuccessful, we run the MoMC clique solver [LJM17] on the complement of the kernel, also using a short time limit[1]. Sometimes kernelization can make the problem harder for MoMC. Therefore, if the first call was unsuccessful we also run MoMC on the complement of the original (unkernelized) input with the same short time limit.

Phase 4. (Branch-and-Reduce, long) If we have still not found a solution, we run branch-and-reduce on the kernel using initial solution S_{init} and a longer time limit. We opt for this second phase because—while most graphs amenable to reductions are solved very quickly with branch-and-reduce (less than a second)—experiments by Akiba and Iwata [AI16] showed that other slower instances either finish in at most a few minutes, or take significantly longer—more than the time limit allotted for the challenge. This second phase of branch-and-reduce is meant to catch any instances that still benefit from reductions.

Phase 5. (Branch-and-Bound, remaining time) If all previous phases were unsuccessful, we run MoMC on the original (unkernelized) input graph until the end of the time given to the program by the challenge. This is meant to capture only the hardest-to-compute instances.

The algorithm time limits (discussed in the next section) and ordering were carefully chosen so that the overall algorithm outputs solutions of the "easy" instances *quickly*, while still being able to solve hard instances.

3.4.4 Experimental Results

We now look at the impact of the algorithmic components on the number of instances solved. Here, we focus on the instances of the PACE 2019 Implementation Challenge, Vertex Cover Track A [DFH19a]. This set contains 200 instances overall, split up into 100 public instances that were known before the competition and 100 private instances that were used to evaluate the solvers. We also summarize

[1] Note that repeatedly checking the time can slow down a highly optimized branch-and-bound solver considerably; we therefore simulate time checking by using a limit on the number of branches.

the results comparing against the second and third best competing algorithms on the private instances during the challenge (the running times can be found at `https://www.optil.io/optilion/problem/3155`).

3.4.4.a) Methodology and Setup

All of our experiments were run on a machine with four sixteen-core Intel Xeon Haswell-EX E7-8867 processors running at 2.5 GHz, 1 TB of main memory, and 32 768 KB of L2-Cache. The machine runs Debian GNU/Linux 9 and Linux kernel version 4.9.0-9. All algorithms were implemented in C++11 and compiled with gcc version 6.3.0 with optimization flag `-O3`. Our source code is publicly available under the MIT license at [Hes+19a] and on github[2]. Each algorithm was run sequentially with a time limit of 30 minutes—the time allotted to solve a single data set in the PACE 2019 Implementation Challenge. Our primary focus is on the total number of instances solved.

3.4.4.b) Evaluation

We now explain the main configuration that we use in our experimental setup. In the following, MoMC runs the MoMC clique solver by Li et al. [LJM17] on the complement of the input graph; RMoMC applies reductions to the input graph exhaustively, and then runs MoMC on the complement of the resulting kernel; LSBnR applies reductions exhaustively, then runs local search to obtain a high-quality solution on the kernel which is used as a initial bound in the branch-and-reduce algorithm that is run on the kernel; BnR applies reductions and then runs the branch-and-reduce algorithm on the kernel (no local search is used to improve an initial bound); FullA is the full algorithm as described in the previous section, using a short time limit of one second and a long time limit of thirty seconds.

Tables 3.1 and 3.2 give an overview of the public instances that each of the solvers solved, including the kernel size, and the minimum vertex cover size for those instances solved by any of the four algorithms. Overall, MoMC can solve 30 out of the 100 instances. Applying reductions first enables RMoMC to solve 68 instances. However, curiously, there are two instances (instances 131 and 157) that MoMC solves, but that RMoMC can not solve. In these cases, kernelization reduced the number of nodes, but *increased* the number of edges. This is due to the *alternative* reduction, which in some cases can create more edges than initially present. This is why we choose to also run MoMC on the unkernelized input graph in FullA (in order to solve those instances as well).

LSBnR solves 55 of the 100 instances. Priming the branch-and-reduce algorithm with an initial solution computed by local search has a significant impact: LSBnR solves 13 more instances than BnR, which solves 42 instances. In particular, using local search to find an initial bound helps to solve large instances in which the initial kernelization step does not reduce the graph fully. Surprisingly, RMoMC solves 26

[2] `https://github.com/KarlsruheMIS/pace-2019`

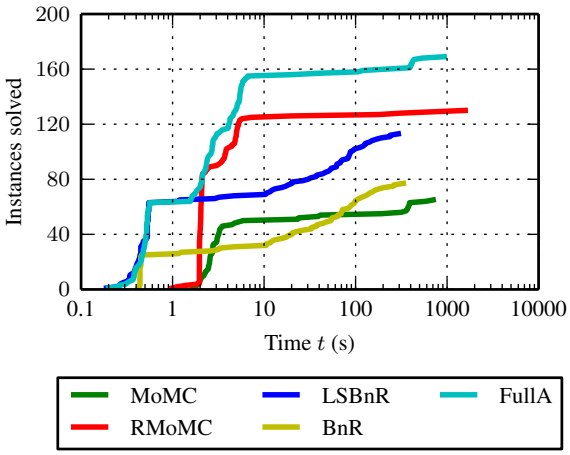


Figure 3.8: Number of instances solved over time by each algorithm over *all* instances. At each time step t, we count each instance solved by the algorithm in at most t seconds.

instances that BnR does not (and even LSBnR is only able to solve one of these instances). To the best of our knowledge, this is the first time that kernelization followed by branch-and-bound is shown to significantly outperform branch-and-reduce. Our full algorithm FullA solves 82 of the 100 instances and, as expected, dominates each of the other configurations. This can be further seen from the plot in Figure 3.8, which shows how many instances each algorithm solves over time (this includes all 100 public and 100 private instances of the challenge). Note that LSBnR and RMoMC solve more instances in narrow time gaps, due to FullA's set up cost and running multiple algorithms. However, FullA quickly makes up for this and overtakes all algorithms at approximately eight seconds. In addition to the 100 public instances, the PACE Implementation Challenge tests all submissions on 100 private instances. Tables 3.3 and 3.4 give detailed per instances results on those instances. The results are similar to the results on the public instances. On the private instances, MoMC can solve 35 out of the 100 instances, RMoMC solves 62, LSBnR solves 58 and BnR solves 35 instances. Our full algorithm FullA solved 87 of the 100 instances, which is 10 more instances than the second-place submission (peaty [PT19], solving 77), and 11 more than the third-place submission (bogdan [Zav19]), solving 76). Our solver dominates these other solvers: with the exception of one graph, our algorithm solves all instances that peaty and bogdan can solve combined.

We briefly describe these two solvers. The peaty solver uses reductions to compute a problem kernel of the input followed by an unpublished maximum weight clique solver on the complement of each of the connected components of the kernel to assemble a solution. The clique solver is similar to MaxCLQ by Li and Quan [LQ10],

but is more general. Local search is used to obtain an initial solution. On the other hand, bogdan implemented a small suite of simple reductions (including vertex folding, isolated clique removal, and degree-one removal) together with a recent maximum clique solver by Szabó and Zavalnij [SZ18].

Lastly, we note that our choice of using MoMC as our chosen branch-and-bound solver is significant on the private instances. Eight instances solved exclusively by our solver are solved in Phase 5, where MoMC is run until the end of the challenge time limit.

3.4.5 Conclusion

We presented the winning solver of the PACE 2019 Implementation Challenge Vertex Cover Track. Our algorithm uses a portfolio of techniques, including an aggressive kernelization strategy with all known reduction rules, local search, branch-and-reduce, and a state-of-the-art branch-and-bound solver. Of particular interest is that several of our techniques were not from the literature on the vertex over problem: they were originally published to solve the (complementary) maximum independent set and maximum clique problems. Lastly, our experiments show the impact of the different solver techniques on the number of instances solved during the challenge. In particular, the results emphasize that data reductions are an important tool to boost the performance of branch-and-bound, and local search is highly effective to boost the performance of branch-and-reduce.

Acknowledgments

We wish to thank the organizers of the PACE 2019 Implementation Challenge for providing us with the opportunity and means to test our algorithmic ideas. We also are indebted to Takuya Akiba and Yoichi Iwata for sharing their original branch-and-reduce source code[3], and to Chu-Min Li, Hua Jiang, and Felip Manyà for not only sharing—but even open sourcing—their code for MoMC at our request[4]. Their solver was of critical importance to our algorithm's success.

[3] https://github.com/wata-orz/vertex_cover

[4] https://home.mis.u-picardie.fr/~cli/EnglishPage.html

Table 3.1: Detailed per instance results for public instances. The columns n and m refer to the number of nodes and edges of the input graph, n' and m' refer to the number of nodes and edges of the kernel graph after reductions have been applied exhaustively, and $|VC|$ refers to the size of the minimum vertex cover of the input graph. We list a '✓' when a solver successfully solved the given instance in the time limit, and '–' otherwise.

inst#	n	m	n'	m'	MoMC	RMoMC	LSBnR	BnR	FullA	$\lvert VC\rvert$
001	6 160	40 207	0	0	–	✓	✓	✓	✓	2 586
003	60 541	74 220	0	0	–	✓	✓	✓	✓	12 190
005	200	819	192	800	✓	✓	✓	✓	✓	129
007	8 794	10 130	0	0	–	✓	✓	✓	✓	4 397
009	38 452	174 645	0	0	–	✓	✓	✓	✓	21 348
011	9 877	25 973	0	0	–	✓	✓	✓	✓	4 981
013	45 307	55 440	0	0	–	✓	✓	✓	✓	8 610
015	53 610	65 952	0	0	–	✓	✓	✓	✓	10 670
017	23 541	51 747	0	0	–	✓	✓	✓	✓	12 082
019	200	884	194	862	✓	✓	✓	✓	✓	130
021	24 765	30 242	0	0	–	✓	✓	✓	✓	5 110
023	27 717	133 665	0	0	–	✓	✓	✓	✓	16 013
025	23 194	28 221	0	0	–	✓	✓	✓	✓	4 899
027	65 866	81 245	0	0	–	✓	✓	✓	✓	13 431
029	13 431	21 999	0	0	–	✓	✓	✓	✓	6 622
031	200	813	198	818	✓	✓	✓	✓	✓	136
033	4 410	6 885	138	471	–	✓	✓	✓	✓	2 725
035	200	884	189	859	✓	✓	✓	✓	✓	133
037	198	824	194	810	✓	✓	✓	✓	✓	131
039	6 795	10 620	219	753	–	✓	✓	✓	✓	4 200
041	200	1 040	200	1 023	✓	✓	✓	✓	✓	139
043	200	841	198	844	✓	✓	✓	✓	✓	139
045	200	1 044	200	1 020	✓	✓	✓	✓	✓	137
047	200	1 120	198	1 080	✓	✓	✓	✓	✓	140
049	200	957	198	930	✓	✓	✓	✓	✓	136
051	200	1 135	200	1 098	✓	✓	✓	✓	✓	140
053	200	1 062	200	1 026	✓	✓	✓	✓	✓	139
055	200	958	194	938	✓	✓	✓	✓	✓	134
057	200	1 200	197	1 139	✓	✓	✓	✓	✓	142
059	200	988	193	954	✓	✓	✓	✓	✓	137
061	200	952	198	914	✓	✓	✓	✓	✓	135
063	200	1 040	200	1 011	✓	✓	✓	✓	✓	138
065	200	1 037	200	1 011	✓	✓	✓	✓	✓	138
067	200	1 201	200	1 174	✓	✓	✓	✓	✓	143
069	200	1 120	196	1 077	✓	✓	✓	✓	✓	140
071	200	984	200	952	✓	✓	✓	✓	✓	136
073	200	1 107	200	1 078	✓	✓	✓	✓	✓	139
075	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
077	200	988	193	954	✓	✓	✓	✓	✓	137
079	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
081	199	1 124	197	1 087	✓	✓	✓	✓	✓	141
083	200	1 215	198	1 182	✓	✓	✓	✓	✓	144
085	11 470	17 408	3 539	25 955	–	–	–	–	–	
087	13 590	21 240	441	1 512	–	✓	–	–	✓	8 400
089	57 316	77 978	16 834	54 847	–	–	–	–	–	
091	200	1 196	200	1 163	✓	✓	✓	✓	✓	145
093	200	1 207	200	1 162	✓	✓	✓	✓	✓	143
095	15 783	24 663	510	1 746	–	✓	–	–	✓	9 755
097	18 096	28 281	579	1 995	–	✓	–	–	✓	11 185
099	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300

Table 3.2: Detailed per instance results for public instances. The columns n and m refer to the number of nodes and edges of the input graph, n' and m' refer to the number of nodes and edges of the kernel graph after reductions have been applied exhaustively, and $|VC|$ refers to the size of the minimum vertex cover of the input graph. We list a '✓' when a solver successfully solved the given instance in the time limit, and '–' otherwise.

inst#	n	m	n'	m'	MoMC	RMoMC	LSBnR	BnR	FullA	$\|VC\|$
101	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
103	15 783	24 663	513	1 752	–	✓	–	–	✓	9 755
105	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
107	13 590	21 240	435	1 500	–	✓	–	–	✓	8 400
109	66 992	90 970	20 336	66 350	–	–	–	–	–	
111	450	17 831	450	17 831	✓	✓	–	–	✓	420
113	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
115	18 096	28 281	573	1 986	–	✓	–	–	✓	11 185
117	18 096	28 281	582	2 007	–	✓	–	–	✓	11 185
119	18 096	28 281	588	2 016	–	✓	–	–	✓	11 185
121	18 096	28 281	579	1 998	–	✓	–	–	✓	11 185
123	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
125	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
127	18 096	28 281	582	2 001	–	✓	–	–	✓	11 185
129	15 783	24 663	507	1 752	–	✓	–	–	✓	9 755
131	2 980	5 360	2 179	6 951	✓	–	–	–	✓	1 920
133	15 783	24 663	507	1 746	–	✓	–	–	✓	9 755
135	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
137	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
139	18 096	28 281	579	1 995	–	✓	–	–	✓	11 185
141	18 096	28 281	576	1 995	–	✓	–	–	✓	11 185
143	18 096	28 281	582	2 001	–	✓	–	–	✓	11 185
145	18 096	28 281	576	1 989	–	✓	–	–	✓	11 185
147	18 096	28 281	567	1 974	–	✓	–	–	✓	11 185
149	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
151	15 783	24 663	501	1 728	–	✓	–	–	✓	9 755
153	29 076	45 570	2 124	16 266	–	–	–	–	–	
155	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
157	2 980	5 360	2 169	6 898	✓	–	–	–	✓	1 920
159	18 096	28 281	582	2 004	–	✓	–	–	✓	11 185
161	138 141	227 241	41 926	202 869	–	–	–	–	–	
163	18 096	28 281	582	2 004	–	✓	–	–	✓	11 185
165	18 096	28 281	576	1 995	–	✓	–	–	✓	11 185
167	15 783	24 663	510	1 746	–	✓	–	–	✓	9 755
169	4 768	8 576	3 458	11 014	–	–	–	–	–	
171	18 096	28 281	576	1 989	–	✓	–	–	✓	11 185
173	56 860	77 264	17 090	55 568	–	–	–	–	–	
175	3 523	6 446	2 723	8 570	–	–	–	–	–	
177	5 066	9 112	3 704	11 797	–	–	–	–	–	
179	15 783	24 663	504	1 740	–	✓	–	–	✓	9 755
181	18 096	28 281	573	1 989	–	✓	✓	–	✓	11 185
183	72 420	118 362	30 340	133 872	–	–	–	–	–	
185	3 523	6 446	2 723	8 568	–	–	–	–	–	
187	4 227	7 734	3 264	10 286	–	–	–	–	–	
189	7 400	13 600	5 802	18 212	–	–	–	–	–	
191	4 579	8 378	3 539	11 137	–	–	–	–	–	
193	7 030	12 920	5 510	17 294	–	–	–	–	–	
195	1 150	81 068	1 150	81 068	–	–	–	–	–	
197	1 534	127 011	1 534	127 011	–	–	–	–	–	
199	1 534	126 163	1 534	126 163	–	–	–	–	–	

Table 3.3: Detailed per instance results for private instances. The columns n and m refer to the number of nodes and edges of the input graph, n' and m' refer to the number of nodes and edges of the kernel graph after reductions have been applied exhaustively, and $|VC|$ refers to the size of the minimum vertex cover of the input graph. We list a '✓' when a solver successfully solved the given instance in the time limit, and '–' otherwise.

inst#	n	m	n'	m'	MoMC	RMoMC	LSBnR	BnR	FullA	$\lvert VC\rvert$
002	51 795	63 334	0	0	–	✓	✓	✓	✓	10 605
004	8 114	26 013	0	0	–	✓	✓	✓	✓	2 574
006	200	751	188	716	✓	✓	✓	✓	✓	126
008	7 537	72 833	0	0	–	✓	✓	✓	✓	3 345
010	199	774	189	756	✓	✓	✓	✓	✓	127
012	53 444	68 044	0	0	–	✓	✓	✓	✓	10 918
014	25 123	31 552	0	0	–	✓	✓	✓	✓	5 111
016	153	802	153	802	–	–	–	–	–	
018	49 212	63 601	0	0	–	✓	✓	✓	✓	10 201
020	57 287	71 155	0	0	–	✓	✓	✓	✓	11 648
022	12 589	33 129	0	0	–	✓	✓	✓	✓	6 749
024	7 620	47 293	0	0	–	✓	✓	✓	✓	4 364
026	6 140	36 767	0	0	–	✓	✓	✓	✓	2 506
028	54 991	67 000	0	0	–	✓	✓	✓	✓	11 211
030	62 853	79 557	0	0	–	✓	✓	✓	✓	13 338
032	1 490	2 680	1 081	3 426	✓	–	–	–	✓	960
034	1 490	2 680	1 090	3 467	✓	✓	–	–	✓	960
036	26 300	41 500	500	3 000	–	✓	✓	✓	✓	16 300
038	786	14 024	460	6 623	✓	✓	✓	✓	✓	605
040	210	625	210	625	✓	✓	–	–	✓	145
042	200	974	200	952	✓	✓	✓	✓	✓	136
044	200	1 186	200	1 147	✓	✓	✓	✓	✓	142
046	200	812	200	812	✓	✓	✓	✓	✓	137
048	200	1 052	198	1 022	✓	✓	✓	✓	✓	138
050	200	1 048	200	1 025	✓	✓	✓	✓	✓	140
052	200	1 019	198	1 000	✓	✓	✓	✓	✓	138
054	200	985	198	951	✓	✓	✓	✓	✓	137
056	200	1 117	200	1 089	✓	✓	✓	✓	✓	141
058	200	1 202	200	1 171	✓	✓	✓	✓	✓	142
060	200	1 147	200	1 118	✓	✓	✓	✓	✓	141
062	199	1 164	199	1 128	✓	✓	✓	✓	✓	141
064	200	1 071	198	1 040	✓	✓	✓	✓	✓	138
066	200	884	198	875	✓	✓	✓	✓	✓	134
068	200	983	198	961	✓	✓	✓	✓	✓	135
070	200	887	198	856	✓	✓	✓	✓	✓	133
072	200	1 204	198	1 176	✓	✓	✓	✓	✓	140
074	200	820	194	785	✓	✓	✓	✓	✓	132
076	26 300	41 500	500	3 000	–	✓	✓	–	✓	16 300
078	11 349	17 739	357	1 245	–	✓	–	–	✓	7 015
080	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
082	200	978	196	956	✓	✓	✓	✓	✓	138
084	13 590	21 240	435	1 503	–	✓	–	–	✓	8 400
086	26 300	41 500	500	3 000	–	✓	✓	–	✓	16 300
088	26 300	41 500	500	3 000	–	✓	✓	–	✓	16 300
090	11 349	17 739	357	1 245	–	✓	–	–	✓	7 015
092	450	17 794	450	17 794	✓	✓	–	–	✓	420
094	5 960	10 720	4 217	13 456	–	–	–	–	–	
096	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
098	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
100	26 300	41 500	500	3 000	–	✓	✓	✓	✓	16 300

Table 3.4: Detailed per instance results for private instances. The columns n and m refer to the number of nodes and edges of the input graph, n' and m' refer to the number of nodes and edges of the kernel graph after reductions have been applied exhaustively, and $|VC|$ refers to the size of the minimum vertex cover of the input graph. We list a '✓' when a solver successfully solved the given instance in the time limit, and '–' otherwise.

inst#	n	m	n'	m'	MoMC	RMoMC	LSBnR	BnR	FullA	$\|VC\|$
102	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
104	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
106	2 980	5 360	2 136	6 809	✓	–	–	–	✓	1 920
108	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
110	98 128	161 357	29 168	140 392	–	–	–	–	–	
112	18 096	28 281	576	1 992	–	✓	–	–	✓	11 185
114	15 783	24 663	504	1 740	–	✓	–	–	✓	9 755
116	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
118	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
120	70 144	116 378	6 029	38 285	–	–	–	–	–	
122	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
124	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
126	18 096	28 281	582	2 001	–	✓	–	–	✓	11 185
128	26 300	41 500	500	3 000	–	–	–	–	–	
130	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
132	15 783	24 663	513	1 755	–	✓	–	–	✓	9 755
134	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
136	18 096	28 281	585	2 007	–	✓	–	–	✓	11 185
138	18 096	28 281	576	1 992	–	✓	–	–	✓	11 185
140	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
142	2 980	5 360	2 180	6 946	✓	–	–	–	✓	1 920
144	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
146	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
148	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
150	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
152	13 590	21 240	438	1 506	–	✓	✓	–	✓	8 400
154	15 783	24 663	504	1 737	–	✓	–	–	✓	9 755
156	450	17 809	450	17 809	✓	✓	–	–	✓	420
158	15 783	24 663	507	1 746	–	✓	–	–	✓	9 755
160	18 096	28 281	576	1 989	–	✓	–	–	✓	11 185
162	50 635	83 075	13 066	63 758	–	–	–	–	–	
164	29 296	46 040	1 210	8 666	–	–	–	–	–	
166	3 278	5 896	2 400	7 643	✓	–	–	–	–	2 112
168	2 980	5 360	2 180	6 943	✓	–	–	–	✓	1 920
170	15 783	24 663	507	1 746	–	✓	–	–	✓	9 755
172	4 025	7 435	3 158	9 863	–	–	–	–	–	
174	2 980	5 360	2 180	6 955	✓	–	–	–	✓	1 920
176	15 783	24 663	501	1 734	–	✓	–	–	✓	9 755
178	18 096	28 281	573	1 995	–	✓	–	–	✓	11 185
180	15 783	24 663	501	1 731	–	✓	–	–	✓	9 755
182	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
184	6 290	11 560	4 904	15 397	–	–	–	–	–	
186	26 300	41 500	500	3 000	–	–	✓	–	✓	16 300
188	6 660	12 240	5 220	16 375	–	–	–	–	–	
190	3 875	7 090	2 997	9 424	–	–	–	–	–	
192	2 980	5 360	2 180	6 941	✓	–	–	–	✓	1 920
194	1 150	80 851	1 150	80 851	✓	✓	–	–	✓	1 100
196	1 534	126 082	1 534	126 082	–	–	–	–	–	
198	1 150	80 072	1 150	80 072	✓	✓	–	–	✓	1 100
200	1 150	80 258	1 150	80 258	✓	✓	–	–	✓	1 100

3.5 Targeted Branching

Abstract. *Previous results in improving the performance of branch-and-reduce solvers for the maximum independent set problem were to a large part achieved by developing new, more practical reduction rules. However, other components that have been shown to have a significant impact on the performance of these algorithms have not received as much attention. One of these is the branching strategy, which determines what vertex is included or excluded in a potential solution. Even now, the most commonly used strategy selects vertices solely based on their degree and does not take into account other factors that contribute to the performance of the algorithm.*

In this work, we develop and evaluate several novel branching strategies for both branch-and-bound and branch-and-reduce algorithms. Our strategies are based on one of two approaches which are motivated by existing research. They either (1) aim to decompose the graph into two or more connected components which can then be solved independently, or (2) try to remove vertices that hinder the application of a reduction rule which can lead to smaller graphs. Our experimental evaluation on a large set of real-world instances indicates that our strategies are able to improve the performance of the state-of-the-art branch-and-reduce algorithm by Akiba and Iwata. To be more specific, our reduction-based packing branching rule is able to outperform the default branching strategy of selecting a vertex of highest degree on 65% *of all instances tested. Furthermore, our decomposition-based strategy based on edge cuts is able to achieve a speedup of* 2.29 *on sparse networks (*1.22 *on all instances).*

References and Attributions. This section is based on the conference paper [HLS21a]. Together with Sebastian Lamm, the author of this thesis is a main author of this paper with editing done by Christian Schorr. The author made major contributions to the design of the branching rules, instance selections as well as the experiment design. In particular, the author made major contributions to the nested dissection-based and packing-based approaches. The ideas for the other reduction-based approaches were developed in close cooperation with Sebastian Lamm. The implementation and experiments were done by Christian Schorr. Large parts of this section were copied verbatim from the conference paper or the corresponding technical report [HLS21b].

3.5.1 Introduction

Due to the practical impact of data reduction, most of the research aimed at improving the performance of branch-and-reduce algorithms so far has been focused on either proposing more practically efficient special cases of already existing rules [CLZ17; Dah+16], or maintaining dependencies between reduction rules to

reduce unnecessary checks [AC20; HSS19a]. However, improving other aspects of branch-and-reduce has been shown to benefit its performance [PvdG21]. The branching strategy in particular has been shown to have a significant impact on the running time [AI16]. Up to now, the most frequently used branching strategy employed in state-of-the-art solvers selects branching vertices solely based on their degree. Other factors, such as the actual reduction rules used during the algorithm are rarely taken into account. Recently, there have been some attempts to incorporate such branching strategies for other problems, e.g., finding a maximum k-plex [Gao+18].

3.5.1.a) Contribution

In this section, we propose and examine several novel strategies for selecting branching vertices. These strategies follow two main approaches that are motivated by existing research: (1) Branching on vertices that decompose the graph into several connected components that can be solved independently. Solving components individually has been shown to substantially improve the performance of branch-and-reduce in practice, especially when the size of the largest component is small [AC20]. (2) Branching on vertices whose removal leads to reduction rules becoming applicable again. In turn, this leads to a smaller reduced graph and thus improved performance. For each approach we present several concrete strategies that vary in their complexity. Finally, we evaluate their performance by comparing them to the aforementioned default strategy used in the state-of-the-art solver by Akiba and Iwata [AI16]. For this purpose we make use of a wide spectrum of instances from different graph classes and applications. Our experiments indicate that our strategies are able to find an optimal solution faster than the default strategy on a large set of instances. In particular, our reduction-based packing rule is able to outperform the default strategy on 65% of all instances. Furthermore, our decomposition-based strategies achieve a speedup of 1.22 (over the default strategy) over all instances.

3.5.2 Related Work

The most commonly used branching strategy for maximum independent sets and minimum vertex covers is to select a vertex of maximum degree. Fomin et al. [FGK09] show that using a vertex of maximum degree that also minimizes the number of edges between its neighbors is optimal with respect to their complexity measure. The algorithm by Akiba and Iwata [AI16] (which we augment with our new branching rules) also uses this strategy. Akiba and Iwata also compare this strategy to branching on a vertex of minimum degree and a random vertex. They show that both of these perform substantially worse than branching on a maximum degree vertex.

Xiao and Nagamochi [XN17] also use this strategy in most cases. For dense subgraphs, however, they use an edge branching strategy: They branch on an edge $\{u, v\}$ where $|N(u) \cap N(v)|$ is sufficiently large (depending on the maximum degree

of the graph) by excluding both u and v in one branch and applying the alternative reduction (see Sections 3.3.1 and 3.5.4.b)) to $\{u\}$ and $\{v\}$ in the other branch.

Bourgeois et al. [Bou+12] use maximum degree branching as long as there are vertices of degree at least five. Otherwise, they utilize specialized algorithms to solve subinstances with an average degree of three or four. Those algorithms perform a rather complex case analysis to find a suitable branching vertex. The analysis is based on exploiting structures that contain 3- or 4-cycles. Branching on specific vertices in such structures often enables further reduction rules to be applied.

Chen et al. [CKX10] use a notion of *good pairs* that are advantageous for branching. They chose these good pairs by a set of rules which are omitted here. They combine these with so-called *tuples* of a set of vertices and the number of vertices from this set that have to be included in an MIS. This information can be used when branching on a vertex contained in that set to remove further vertices from the graph. Akiba and Iwata [AI16] use the same concept in their *packing* rule. Chen et al. combine good pairs, tuples and high degree vertices for their branching strategy.

Most algorithms for MC (e.g., [ST14; Tom+13]) compute a greedy coloring and branch on vertices with a high coloring number. More sophisticated MC algorithms use MaxSAT encodings to prune the set of branching vertices [LFX13; LJM17; LQ10]. Li et al. [LJX15] combine greedy coloring and MaxSAT reasoning to further reduce the number of branching vertices.

Another approach used for MC is using the *degeneracy order* $v_1 < v_2 < \cdots < v_n$ where v_i is a vertex of smallest degree in $G - \{v_1, \dots v_{i-1}\}$. Carraghan and Pardalos [CP90] present an algorithm that branches in descending degeneracy order. Li et al. [LFX13] introduce another vertex ordering using iterative maximum independent set computations (which might be easier than MC on some graphs) and breaking ties according to the degeneracy order.

The Algorithm by Akiba and Iwata [AI16]. We now explain the aspects of Akiba and Iwata's algorithm most relevant to our techniques. Their algorithm is the baseline which we use to develop new branching strategies. Akiba and Iwata repeatedly reduce the instance size using a set of polynomial-time reduction rules and then branch on a vertex once no more reduction rules can be applied. Since branching removes at least one vertex from the graph, more reduction rules might be applicable afterwards. The set of reductions used in their algorithm is described in Section 3.3.1. Some reduction rules are explained in more detail in Section 3.5.4, where we show how to target particular reduction rules when branching. Akiba and Iwata apply the reduction rules in a predefined order. For each rule, their algorithm iterates over all vertices in the graph and checks whether the rule can be applied. If a rule is applied successfully, this process is restarted from the first reduction rule. In order to prune the search space, bounds on the largest possible independent set of a branch are computed. They implement three different methods for determining upper bounds: clique cover, LP relaxation and cycle cover. Additionally, they employ special reduction rules that can be applied during branching. Another

optimization done by their algorithm is to solve connected components separately. We utilize this in Section 3.5.3, where we introduce branching rules that decompose the graph into connected components.

3.5.3 Decomposition Branching

Our first approach to improve the default branching strategy of using a vertex of larges degree found in many state-of-the-art algorithms (including that of Akiba and Iwata [AI16]) is to decompose the graph into several connected components. Subsequently, processing these components individually has been shown to improve the performance of branch-and-reduce in practice [AC20]. To this end, we now present three concrete strategies with varying computational complexity: articulation points, edge cuts and nested dissections.

3.5.3.a) Articulation Points

First, we are concerned with finding single vertices that are able to decompose a graph into at least two separated components. Such points are called *articulation points* (or cut vertices). Articulation points can be computed in linear time $\mathcal{O}(n+m)$ using a simple depth-first search (DFS) algorithm (see Hopcroft and Tarjan [HT73] for a detailed description). In particular, a vertex v is an articulation point if it is either the root of the DFS tree and has at least two children or any non-root vertex that has a child u, such that no vertex in the subtree rooted at u has a back edge to one of the ancestors of v.

For our first branching strategy we maintain a set of articulation points $A \subseteq V$. When selecting a branching vertex, we first discard all invalid vertices from A, i. e., vertices that were removed from the graph by a preceding data reduction step. If this results in A becoming empty, a new set of articulation points is computed on the current graph in linear time. However, if no articulation points exist, we select a vertex based on the default branching strategy. Otherwise, if A contains at least one vertex, an arbitrary one from A is selected as the branching vertex. Figure 3.9 illustrates branching on an articulation point.

Even though this strategy introduces only a small (linear) overhead, finding articulation points can be rare depending on the type of graph. This results in the default branching strategy being selected rather frequently. Furthermore, our preliminary experiments indicate that articulation points are rarely found at higher depths in the branching tree. However, due to their low overhead, we can justify searching for them whenever A becomes empty.

3.5.3.b) Edge Cuts

To alleviate the restrictive nature of finding articulation points, we now propose a more flexible branching strategy based on *(minimal) edge cuts*. In general, we aim

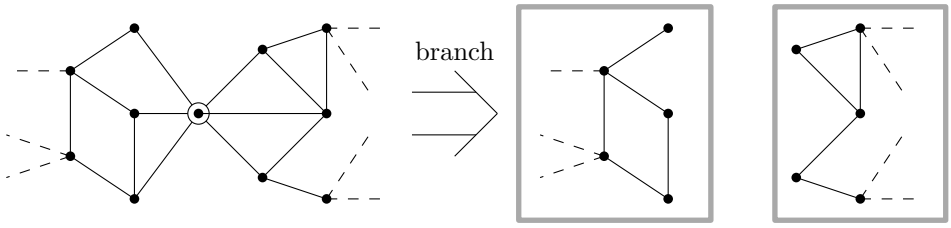


Figure 3.9: Branching on an articulation point (circled vertex) decomposes the graph into two connected components (gray boxes) that can be solved independently. The graphic shows the branch in which the vertex is excluded from the independent set.

to find small vertex separators, i. e., a set of vertices whose removal disconnects the graph. We do so by making use of minimum edge cuts.

A *cut* (S, T) is a partitioning of V into two sets S and $T = V \setminus S$. Furthermore, a cut is called minimum if its *cut set* $C = \{\{u, v\} \in E \mid u \in S, v \in T\}$ has minimal cardinality. However, in practice, finding minimum cuts often yields trivial cuts with either S or T only consisting of a single vertex with minimum degree. Thus, we are interested in finding *s-t-cuts*, i. e., cuts where S and T contain specific vertices $s, t \in V$. Finding these cuts can be done efficiently in practice, e. g., using a preflow push algorithm [GT88]. However, selecting the vertices s and t to ensure reasonably balanced cuts can be tricky. Natural choices include random vertices, as well as vertices that are far apart in terms of their shortest path distance. Our preliminary experiments indicate that selecting random vertices of maximum degree for s and t seems to produce the best results. Finally, to derive a vertex separator from a cut, one can compute a minimum vertex cover on the bipartite graph induced by the cut set, e. g., using the Hopcroft-Karp algorithm [HK73]. This separator can then be used to select branching vertices. In particular, we continuously branch on vertices from the separator.

Overall, our second strategy works similar to the first one: We maintain a set of possible branching vertices that were selected by computing a minimum s-t-cut and turning it into a vertex separator. Vertices that were removed by data reduction are discarded from this set and once it is empty a new cut computation is started. However, in contrast to the first strategy, finding a set of suitable branching vertices is much more likely. In order to avoid separators that contain too many vertices, and thus would require too many branching steps to disconnect the graph, we only keep those that do not exceed a certain size and balance threshold. The specific values for these thresholds are presented in Section 3.5.5.b). Finally, if no suitable separator is found, we use the default branching strategy (branching on a vertex of largest degree). Furthermore, in this case we do not try to find a new separator for a fixed number of branching steps as finding one is both unlikely and costly.

3.5.3.c) Nested Dissection

Both of our previous strategies dynamically maintain a set of branching vertices. Even though this comes at the advantage that most of the computed vertices remain viable candidates for some branching steps, it introduces a noticeable overhead. To alleviate this, our last strategy uses a static ordering of possible branching vertices that is computed once at the beginning of the algorithm. For this purpose we make use of a *nested dissection ordering* [Geo73].

A nested dissection ordering of the vertices of a graph G is obtained by recursively computing balanced bipartitions (A, B) and a vertex separator S, that separates A and B. The actual ordering is then given by concatenating the orderings of A and B followed by the vertices of S. Thus, if we select branching vertices based on the reverse of a nested dissection ordering, we continuously branch on vertices that disconnect the graph into balanced partitions. We compute such an ordering once, after finishing the initial data reduction phase.

There are two main optimizations that we use when considering the nested dissection ordering. First, we limit the number of recursive calls during the nested dissection computation, because we noticed that vertices at the end of the ordering seldom lead to a decomposition of the graph. This is due to the graph structure being changed by data reduction which can lead to separators becoming invalid. Furthermore, similar to the edge-cut-based strategy, we limit the size of separators considered during branching using a threshold. Again, this is done to ensure that we do not require too many branching steps to decompose the graph. The specific value for this size threshold is given in Section 3.5.5.b). If any separator in the nested dissection exceeds this threshold, we use the default branching strategy.

3.5.4 Reduction Branching

Our second approach to selecting good branching vertices is to choose a vertex whose removal will enable the application of new reduction rules. During every reduction step we find a list of candidate vertices to branch on. The following sections will demonstrate how we identify such branching candidate vertices with little computational overhead in practice. For an easier overview we will also repeat the reduction rules used here. Out of the candidates found we then select a vertex of maximum degree. If the degree of all candidate vertices lies below a threshold (defined in Section 3.5.5.b)) or no candidate vertices were found, we fall back to branching on a vertex of maximum degree. The rational here is that a vertex of large degree changes the structure of the graph more than a vertex of small degree even if that vertex is guaranteed to enable the application of a reduction rule. Also, our current strategies (except the packing-based rule in Section 3.5.4.d)) only enable the application of the targeted reduction rule in the branch that excludes the vertex from the independent set, the *excluding branch*. However, in the case that includes it into the independent set (*including branch*) all neighbors are removed from the

graph as well because they already have an adjacent vertex in the solution. Thus, in both branches multiple vertices are removed.

We also performed some preliminary experiments with storing the candidate vertices in a priority queue without resetting after every branch. However, changes were too frequent for this approach to be faster because of the high amount of priority queue operations.

3.5.4.a) Almost Twins

The first reduction we target is the *twin* reduction by Xiao and Nagamochi [XN13]:

Definition 3.1

(Twins [XN13]) In a graph $G = (V, E)$ two vertices u and v are called twins if $N(u) = N(v)$ and $d(u) = d(v) = 3$.

Theorem 3.2

(Twin Reduction [XN13]) In a graph $G = (V, E)$ let vertices u and v be twins. If there is an edge among $N(u)$, then there is always an MIS that includes $\{u, v\}$ and therefore excludes $N(u)$. Otherwise, let $G' = (V', E')$ be the graph with $V' = (V \setminus N[\{u, v\}]) \cup \{w\}$ where $w \notin V$ and $E' = (E \cap \binom{V'}{2}) \cup \{\{w, x\} \mid x \in N^2(u)\}$ and let I' be an MIS in G'. Then,

$$I = \begin{cases} I' \cup \{u, v\} & \text{, if } w \notin I' \\ (I' \setminus \{w\}) \cup N(u) & \text{, else} \end{cases}$$

is an MIS in G.

We now define *almost twins* as follows:

Definition 3.3

(Almost Twins) In a graph $G = (V, E)$ two non adjacent vertices u and v are called almost twins if $d(u) = 4$, $d(v) = 3$ and $N(v) \subseteq N(u)$ (i. e., $N(u) = N(v) \cup \{w\}$).

Clearly, after removing w, u and v are twins so we can apply the twin reduction. Finding almost twins can be done while searching for twins: The original algorithm checks for each vertex v of degree-3 whether there is a vertex $u \in N^2(v)$ with $d(u) = 3$ and $N(u) = N(v)$. We augment this algorithm by simultaneously also searching for $u \in N^2(v)$ with $d(u) = 4$ and $N(v) \subset N(u)$. This induces about the same computational cost for degree-4 vertices in $N^2(v)$ as for degree 3 vertices. While there might be instances where this causes high overhead, we expect the practical slowdown to be small. Figure 3.10 illustrates branching for almost twins.

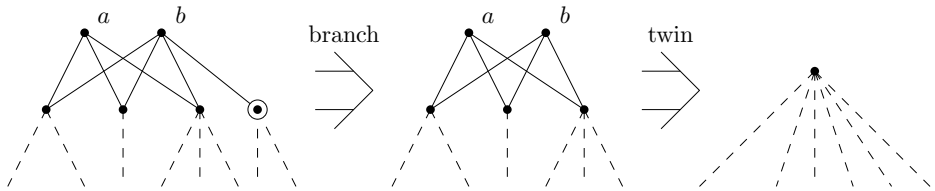


Figure 3.10: Vertices a and b are almost twins. After branching on the circled vertex they become twins (in the excluding branch) and can be reduced.

3.5.4.b) Almost Funnels

Next, we consider the *funnel* reduction which is a special case of the *alternative* reduction by Xiao and Nagamochi [XN13]:

Definition 3.4

(Alternative Sets [XN13]) In a graph $G = (V, E)$ two non empty, disjoint subsets $A, B \subseteq V$ are called alternatives if $|A| = |B|$ and there is an MIS I in G such that $I \cap (A \cup B)$ is either A or B.

Theorem 3.5

(Alternative Reduction [XN13]) In a graph $G = (V, E)$ let A and B be alternative sets. Let $G' = (V', E')$ be the graph with $V' = V \setminus (A \cup B \cup (N(A) \cap N(B)))$ and $E' = \{\{u, v\} \in E \mid u, v \in V'\} \cup \{\{u, v\} \mid u \in N(A) \setminus N[B], v \in N(B) \setminus N[A]\}$ and let I' be an MIS in G'. Then,

$$I = \begin{cases} I' \cup A & \text{, if } (N(A) \setminus N[B]) \cap I' = \varnothing \\ I' \cup B & \text{, else} \end{cases}$$

is an MIS in G.

Note that the alternative reduction adds new edges between existing vertices of the graph which might not be beneficial in every case. To counteract this, the algorithm by Akiba and Iwata [AI16] only uses special cases, one of which is the funnel reduction:

Definition 3.6

(Funnel [XN13]) In a graph $G = (V, E)$ two adjacent vertices u and v are called funnels if $G[N(v) \setminus \{u\}]$ is a complete graph, i.e, if $N(v) \setminus \{u\}$ is a clique.

Theorem 3.7

(Funnel Reduction [XN13]) In a graph $G = (V, E)$ let u and v be funnels. Then, $\{u\}$ and $\{v\}$ are alternative sets.

Again, we define a structure that is covered by the funnel reduction after removal of a single vertex:

Definition 3.8

(Almost Funnel) In a graph $G = (V, E)$ two adjacent vertices u and v are called almost funnels if u and v are not funnels and there is a vertex w such that $N(v) \setminus \{u, w\}$ induces a clique.

By removing w, u and v become funnels. The original funnel algorithm checks whether u and v are funnels by iterating over the vertices in $N(v) \setminus \{u\}$ and checking whether they are adjacent to *all* previous vertices. Once a vertex is found that is not adjacent to all previous vertices, the algorithm concludes that u and v are not funnels and terminates. We augment this algorithm by not immediately terminating in this case. Instead, we consider the following two cases: Either the current vertex w is not adjacent to at least two of the previous vertices. In this case, we can check whether $N(v) \setminus \{u, w\}$ induces a clique. In the second case, w is adjacent to all but one previous vertex w'. In this case, both w and w' might be candidate branching vertices. Thus, we check whether $N(v) \setminus \{u, w\}$ or $N(v) \setminus \{u, w'\}$ induce a clique. This adds up to two additional clique checks (of slightly smaller size) to the one clique check in the original algorithm.

3.5.4.c) Almost Unconfined

The core idea of the *unconfined* reduction by Xiao and Nagamochi [XN13] is to detect vertices not required for an MIS that can therefore be removed from the graph by algorithmically contradicting the assumption that every MIS contains the vertex.

Definition 3.9

(Child, Parent [XN13]) In a graph $G = (V, E)$ with an independent set I, a vertex v is called a child of I if $|N(v) \cap I| = 1$ and the unique neighbor of v in I is called the parent of v.

Algorithm 3.1 shows the algorithm used by Akiba and Iwata [AI16] to detect so called *unconfined* vertices.

Theorem 3.10

(Unconfined Reduction [XN13]) In a graph $G = (V, E)$, if Algorithm 3.1 returns true for an unconfined vertex v, then there is always an MIS that does not contain v.

Again, we define a vertex to be almost unconfined:

Definition 3.11

(Almost Unconfined) In a graph $G = (V, E)$ a vertex v is called almost unconfined if v is not unconfined but there is a vertex w such that v is unconfined in $G - \{w\}$.

Algorithm 3.1 : Unconfined – Xiao and Nagamochi [XN13]

Input : A graph G, a vertex v

```
1 Unconfined(G, v) begin
2     S ← {v}
3     while S has child u with |N(u) \ N[S]| ≤ 1 do
4         if |N(u) \ N[S]| = 0 then
5             return true
6         else
7             {w} ← N(u) \ N[v]          — by assumption w also has to
8             S ← S ∪ {w}                — be contained in every MIS
9     return false
```

Output : true if v is unconfined, false otherwise

Here, we only present an augmentation that detects *some* almost unconfined vertices. In particular, if at any point during the algorithm there is only *one* extending child, i.e., a child u of S with $N(u) \setminus N[S] = \{w\}$, then removal of w makes v unconfined. During Algorithm 3.1 we collect all these vertices w and add them to the set of candidate branching vertices if the algorithm cannot already remove v. This only adds the overhead of temporarily storing the potential candidates and adding them to the actual candidate list if v is not removed.

3.5.4.d) Almost Packing

The core idea behind the packing rule by Akiba and Iwata [AI16] is that when the excluding branch of a vertex v is selected, one can assume that *no* maximum independent set contains v. Otherwise, if there is a maximum independent set that contains v, the algorithm finds it in the branch including v. Based on the assumption that no maximum independent set includes a vertex v, constraints for the remaining vertices can be derived. For example, a maximum independent set that does not contain v has to include at least two neighbors of v. The corresponding constraint is $\sum_{u \in N(v)} x_u \geq 2$, where x_u is a binary variable that indicates whether a vertex is included in the current solution. Otherwise, we will find a solution of the same size in the branch including v. The algorithm creates such constraints when branching or reducing, and updates them accordingly during the data reductions and branching steps. When a vertex v is eliminated from the graph, x_v gets removed from all constraints. If v is included into the current solution, the corresponding right sides are also decreased by one.

A constraint $\sum_{v \in S \subset V} x_v \geq k$ can be utilized in two reductions. Firstly, if k is equal to the number of variables $|S|$, all vertices from S have to be included into the current solution. If there are edges between vertices from S, then no valid solution can include all vertices from S, so the branch is pruned. Secondly, if there is a vertex

v such that $|S| - |N(v) \cap S| < k$, then v has to be excluded from the current solution. If $k > |S|$, the constraint can not be fulfilled and the current branch is pruned.

In our branching strategy we target both reductions. If there is a constraint $\sum_{v \in S \subset V} x_v \geq k$, where $|S| = k + 1$, excluding any vertex of S from the solution or including a vertex of S that has one neighbor in S enables the first reduction. Thus, we consider all vertices in S for branching. Note that including a vertex from S that has more than one neighbor in S makes the constraint unfulfillable and the branch is pruned.

If there is a constraint $\sum_{v \in S \subset V} x_v \geq k$ and a vertex v, such that $k = |S| - |N(v) \cap S|$, excluding any vertex of $S \setminus N(v)$ from the solution or including a vertex of $S \setminus N(v)$ that has at least one neighbor in $S \setminus N(v)$ enables the second reduction. Thus, we consider all vertices in $S \setminus N(v)$ for branching.

Note that in contrast to our previous reduction-based branching rules, packing reductions can also be applied in the including branch in many cases.

Detecting these branching candidates can be done with small constant overhead whilst performing the packing reduction.

3.5.5 Experimental Evaluation

In this section, we present the results of our experimental evaluation. Tables and figures here show aggregated results. For detailed results for all of our algorithms across all instances, see Section 3.5.7.

3.5.5.a) Experimental Environment

We augment a C++-adaptation of the algorithm by Akiba and Iwata [AI16] with our branching strategies and compile it with g++ 9.3.0 using full optimizations (`-O3`). Our code is publicly available on GitHub[5]. We execute all our experiments on a machine with 4 8-core Intel Xeon E5-4640 CPUs (2.4 GHz) and 512 GiB DDR3-PC1600 RAM running Ubuntu 20.04.1 with Linux Kernel 5.4.0-64. To speed up our experiments we use two identical machines and run at most 8 instances at once on the same machine (using the same machine for all algorithms on a specific instance). All numbers reported are arithmetic means of three runs with a timeout of ten hours.

3.5.5.b) Algorithm Configuration

We use a C++ adaptation of the implementation by Akiba and Iwata [AI16] in its default configuration as a basis for our algorithm. During preliminary experiments we found suitable values for the parameters of our techniques. These experiments were run on a subset of our total instance set. We use the geometric mean over all instances of the speedup over the default branching strategy as a basis for the following decisions: for the technique based on edge cuts, we only

[5] `https://github.com/Hespian/CutBranching`

use cuts that contain at most 25 vertices and where the smaller side of the cut contains at least ten percent of the remaining vertices. If no suitable separator is found, we skip ten branching steps. For computing nested dissections, we use InertialFlowCutter [Got+19] with the KaFFPa [SS13] backend. The KaFFPa partitioner is configured to use the *strong* preset with a fixed seed of 42. For branching, we use three levels of nested dissections with a minimum balance of at least 40% of the vertices in the smaller part of each dissection. Furthermore, we only use the nested dissection if separators contain at most 50 vertices. For the reduction-based branching rules, we fall back to the default branching strategy if all candidates have a degree of less than $\Delta - k$. In the case of twin-, funnel- and unconfined-reduction-based branching strategies we choose k as 2. For the packing-reduction-based branching rule, k is set to 5 and for the combined branching rule, k is set to 4.

3.5.5.c) Instances

We use instances from several sources: The "easy" instances used for the PACE 2019 Challenge on Minimum Vertex Cover [DFH19b]. Complements of Maximum Clique instances from the second DIMACS Implementation Challenge [Joh93] and sparse instances from the Stanford Network Analysis Project (SNAP) [LK14], the 9th DIMACS Implementation Challenge on Shortest Paths [DGJ09] and the Network Data Repository [RA15]. Detailed instance information can be found in Table 3.5. Directed instances were converted into undirected graphs by ignoring the direction of edges and removing duplicates. Our original set of instances contained the first 80 PACE instances, 53 DIMACS instances and 34 sparse networks. From these instances, we excluded all instances that (1) required no branches, (2) on which all techniques had a running time of less than 0.1 seconds, or (3) on which no technique was able to find a solution within 10 hours. The remaining set of instances is composed of 48 PACE instances, 37 DIMACS instances and 16 sparse networks.

3.5.5.d) Plot Types

Figures 3.11 and 3.12 show performance profiles [DM02] of the running time and number of branches of our decomposition-based branching strategies: Let $\mathcal{T}$ be the set of all techniques we want to compare, $\mathcal{I}$ the set of instances, and $t_T(I)$ the running time/number of branches of technique $T \in \mathcal{T}$ on instance $I \in \mathcal{I}$. The y-axis shows for each technique T the fraction of instances for which $t_T(I) \leq \tau \cdot \min_{T' \in \mathcal{T}} t_{T'}(I)$, where τ is shown on the x-axis. For $\tau = 1$, the y-axis shows the fraction of instances on which a technique performs best. Note that these plots compare the performance of a technique relative to the best performing technique and do not show a ranking of all techniques. Instances that were not finished by a technique within the time limit are marked with ⏲.

The top plot always shows the performance profiles in terms of their running times on our benchmark machines. The bottom plot shows the number of branching

Table 3.5: Number of vertices $|V|$ and edges $|E|$ for each graph.

PACE [DFH19b] instances:		
Graph	$\|V\|$	$\|E\|$
05	200	798
06	200	733
10	199	758
16	153	802
19	200	862
31	200	813
33	4 410	6 885
35	200	864
36	26 300	41 500
37	198	808
38	786	14 024
39	6 795	10 620
40	210	625
41	200	1 023
42	200	952
43	200	841
44	200	1 147
45	200	1 020
46	200	812
47	200	1 093
48	200	1 025
49	200	933
50	200	1 025
51	200	1 098
52	200	992
53	200	1 026
54	200	961
55	200	938
56	200	1 089
57	200	1 160
58	200	1 171
59	200	961
60	200	1 118
61	200	931
62	199	1 128
63	200	1 011
64	200	1 042
65	200	1 011
66	200	866
67	200	1 174
68	200	961
69	200	1 083
70	200	860
71	200	952
72	200	1 167
73	200	1 078
74	200	805
77	200	961

DIMACS [Joh93] instances:		
Graph	$\|V\|$	$\|E\|$
C125.9	125	787
MANN_a27	378	702
MANN_a45	1 035	1 980
brock200_1	200	5 066
brock200_2	200	10 024
brock200_3	200	7 852
brock200_4	200	6 811
gen200_p0.9_44	200	1 990
gen200_p0.9_55	200	1 990
hamming8-4	256	11 776
johnson16-2-4	120	1 680
keller4	171	5 100
p_hat1000-1	1 000	377 247
p_hat1000-2	1 000	254 701
p_hat1500-1	1 500	839 327
p_hat300-1	300	33 917
p_hat300-2	300	22 922
p_hat300-3	300	11 460
p_hat500-1	500	93 181
p_hat500-2	500	61 804
p_hat500-3	500	30 950
p_hat700-1	700	183 651
p_hat700-2	700	122 922
san1000	1 000	249 000
san200_0.7_1	200	5 970
san200_0.7_2	200	5 970
san200_0.9_1	200	1 990
san200_0.9_2	200	1 990
san200_0.9_3	200	1 990
san400_0.5_1	400	39 900
san400_0.7_1	400	23 940
san400_0.7_2	400	23 940
san400_0.7_3	400	23 940
sanr200_0.7	200	6 032
sanr200_0.9	200	2 037
sanr400_0.5	400	39 816
sanr400_0.7	400	23 931

Sparse networks:			
Graph	$\|V\|$	$\|E\|$	source
as-skitter	1 696 415	11 095 298	[LK14]
baidu-relatedpages	415 641	2 374 044	[RA15]
bay	321 270	397 415	[DGJ09]
col	435 666	521 200	[DGJ09]
fla	1 070 376	1 343 951	[DGJ09]
hudong-internallink	1 984 484	14 428 382	[RA15]
in-2004	1 382 870	13 591 473	[RA15]
libimseti	220 970	17 233 144	[RA15]
musae-twitch_DE	9 498	153 138	[LK14]
musae-twitch_FR	6 549	112 666	[LK14]
petster-fs-dog	426 820	8 543 549	[RA15]
soc-LiveJournal1	4 847 571	42 851 237	[LK14]
web-BerkStan	685 230	6 649 470	[LK14]
web-Google	875 713	4 322 051	[LK14]
web-NotreDame	325 730	1 090 108	[LK14]
web-Stanford	281 903	1 992 636	[LK14]

points where the algorithm had to consider both subproblems without being able to prune the search after finishing the first branch.

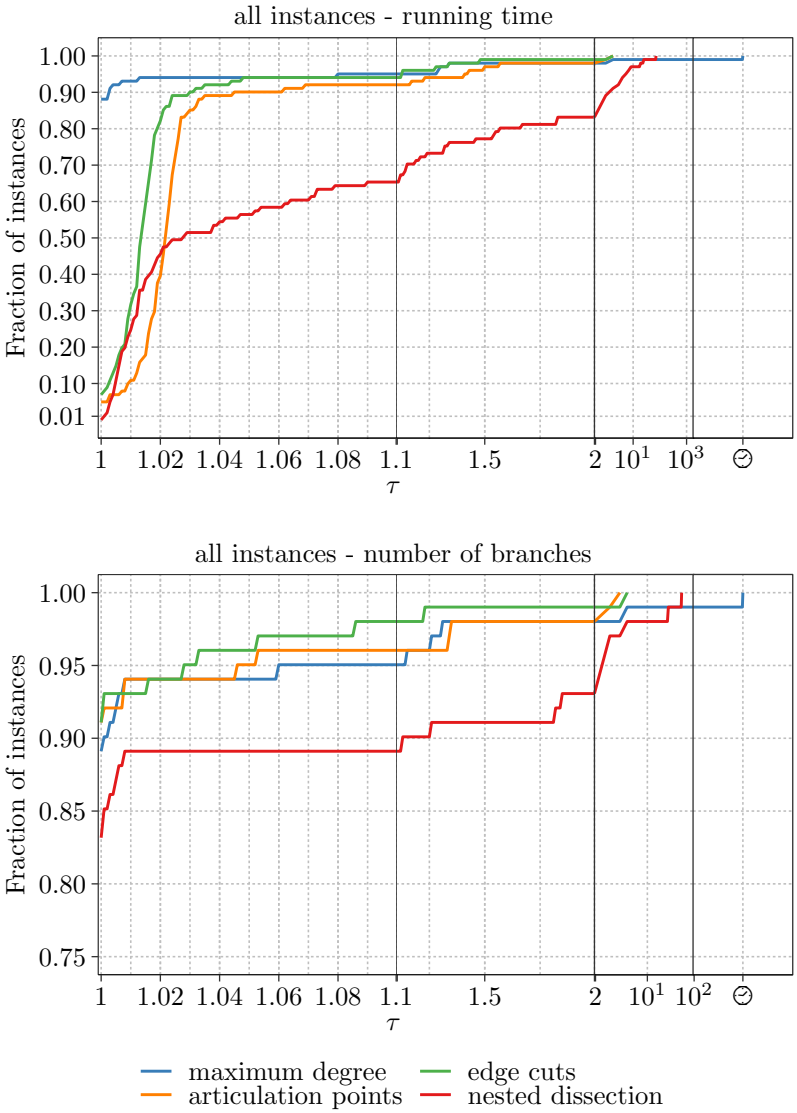


Figure 3.11: Performance plots for decomposition-based branching strategies

3.5.5.e) Decomposition Branching

The running time plot in Figure 3.11 shows that for most instances, the default strategy of branching on a vertex of maximum degree outperforms our decomposition-based approaches. However, for instances that have suitable candidates for decom-

Table 3.6: Speedup of our decomposition-based techniques over maximum degree branching.

	PACE	DIMACS	Sparse net.	All Instances
articulation points	0.99	0.99	2.17	1.20
edge cuts	1.00	0.99	**2.29**	**1.22**
nested dissections	1.00	0.99	2.15	1.21

position, such as sparse networks, substantial speedups compared to the default strategy can be seen. To be more specific, assigning a time of ten hours (our timeout threshold) for unfinished instances, we achieve a total speedup[6] of 2.15 to 2.29 over maximum degree branching for our decomposition-based techniques on sparse networks (see Table 3.6). In particular, there is one instance (web-stanford) that causes a timeout with the default strategy but can be solved in 8 (articulation points) to 43 (nested dissections) seconds using a decomposition-based approach. Table 3.6 shows that overall, our technique using edge cuts seems to be the most beneficial, achieving an overall speedup of 22% over maximum degree. Finally, Figure 3.11 shows that most running times are only slightly slower than the default strategy with a few instances showing a speedup. This is mainly because the number of branches required to solve the instances does not change in most cases and most of the running time difference is caused by the overhead from searching for branching vertices.

3.5.5.f) Reduction Branching

Figure 3.12 shows the performance profiles (see Section 3.5.5.d)) for our reduction-based branching strategies. Here, we see that targeting the packing reduction results in the fastest time for the most number of instances. In fact, targeting the packing reduction performs better than maximum degree branching on all but 3 PACE instances, achieving a speedup of 34% (Table 3.7) on these instances. On the DIMACS instances, performance is closer to that of maximum degree with an overall speedup of 4%. On sparse networks, packing is only faster than maximum degree branching on 6 out of 16 instances but still achieves an overall speedup of 31% due to being considerably faster on some of the longer running instances. The performance of our packing-based technique might be explained by its property of enabling a reduction in both the including and the excluding branch, while our other reduction-based techniques only enable a reduction in the excluding branch. Our funnel-based technique is faster than maximum degree branching for all but 4 of the PACE instances, resulting in a speedup of 14% on these instances but only a 2% speedup over all instances due to slightly slower running times on the other instance classes. We also show results for a strategy that targets all reduction

[6] calculated by dividing the running times to solve all instances for two algorithms, excluding instances unsolved by both algorithms

Table 3.7: Speedup of our reduction-based techniques over maximum degree branching.

	PACE	DIMACS	Sparse net.	All Instances
Twin	1.00	1.00	0.97	0.99
Funnel	1.14	0.99	0.98	1.02
Unconfined	0.79	1.00	0.86	0.92
Packing	**1.34**	**1.04**	**1.31**	**1.16**
Combined	1.14	1.03	1.30	1.12

rules described in Section 3.5.4 (called *combined*). Even though this approach leads to the second lowest number of branches for most instances, the time required to identify candidate vertices for all reduction rules causes too big of an overhead to be competitive. In fact, preliminary experiments showed that the number of branches is still small for a technique that only combines twin-, funnel- and unconfined-based branching. Optimizing the algorithms to identify candidate vertices could lead to making this combined strategy competitive.

3.5.6 Conclusion and Future Work

In this section, we presented several novel branching strategies for the maximum independent set problem. Our strategies either follow a decomposition-based or reduction-rule-based approach. The decomposition-based strategies make use of increasingly sophisticated methods of finding vertices that are likely to decompose the graph into two or more connected components. Even though these strategies often come with a non negligible overhead, they work well for graphs that have a suitable structure, such as social networks. For instances that still favor the default branching strategy of branching on the vertex of highest degree, our reduction-rule-based strategies provide a smaller but more consistent speedup. These rules aim to facilitate the application of reduction rules which leads to smaller graphs that can be solved more quickly.

Overall, using one of our proposed strategies allows us to find the optimal solution the fastest for most instances tested. However, deciding which particular strategy to use for a given instance still remains an open problem. Finding suitable graph characteristics to do so provides an interesting opportunity for future work. Furthermore, our experimental evaluation on a combined approach that tries to use all reduction-rule-based strategies at the same time achieves a smaller number of branches than the default strategy for a large set of instances. However, the running time of this approach still suffers from frequent checks whether a particular vertex is a potential branching vertex. A more sophisticated and incremental way of tracking when a vertex becomes a branching vertex might provide substantial performance benefits. In turn, this might lead to a branching strategy that consistently outperforms branching on the vertex of highest degree independent of the instance type.

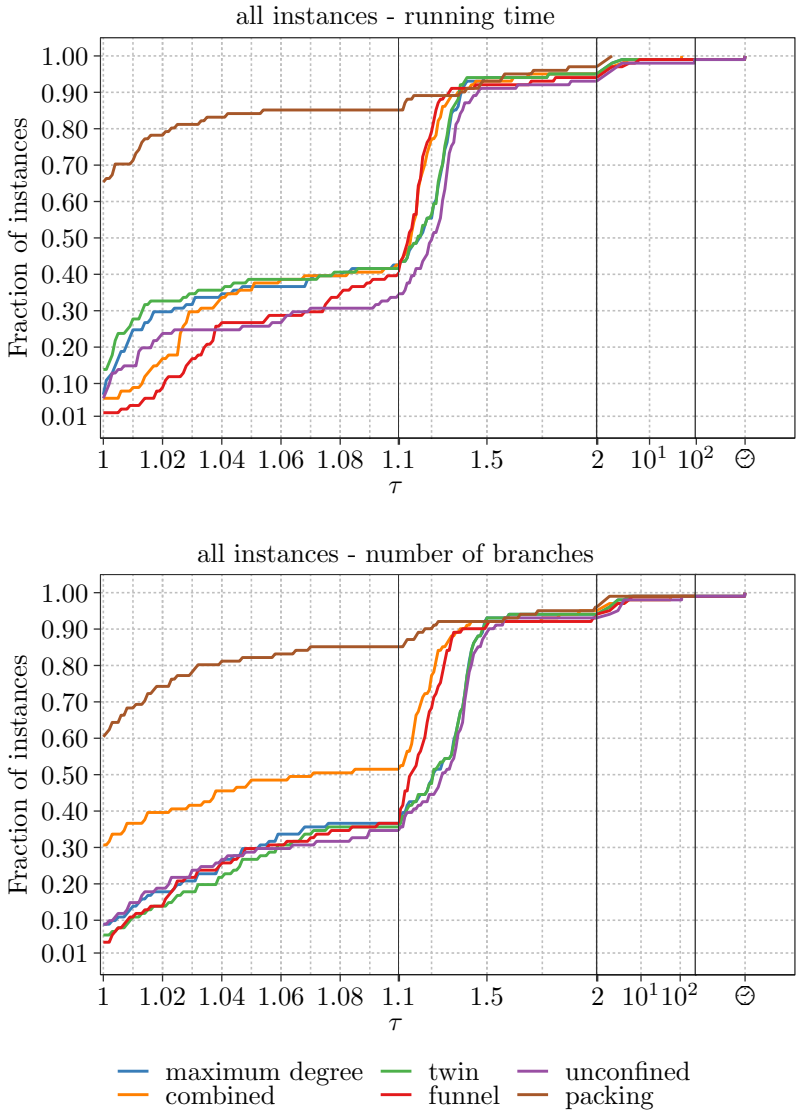


Figure 3.12: Performance plots for reduction-based branching strategies

3.5.7 Detailed Experimental Results

We now present detailed results of our experimental evaluation. Detailed tables show running times t (in seconds) and speedup s. Speedups are computed by dividing the running time of maximum degree branching by the running time of

the respective technique. Timeouts are assigned a running time of ten hours. Note, that this is the same as our time limit. We also present the aggregated speedup s_{total} computed by dividing the running time of both algorithms over all instances (omitting instances were both algorithms do not finish within our time limit). A value is highlighted in bold if it is the best one within a row.

Table 3.8: Detailed results for our decomposition-based strategies on the PACE instances.

Graph	max. deg.	articulation	edge cuts	nested dis.
PACE	t	t (s)	t (s)	t (s)
05	**1.97**	2.00 (0.98)	2.00 (0.99)	2.44 (0.81)
06	**0.85**	0.87 (0.98)	0.87 (0.98)	1.33 (0.64)
10	**2.24**	2.27 (0.99)	2.26 (0.99)	2.66 (0.84)
16	25 836.77	26 175.23 (0.99)	**25 763.30 (1.00)**	25 865.40 (1.00)
19	**3.17**	3.22 (0.98)	3.18 (0.99)	3.63 (0.87)
31	**74.37**	76.03 (0.98)	75.45 (0.99)	74.82 (0.99)
33	**1.01**	1.03 (0.98)	1.02 (0.99)	40.09 (0.03)
35	**7.64**	7.84 (0.97)	7.77 (0.98)	8.13 (0.94)
36	**1.84**	1.87 (0.98)	1.85 (0.99)	3.93 (0.47)
37	**10.27**	10.48 (0.98)	10.47 (0.98)	10.75 (0.96)
38	12.33	11.24 (1.10)	**3.25 (3.79)**	15.35 (0.80)
39	**93.79**	96.82 (0.97)	95.96 (0.98)	95.21 (0.99)
40	**4 690.64**	4 794.37 (0.98)	4 758.15 (0.99)	4 712.57 (1.00)
41	**48.56**	49.84 (0.97)	49.39 (0.98)	49.35 (0.98)
42	**37.32**	38.11 (0.98)	37.91 (0.98)	37.87 (0.99)
43	**175.11**	178.81 (0.98)	177.26 (0.99)	175.24 (1.00)
44	**92.90**	95.13 (0.98)	94.28 (0.99)	93.40 (0.99)
45	**25.41**	26.01 (0.98)	25.73 (0.99)	25.90 (0.98)
46	**109.55**	111.95 (0.98)	111.00 (0.99)	110.22 (0.99)
47	**58.47**	59.70 (0.98)	59.38 (0.98)	59.22 (0.99)
48	**25.28**	25.80 (0.98)	25.60 (0.99)	25.80 (0.98)
49	**17.80**	18.19 (0.98)	18.10 (0.98)	18.30 (0.97)
50	**48.87**	50.01 (0.98)	49.56 (0.99)	49.40 (0.99)
51	**56.70**	58.00 (0.98)	57.63 (0.98)	57.52 (0.99)
52	**22.16**	22.68 (0.98)	22.53 (0.98)	22.69 (0.98)
53	**59.88**	61.42 (0.97)	60.77 (0.99)	60.42 (0.99)
54	**32.08**	32.89 (0.98)	32.73 (0.98)	32.67 (0.98)
55	**6.83**	6.97 (0.98)	6.92 (0.99)	7.32 (0.93)
56	**97.00**	99.09 (0.98)	98.31 (0.99)	97.80 (0.99)
57	**66.01**	67.76 (0.97)	67.18 (0.98)	66.83 (0.99)
58	**48.12**	48.83 (0.99)	48.72 (0.99)	48.63 (0.99)
59	**13.30**	13.60 (0.98)	13.54 (0.98)	13.80 (0.96)
60	**79.56**	81.58 (0.98)	80.94 (0.98)	80.23 (0.99)
61	**21.91**	22.31 (0.98)	22.26 (0.98)	22.36 (0.98)
62	**66.22**	68.48 (0.97)	67.40 (0.98)	66.80 (0.99)
63	**69.06**	70.55 (0.98)	69.91 (0.99)	69.35 (1.00)
64	**29.58**	30.07 (0.98)	29.99 (0.99)	30.09 (0.98)
65	**36.84**	37.53 (0.98)	37.28 (0.99)	37.29 (0.99)
66	**8.06**	8.28 (0.97)	8.23 (0.98)	8.63 (0.93)
67	**122.74**	124.79 (0.98)	124.25 (0.99)	123.38 (0.99)
68	**8.79**	8.92 (0.99)	8.86 (0.99)	9.24 (0.95)
69	**43.11**	44.13 (0.98)	43.85 (0.98)	43.63 (0.99)
70	**11.79**	12.00 (0.98)	11.97 (0.99)	12.25 (0.96)
71	**36.20**	36.83 (0.98)	36.66 (0.99)	36.64 (0.99)
72	**46.44**	47.47 (0.98)	46.91 (0.99)	46.86 (0.99)
73	**43.02**	44.07 (0.98)	43.77 (0.98)	43.65 (0.99)
74	**7.06**	7.24 (0.97)	7.14 (0.99)	7.49 (0.94)
77	**13.30**	13.65 (0.97)	13.51 (0.98)	13.79 (0.96)
s_{total}	**1.00**	0.99	1.00	1.00

Table 3.9: Detailed results for our decomposition-based strategies on the DIMACS instances.

Graph	max. deg.	articulation	edge cuts	nested dis.
DIMACS	t	t (s)	t (s)	t (s)
C125.9	**0.98**	1.01 (0.97)	1.00 (0.98)	1.43 (0.69)
MANN_a27	**0.48**	0.49 (0.98)	0.49 (0.98)	0.98 (0.49)
MANN_a45	**73.80**	75.24 (0.98)	74.93 (0.98)	74.70 (0.99)
brock200_1	**137.34**	140.20 (0.98)	137.56 (1.00)	140.01 (0.98)
brock200_2	**4.59**	4.69 (0.98)	4.70 (0.98)	10.07 (0.46)
brock200_3	22.06	22.33 (0.99)	**21.92 (1.01)**	26.39 (0.84)
brock200_4	**28.34**	28.72 (0.99)	28.35 (1.00)	32.48 (0.87)
gen200_p0.9_44	**152.61**	156.30 (0.98)	154.50 (0.99)	153.49 (0.99)
gen200_p0.9_55	**131.24**	134.64 (0.97)	133.04 (0.99)	132.58 (0.99)
hamming8-4	**19.29**	19.65 (0.98)	19.49 (0.99)	25.38 (0.76)
johnson16-2-4	**39.87**	41.17 (0.97)	40.21 (0.99)	40.33 (0.99)
keller4	**2.62**	2.68 (0.98)	2.65 (0.99)	4.37 (0.60)
p_hat1000-1	**860.24**	868.71 (0.99)	870.04 (0.99)	906.24 (0.95)
p_hat1000-2	**33 035.45**	33 656.50 (0.98)	33 508.10 (0.99)	33 247.45 (0.99)
p_hat1500-1	**8 935.77**	9 015.15 (0.99)	9 015.74 (0.99)	8 994.28 (0.99)
p_hat300-1	**3.70**	3.79 (0.98)	3.82 (0.97)	23.94 (0.15)
p_hat300-2	**5.53**	5.66 (0.98)	5.63 (0.98)	21.76 (0.25)
p_hat300-3	189.58	191.06 (0.99)	**188.96 (1.00)**	196.89 (0.96)
p_hat500-1	**38.63**	39.26 (0.98)	39.41 (0.98)	59.29 (0.65)
p_hat500-2	**96.36**	97.82 (0.99)	97.58 (0.99)	107.29 (0.90)
p_hat500-3	**14 860.70**	14 895.15 (1.00)	14 979.65 (0.99)	14 909.35 (1.00)
p_hat700-1	163.30	**162.84 (1.00)**	163.17 (1.00)	177.34 (0.92)
p_hat700-2	**906.32**	917.87 (0.99)	914.96 (0.99)	917.50 (0.99)
san1000	**895.34**	902.64 (0.99)	903.38 (0.99)	920.28 (0.97)
san200_0.7_1	**10.85**	11.01 (0.98)	10.90 (1.00)	14.45 (0.75)
san200_0.7_2	0.33	0.34 (0.95)	**0.32 (1.01)**	2.34 (0.14)
san200_0.9_1	**13.93**	14.37 (0.97)	14.08 (0.99)	14.94 (0.93)
san200_0.9_2	**34.15**	34.77 (0.98)	34.35 (0.99)	34.90 (0.98)
san200_0.9_3	**1 069.00**	1 094.54 (0.98)	1 078.09 (0.99)	1 071.31 (1.00)
san400_0.5_1	**9.21**	9.35 (0.98)	9.36 (0.98)	16.76 (0.55)
san400_0.7_1	**1 125.52**	1 139.20 (0.99)	1 131.38 (0.99)	1 130.07 (1.00)
san400_0.7_2	3 062.38	**3 053.97 (1.00)**	3 083.59 (0.99)	3 073.66 (1.00)
san400_0.7_3	**4 411.82**	4 464.53 (0.99)	4 447.19 (0.99)	4 423.16 (1.00)
sanr200_0.7	**48.35**	49.51 (0.98)	48.71 (0.99)	52.13 (0.93)
sanr200_0.9	**679.25**	696.41 (0.98)	688.51 (0.99)	680.29 (1.00)
sanr400_0.5	**373.40**	374.20 (1.00)	374.26 (1.00)	380.08 (0.98)
sanr400_0.7	**29 766.80**	30 390.80 (0.98)	30 270.10 (0.98)	30 001.55 (0.99)
s_{total}	**1.00**	0.99	0.99	0.99

Table 3.10: Detailed results for our decomposition-based strategies on sparse networks.

Graph	max. deg.	articulation	edge cuts	nested dis.
Sparse net.	t	t (s)	t (s)	t (s)
as-skitter	**2 058.32**	2 100.57 (0.98)	2 071.06 (0.99)	2 068.46 (1.00)
baidu-relatedpages	**0.82**	0.88 (0.94)	0.86 (0.96)	7.22 (0.11)
bay	1.68	1.87 (0.90)	**1.31 (1.28)**	23.43 (0.07)
col	5 019.93	4 737.48 (1.06)	**3 872.65 (1.30)**	5 101.46 (0.98)
fla	25.33	**23.47 (1.08)**	24.58 (1.03)	329.42 (0.08)
hudong-internallink	**0.99**	1.55 (0.64)	1.46 (0.68)	1.99 (0.50)
in-2004	**5.22**	5.46 (0.96)	5.37 (0.97)	16.18 (0.32)
libimseti	**1 497.59**	1 507.54 (0.99)	1 503.49 (1.00)	1 704.53 (0.88)
musae-twitch_DE	**20 906.93**	21 470.00 (0.97)	20 987.30 (1.00)	20 949.83 (1.00)
musae-twitch_FR	**37.13**	37.81 (0.98)	37.32 (1.00)	41.55 (0.89)
petster-fs-dog	**6.82**	10.21 (0.67)	8.67 (0.79)	12.47 (0.55)
soc-LiveJournal1	**9.87**	11.50 (0.86)	11.06 (0.89)	23.91 (0.41)
web-BerkStan	**134.22**	360.88 (0.37)	138.84 (0.97)	207.92 (0.65)
web-Google	**0.61**	0.85 (0.71)	0.68 (0.89)	1.46 (0.41)
web-NotreDame	12.10	**9.07 (1.33)**	12.11 (1.00)	48.83 (0.25)
web-Stanford	>36 000	**8.38 (>4 294.84)**	27.41 (>1 313.18)	42.80 (>841.16)
s_{total}	1.00	2.17	**2.29**	2.15

Table 3.11: Detailed results for our reduction-based strategies on the PACE instances.

Graph	max. deg.	Twin	Funnel	Unconfined	Packing	Combined
PACE	t	t (s)	t (s)	t (s)	t (s)	t (s)
05	1.97	1.96 (1.01)	1.99 (0.99)	2.04 (0.97)	**1.66 (1.19)**	2.11 (0.93)
06	0.85	0.85 (1.00)	0.74 (1.15)	0.92 (0.92)	**0.67 (1.27)**	0.81 (1.05)
10	2.24	2.23 (1.01)	2.23 (1.00)	2.32 (0.97)	**1.88 (1.19)**	2.06 (1.09)
16	25 836.77	25 856.57 (1.00)	22 446.13 (1.15)	34 642.13 (0.75)	**18 511.88 (1.40)**	22 590.78 (1.14)
19	3.17	3.14 (1.01)	2.90 (1.09)	3.25 (0.98)	**2.60 (1.22)**	3.04 (1.04)
31	74.37	74.31 (1.00)	58.14 (1.28)	73.23 (1.02)	55.99 (1.33)	**54.11 (1.37)**
33	1.01	**1.01 (1.00)**	1.15 (0.88)	1.14 (0.89)	1.02 (0.99)	1.29 (0.79)
35	7.64	7.63 (1.00)	7.37 (1.04)	7.90 (0.97)	**6.54 (1.17)**	7.75 (0.99)
36	**1.84**	1.86 (0.99)	11.44 (0.16)	162.22 (0.01)	1.90 (0.97)	75.52 (0.02)
37	10.27	10.31 (1.00)	10.27 (1.00)	10.63 (0.97)	**8.21 (1.25)**	10.90 (0.94)
38	12.33	12.36 (1.00)	11.08 (1.11)	11.40 (1.08)	11.44 (1.08)	**10.05 (1.23)**
39	93.79	93.99 (1.00)	**32.43 (2.89)**	127.32 (0.74)	93.99 (1.00)	98.25 (0.95)
40	4 690.64	4 689.28 (1.00)	4 285.37 (1.09)	4 530.07 (1.04)	4 176.59 (1.12)	**4 131.79 (1.14)**
41	48.56	48.42 (1.00)	42.00 (1.16)	48.66 (1.00)	**36.87 (1.32)**	38.74 (1.25)
42	37.32	37.19 (1.00)	35.69 (1.05)	37.60 (0.99)	**28.55 (1.31)**	36.07 (1.03)
43	175.11	174.63 (1.00)	158.08 (1.11)	172.91 (1.01)	**130.75 (1.34)**	154.96 (1.13)
44	92.90	92.97 (1.00)	82.64 (1.12)	94.37 (0.98)	**69.68 (1.33)**	90.20 (1.03)
45	25.41	25.37 (1.00)	25.29 (1.01)	26.20 (0.97)	**19.83 (1.28)**	26.38 (0.96)
46	109.55	109.47 (1.00)	92.61 (1.18)	108.01 (1.01)	**79.76 (1.37)**	82.72 (1.32)
47	58.47	58.18 (1.00)	53.01 (1.10)	59.16 (0.99)	**42.32 (1.38)**	52.28 (1.12)
48	25.28	25.21 (1.00)	22.65 (1.12)	25.72 (0.98)	**18.56 (1.36)**	22.93 (1.10)
49	17.80	17.76 (1.00)	16.43 (1.08)	19.02 (0.94)	**12.97 (1.37)**	16.18 (1.10)
50	48.87	48.90 (1.00)	46.07 (1.06)	49.75 (0.98)	**37.70 (1.30)**	47.09 (1.04)
51	56.70	56.58 (1.00)	51.45 (1.10)	57.63 (0.98)	**43.45 (1.31)**	50.32 (1.13)
52	22.16	22.12 (1.00)	20.56 (1.08)	22.99 (0.96)	**15.78 (1.40)**	20.82 (1.06)
53	59.88	59.88 (1.00)	54.78 (1.09)	60.43 (0.99)	**46.87 (1.28)**	55.74 (1.07)
54	32.08	32.02 (1.00)	29.29 (1.10)	32.89 (0.98)	**26.55 (1.21)**	27.76 (1.16)
55	6.83	6.80 (1.00)	6.50 (1.05)	6.99 (0.98)	**5.23 (1.31)**	6.35 (1.08)
56	97.00	96.45 (1.01)	88.78 (1.09)	98.09 (0.99)	**70.18 (1.38)**	81.46 (1.19)
57	66.01	65.97 (1.00)	57.60 (1.15)	65.90 (1.00)	**49.95 (1.32)**	52.45 (1.26)
58	48.12	47.74 (1.01)	45.82 (1.05)	48.56 (0.99)	**35.94 (1.34)**	46.62 (1.03)
59	13.30	13.30 (1.00)	12.73 (1.04)	13.72 (0.97)	**10.61 (1.25)**	12.30 (1.08)
60	79.56	79.36 (1.00)	71.73 (1.11)	80.70 (0.99)	**59.65 (1.33)**	71.85 (1.11)
61	21.91	21.91 (1.00)	20.47 (1.07)	22.28 (0.98)	**17.50 (1.25)**	21.06 (1.04)
62	66.22	66.18 (1.00)	59.16 (1.12)	67.83 (0.98)	**49.87 (1.33)**	59.64 (1.11)
63	69.06	68.81 (1.00)	61.23 (1.13)	70.81 (0.98)	**53.40 (1.29)**	58.65 (1.18)
64	29.58	29.38 (1.01)	26.96 (1.10)	29.46 (1.00)	**22.35 (1.32)**	26.78 (1.10)
65	36.84	36.72 (1.00)	33.42 (1.10)	37.93 (0.97)	**28.23 (1.30)**	31.17 (1.18)
66	8.06	8.06 (1.00)	7.47 (1.08)	8.21 (0.98)	**6.21 (1.30)**	7.97 (1.01)
67	122.74	122.34 (1.00)	113.33 (1.08)	123.58 (0.99)	**95.55 (1.28)**	112.43 (1.09)
68	8.79	8.75 (1.00)	8.92 (0.99)	8.94 (0.98)	**6.69 (1.31)**	8.57 (1.03)
69	43.11	43.11 (1.00)	38.46 (1.12)	44.18 (0.98)	**33.88 (1.27)**	39.86 (1.08)
70	11.79	11.73 (1.00)	10.09 (1.17)	12.22 (0.96)	**9.71 (1.21)**	9.76 (1.21)
71	36.20	35.91 (1.01)	32.22 (1.12)	35.37 (1.02)	**27.23 (1.33)**	33.39 (1.08)
72	46.44	46.18 (1.01)	41.66 (1.11)	46.68 (0.99)	**36.28 (1.28)**	41.86 (1.11)
73	43.02	43.00 (1.00)	40.38 (1.07)	43.77 (0.98)	**31.91 (1.35)**	43.51 (0.99)
74	7.06	7.06 (1.00)	6.67 (1.06)	7.86 (0.90)	**5.48 (1.29)**	6.96 (1.01)
77	13.30	13.25 (1.00)	12.74 (1.04)	13.80 (0.96)	**10.61 (1.25)**	12.31 (1.08)
s_{total}	1.00	1.00	1.14	0.79	**1.34**	1.14

Table 3.12: Detailed results for our reduction-based strategies on the DIMACS instances.

Graph	max. deg.	Twin	Funnel	Unconfined	Packing	Combined
DIMACS	t	t (s)	t (s)	t (s)	t (s)	t (s)
C125.9	0.98	0.98 (1.00)	0.92 (1.07)	0.98 (1.00)	**0.85 (1.15)**	0.91 (1.08)
MANN_a27	0.48	0.48 (1.00)	0.57 (0.85)	0.52 (0.92)	**0.48 (1.01)**	0.59 (0.82)
MANN_a45	73.80	73.76 (1.00)	83.81 (0.88)	78.58 (0.94)	**71.86 (1.03)**	85.47 (0.86)
brock200_1	137.34	136.98 (1.00)	140.15 (0.98)	137.32 (1.00)	**135.14 (1.02)**	138.64 (0.99)
brock200_2	4.59	4.60 (1.00)	4.71 (0.98)	4.59 (1.00)	**4.58 (1.00)**	4.70 (0.98)
brock200_3	22.06	21.78 (1.01)	22.38 (0.99)	21.85 (1.01)	**21.76 (1.01)**	22.46 (0.98)
brock200_4	28.34	**28.15 (1.01)**	29.09 (0.97)	28.16 (1.01)	28.25 (1.00)	29.24 (0.97)
gen200_p0.9_44	152.61	152.40 (1.00)	136.94 (1.11)	169.47 (0.90)	**132.81 (1.15)**	149.63 (1.02)
gen200_p0.9_55	131.24	131.20 (1.00)	125.61 (1.04)	127.51 (1.03)	102.10 (1.29)	**50.64 (2.59)**
hamming8-4	19.29	19.30 (1.00)	19.78 (0.98)	**19.12 (1.01)**	19.35 (1.00)	19.67 (0.98)
johnson16-2-4	39.87	39.79 (1.00)	41.63 (0.96)	41.40 (0.96)	**38.70 (1.03)**	43.09 (0.93)
keller4	2.62	2.62 (1.00)	2.68 (0.98)	2.63 (1.00)	**2.58 (1.02)**	2.65 (0.99)
p_hat1000-1	860.24	**859.74 (1.00)**	870.92 (0.99)	873.91 (0.98)	862.77 (1.00)	871.60 (0.99)
p_hat1000-2	33 035.45	33 314.15 (0.99)	32 999.15 (1.00)	32 812.80 (1.01)	**30 913.22 (1.07)**	31 202.52 (1.06)
p_hat1500-1	8 935.77	**8 935.50 (1.00)**	9 009.69 (0.99)	8 954.18 (1.00)	8 958.19 (1.00)	9 046.97 (0.99)
p_hat300-1	3.70	3.69 (1.00)	3.78 (0.98)	3.69 (1.00)	**3.68 (1.00)**	3.78 (0.98)
p_hat300-2	5.53	5.53 (1.00)	5.68 (0.97)	5.54 (1.00)	**5.48 (1.01)**	5.63 (0.98)
p_hat300-3	189.58	187.77 (1.01)	189.16 (1.00)	185.68 (1.02)	**175.01 (1.08)**	179.53 (1.06)
p_hat500-1	38.63	38.70 (1.00)	39.36 (0.98)	39.03 (0.99)	**38.61 (1.00)**	39.34 (0.98)
p_hat500-2	96.36	96.39 (1.00)	97.87 (0.98)	96.21 (1.00)	**95.08 (1.01)**	96.96 (0.99)
p_hat500-3	14 860.70	14 887.15 (1.00)	14 624.90 (1.02)	14 765.90 (1.01)	**13 429.92 (1.11)**	13 712.38 (1.08)
p_hat700-1	163.30	**160.75 (1.02)**	163.63 (1.00)	160.81 (1.02)	163.24 (1.00)	163.31 (1.00)
p_hat700-2	906.32	908.46 (1.00)	914.56 (0.99)	906.78 (1.00)	**866.08 (1.05)**	879.99 (1.03)
san1000	**895.34**	898.16 (1.00)	906.21 (0.99)	901.40 (0.99)	913.29 (0.98)	932.29 (0.96)
san200_0.7_1	10.85	**10.78 (1.01)**	11.01 (0.99)	10.91 (0.99)	10.93 (0.99)	11.06 (0.98)
san200_0.7_2	0.33	0.32 (1.04)	0.33 (0.98)	**0.31 (1.07)**	0.32 (1.01)	0.33 (0.99)
san200_0.9_1	13.93	13.90 (1.00)	13.35 (1.04)	**4.94 (2.82)**	12.03 (1.16)	12.13 (1.15)
san200_0.9_2	34.15	33.87 (1.01)	21.46 (1.59)	12.32 (2.77)	15.80 (2.16)	**10.01 (3.41)**
san200_0.9_3	1 069.00	1 068.17 (1.00)	1 016.33 (1.05)	639.01 (1.67)	843.40 (1.27)	**600.71 (1.78)**
san400_0.5_1	9.21	9.21 (1.00)	9.37 (0.98)	**9.13 (1.01)**	9.24 (1.00)	9.37 (0.98)
san400_0.7_1	1 125.52	**1 121.99 (1.00)**	1 146.32 (0.98)	1 125.12 (1.00)	1 132.10 (0.99)	1 151.14 (0.98)
san400_0.7_2	3 062.38	3 063.23 (1.00)	3 066.62 (1.00)	3 463.29 (0.88)	**3 048.94 (1.00)**	3 489.72 (0.88)
san400_0.7_3	4 411.82	4 405.26 (1.00)	4 487.18 (0.98)	**4 398.18 (1.00)**	4 497.81 (0.98)	4 521.80 (0.98)
sanr200_0.7	48.35	**48.34 (1.00)**	50.09 (0.97)	48.41 (1.00)	48.49 (1.00)	50.25 (0.96)
sanr200_0.9	679.25	679.65 (1.00)	633.59 (1.07)	664.95 (1.02)	**531.48 (1.28)**	567.49 (1.20)
sanr400_0.5	373.40	**370.59 (1.01)**	376.93 (0.99)	377.71 (0.99)	370.72 (1.01)	376.10 (0.99)
sanr400_0.7	29 766.80	29 838.40 (1.00)	30 466.35 (0.98)	29 844.65 (1.00)	**29 473.60 (1.01)**	30 242.80 (0.98)
s_{total}	1.00	1.00	0.99	1.00	**1.04**	1.03

Table 3.13: Detailed results for our reduction-based strategies on sparse networks.

Graph	max. deg.	Twin	Funnel	Unconfined	Packing	Combined
Sparse net.	t	t (s)	t (s)	t (s)	t (s)	t (s)
as-skitter	2 058.32	2 054.41 (1.00)	1 849.79 (1.11)	1 977.94 (1.04)	**1 681.87 (1.22)**	1 704.73 (1.21)
baidu-relatedpages	0.82	**0.80 (1.02)**	0.84 (0.97)	0.85 (0.97)	0.83 (0.98)	0.93 (0.88)
bay	**1.68**	1.68 (1.00)	8.22 (0.20)	4.71 (0.36)	1.89 (0.89)	8.38 (0.20)
col	**5 019.93**	5 752.08 (0.87)	5 416.72 (0.93)	8 187.80 (0.61)	9 370.05 (0.54)	5 924.10 (0.85)
fla	**25.33**	25.41 (1.00)	45.62 (0.56)	76.60 (0.33)	34.78 (0.73)	42.75 (0.59)
hudong-internallink	**0.99**	1.31 (0.76)	1.27 (0.78)	1.21 (0.82)	1.55 (0.64)	1.12 (0.88)
in-2004	5.22	**4.88 (1.07)**	5.25 (0.99)	10.85 (0.48)	5.50 (0.95)	10.73 (0.49)
libimseti	1 497.59	1 452.17 (1.03)	1 620.09 (0.92)	**1 440.71 (1.04)**	1 476.25 (1.01)	1 706.07 (0.88)
musae-twitch_DE	20 906.93	20 996.87 (1.00)	21 190.67 (0.99)	22 650.53 (0.92)	**19 345.03 (1.08)**	23 006.50 (0.91)
musae-twitch_FR	37.13	37.04 (1.00)	38.58 (0.96)	41.15 (0.90)	**35.60 (1.04)**	42.46 (0.87)
petster-fs-dog	6.82	**6.62 (1.03)**	8.16 (0.84)	8.66 (0.79)	9.68 (0.70)	9.20 (0.74)
soc-LiveJournal1	9.87	**6.64 (1.49)**	9.57 (1.03)	9.49 (1.04)	11.33 (0.87)	10.69 (0.92)
web-BerkStan	134.22	135.47 (0.99)	**122.30 (1.10)**	146.94 (0.91)	123.60 (1.09)	174.07 (0.77)
web-Google	0.61	**0.53 (1.15)**	0.69 (0.87)	0.68 (0.89)	0.78 (0.78)	0.68 (0.89)
web-NotreDame	**12.10**	12.63 (0.96)	15.23 (0.79)	12.38 (0.98)	14.09 (0.86)	17.52 (0.69)
web-Stanford	>36 000	>36 000	>36 000	>36 000	**17 886.35 (>2.01)**	17 989.97 (>2.00)
s_{total}	1.00	0.97	0.98	0.86	**1.31**	1.30

Chapter 4

4 Maximum Cuts

Abstract. *For the fundamental* Max Cut *problem, kernelization algorithms are theoretically highly efficient for various parameterizations. However, the efficacy of these reduction rules in practice—to aid solving highly challenging benchmark instances to optimality—remains entirely unexplored.*

We engineer a new suite of efficient data reduction rules that subsume most of the previously published rules, and demonstrate their significant impact on benchmark data sets, including synthetic instances, and data sets from the VLSI and image segmentation application domains. Our experiments reveal that current state-of-the-art solvers can be sped up by up to multiple orders of magnitude *when combined with our data reduction rules. On social and biological networks in particular, kernelization enables us 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.*

References and Attribution. This chapter is based on the conference paper [Fer+20]. Together with Sebastian Lamm, the author of this thesis is a main author of this paper with editing done by Damir Ferizovic, Matthias Mnich, Christian Schulz and Darren Strash. The author made major contributions to the theoretical proofs and analyses of the reduction rules developed as well as the time stamping approach and the experiment design. The implementation was done by Damir Ferizovic and the experiments were done by Sebastian Lamm and Damir Ferizovic. Large parts of this chapter were copied verbatim from the conference paper or the corresponding technical report [Fer+19].

4.1 Introduction

The (unweighted) Max Cut problem is to partition the vertex set of a given graph $G = (V, E)$ into two sets $S \subseteq V$ and $V \setminus S$ so as to maximize the total number of edges between those two sets. Such a partition is called a *maximum cut*. Computing a maximum cut of a graph is a well-known problem in the area of computer science; it is one of Karp's 21 NP-complete problems [Kar72]. While signed and weighted variants are often considered throughout the literature [Bar82; Bar96; Bar+88; Chi+07; dSHK13; Har59; HLW02], the simpler (unweighted) case still

presents a significant challenge for researchers, and solving it quickly is of paramount importance to all variants. Max Cut variants have many applications, including social network modeling [Har59], statistical physics [Bar82], portfolio risk analysis [HLW02], VLSI design [Bar+88; Chi+07], network design [Bar96], and image segmentation [dSHK13].

Theoretical approaches to solving Max Cut primarily focus on producing efficient parameterized algorithms through *data reduction rules*, which reduce the input size in polynomial time while maintaining the ability to compute an optimal solution to the original input. If the resulting (irreducible) graph has size bounded by a function of a given parameter, then it is called a *kernel*. Recent works focus on parameters measuring the distance k between the maximum cut size of the input graph and a lower bound ℓ guaranteed for all graphs. The algorithm then must decide if the input graph admits a cut of size $\ell + k$ for a given integer $k \in \mathbb{N}$. Two such lower bounds are the Edwards-Erdős bound [Edw73; Edw75] and the spanning tree bound. Crowston et al. [CJM15] were the first to show that unweighted Max Cut is fixed-parameter tractable when parameterized by distance k above the Edwards-Erdős bound. Moreover, they show the problem admits a polynomial-size kernel with $O(k^5)$ vertices. Their result was extended to the more general Signed Max Cut problem, and the kernel size was decreased to $O(k^3)$ vertices [Cro+13]. Finally, Etscheid and Mnich [EM18] improved the kernel size to an optimal $O(k)$ vertices even for signed graphs, and showed how to compute it in linear time $O(k \cdot (|V|+|E|))$.

Many practical approaches exist to compute a maximum cut or (alternatively) a large cut. Two state-of-the-art exact solvers are Biq Mac (a solver for *bi*nary *q*uadratic and *Ma*x-*C*ut problems) by Rendl et al. [RRW10], and LocalSolver [Ben+11; Gar+14], a powerful generic local search solver that also verifies optimality of a cut. Many heuristic (inexact) solvers are also available, including those using unconstrained binary quadratic optimization [Wan+13], local search [BH13], tabu search [Koc+13], and simulated annealing [AO09].

Curiously, data reduction, which has shown promise at preprocessing large instances of other fundamental NP-hard problems [Abu+07; HSS18; Lam+17], is currently not used in implementations of Max Cut solvers. To the best of our knowledge, no research has been done on the efficiency of data reduction for Max Cut, in particular with the goal of achieving small kernels *in practice*.

Our Results. We introduce new data reduction rules for the Max Cut problem, and show that nearly all previous reduction rules for the Max Cut problem can be encompassed by only four reduction rules. Furthermore, we engineer efficient implementations of these reduction rules and show through extensive experiments that kernelization achieves a substantial reduction on sparse graphs. Our experiments reveal that current state-of-the-art solvers can be sped up by up to *multiple orders of magnitude* when combined with our data reduction rules. We achieve speedups on all instances tested. On social and biological networks in particular, kernelization enables us 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 with our kernelization.

4.2 Preliminaries

For general definitions and notations for graphs see Section 2.1. Here we give some more definitions specific to maximum cuts and the algorithms used in this chapter. Note that in this chapter, we consider *undirected*, *weighted* graphs.

The set of edges between the vertices of different vertex sets $S_1, S_2 \subseteq V$ is written as $E(S_1, S_2) := E \cap (S_1 \times S_2)$. For vertex sets $S \subseteq V(G)$, the set of *external vertices* is $C_{\text{ext}}(S) = \{v \in S \mid \exists w \in V(G) \setminus S, \{v, w\} \in E(G)\}$, which is the set of vertices in S that have some neighbor in G outside S. In similar fashion, $C_{\text{int}}(S) = S \setminus C_{\text{ext}}(S)$ defines the set of *internal vertices*.

A *clique* is a complete subgraph, and a *near-clique* is a clique minus a single edge. A *clique tree* is a connected graph whose biconnected components are cliques, and a *clique forest* is a graph whose connected components are clique trees. In such graphs, we use the term *block* to refer to a biconnected component.

The Max Cut problem is to find a vertex set $S \subseteq V$, such that $|E(S, V \setminus S)|$ is maximized. We denote the cardinality of a maximum cut by $\beta(G)$. At times, we may need to reason about a maximum cut given a fixed partitioning of a subset of G's vertices. A partition of vertices $V' \subseteq V(G)$ is given as a 2-coloring $\delta : V' \to \{0, 1\}$. We let $\beta_\delta(G)$ denote the size of a maximum cut of G, given that $V' \subseteq V(G)$ is partitioned according to δ. The Weighted Max Cut problem is to find a vertex set S of a given graph G with additive weight function ω such that $\omega(E(S, V(G) \setminus S))$ is maximum. The weight of a maximum cut is then given by $\beta(G, \omega) := \omega(E(S, V \setminus S))$. We denote instances of the Max Cut decision problem as $(G, k)_{\text{MC}}$, where G is a graph and $k \in \mathbb{N}_0$, If the weight of a maximum cut in G is at least k, then $(G, k)_{\text{MC}}$ is a "yes"-instance; otherwise, it is a "no"-instance.

We address two more variations of Max Cut in this chapter. The Vertex-Weighted Max Cut problem takes as input a graph G and two vertex weight functions $\omega_0, \omega_1 : V(G) \to \mathbb{R}$; the objective is to compute a bipartition $V_0 \cup V_1 = V(G)$ that maximizes $|E(V_0, V_1)| + \sum_{v \in V_0} w_0(v) + \sum_{v \in V_1} w_1(v)$. The Signed Max Cut problem takes as input a graph G together with an edge labeling $l : E(G) \to \{\text{"+"}, \text{"-"}\}$; the goal is to find an $S \subseteq V(G)$ which maximizes the number negative ("-") edges *between* S and $V(G) \setminus S$ plus the number of positive ("+") edges *inside* of S and $V(G) \setminus S$. Formally, maximize $\beta(G, l) := |E_l^-(S, V(G) \setminus S)| + |E^+(G[S], l) \cup E^+(G[V(G) \setminus S], l)|$, where $E_l^c(S, V(G) \setminus S) := \{e \in E(S, V(G) \setminus S) \mid l(e) = c\}$ and $E^c(G, l) := \{e \in E(G) \mid l(e) = c\}$ for $c \in \{\text{"-"}, \text{"+"}\}$. Similarly, for the neighborhood of a vertex (set), we use the notations $N_l^c(v) := \{w \in V(G) \mid \{v, w\} \in E^c(l)\}$ and $N_l^c(X) := \bigcup_{v \in X} N_l^c(v) \setminus X$. We call a triangle *positive* if its number of "-"-edges is even. Any Max Cut instance can be transformed into a Signed Max Cut instance by labeling all edges with "-".

Let Σ^* denote the set of input instances for a decision problem. A parameterized problem $\Pi \subseteq \Sigma^* \times \mathbb{N}$ is *fixed-parameter tractable* if there is an algorithm $\mathcal{A}$ (called a *fixed-parameter algorithm*) that decides membership in Π for any input pair $(x,k) \in \Sigma^* \times \mathbb{N}$ in time $f(k) \cdot |(x,k)|^{O(1)}$ for some computable function $f : \mathbb{N} \to \mathbb{N}$.

A *data reduction rule* (often shortened to *reduction rule*) for a parameterized problem Π is a function $\phi : \Sigma^* \times \mathbb{N} \to \Sigma^* \times \mathbb{N}$ that maps an instance (x,k) of Π to an equivalent instance (x',k') of Π such that ϕ is computable in time polynomial in $|x|$ and k. We call two instances of Π *equivalent* if either both or none belong to Π. Observe that for two equivalent "yes"-instances $(G, \beta(G))$ and $(G', \beta(G'))$, the relationship $\beta(G) = \beta(G') + k$ holds for some $k \in \mathbb{Z}$.

4.2.1 Related Work

Several studies have been made in the direction of providing fixed-parameter algorithms for the Max Cut problem [Cro+13; CJM15; EM18; MSZ18]. Among these, a fair amount of kernelization rules have been introduced with the goal of effectively reducing Max Cut instances [Cro+13; CJM15; EM18; MSZ18; Pri05; Far+17]. Those reductions typically have some constraints on the subgraphs, like being clique forests or so-called clique-cycle forest. Later, we propose a new set of reductions that does not need this property and cover most of the known reductions [CJM15; EM18; MSZ18; Far+17]. There are other reductions rules that are fairly simplistic and focus on very narrow cases [Pri05]. We now explain the Edwards-Erdős bound and the spanning tree bound.

Edwards-Erdős Bound. For a connected graph, the Edwards-Erdős bound [Edw73; Edw75] is defined as $EE(G) = \frac{|E(G)|}{2} + \frac{|V(G)|-1}{4}$. A linear-time algorithm that computes a cut satisfying the Edwards-Erdős bound for any given graph is provided by Van Ngoc and Tuza [VT93]. The Max Cut Above Edwards-Erdős (Max Cut AEE) problem asks for a graph G and integer $k \in \mathbb{N}_0$ if G admits a cut of size $EE(G) + k$. All kernelization rules for Max Cut AEE require a set $S \subseteq V$ set such that $G - S$ is a clique forest. Etscheid and Mnich [EM18] propose an algorithm that computes such a set S of at most $3k$ vertices in time $O(k \cdot (|V| + |E|))$.

Spanning Tree Bound. Another approach is based on utilizing the spanning forest of a graph [MSZ18]. For a given $k \in \mathbb{N}_0$, a Max Cut of size $|V| - 1 + k$ is searched for. This decision problem is denoted as Max Cut AST (Max Cut Above Spanning Tree). For sparse graphs, this bound is larger than the Edwards-Erdős bound. The reductions for the problem require a set $S \subset V(G)$ such that $G - S$ is a so-called clique-cycle forest.

4.3 New Data Reduction Rules

We now introduce our new data reduction rules and prove their correctness. The main feature of our new rules is that they do not depend on the computation of a clique-forest to determine if they can be applied. Furthermore, our new

Table 4.1: Reduction rules from previous work subsumed by our new rules. A ✓ in row a and column b means that the rule from row a subsumes the rule from column b. If there are multiple ✓s in a column (say, rows a and b in column c), then rules a and b combined subsume rule c.

Source	[Far+17]	[CJM15]			[Cro+13]		[EM18]	[MSZ18]							
Rule	A	5	6	7	8	9	9	6	7	8	9	10	11	12	13
$7_{w=1}$			✓		✓										
1	✓	✓		✓		✓	✓	✓	✓	✓	✓			✓	✓
2								✓		✓	✓	✓	✓		
5			✓												

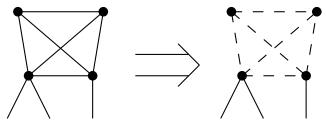


Figure 4.1: Reduction Rule 1. The shown vertices induce a clique of size four with two external vertices. All edges within the clique are removed from the graph. Since the internal vertices have no incident edges after this, they can also be removed from the graph.

rules subsume almost all rules from previous works [Cro+13; CJM15; EM18; MSZ18; Far+17] with the exception of Reduction Rules 10 and 11 by Crowston et al. [Cro+13]. Proofs for this were done by Damir Ferizovic in his Master Thesis [Fer19]. For an overview of how rules are subsumed, consult Table 4.1.

Reduction Rule 1. Let $G = (V, E)$ be a graph and let $S \subseteq V$ induce a clique in G. If $|C_{\text{ext}(G)}(S)| \leq \lceil |S|/2 \rceil$, then $\beta(G) = \beta(G') + \beta(K_{|S|})$ for $G' = (V \setminus C_{\text{int}(G)}(S), E \setminus E(G[S]))$, where K_n is the complete graph of n vertices, i.e., a clique of size n. Figure 4.1 shows an example of an application of Reduction Rule 1.

Proof. Note that any partition of the clique $G[S]$ into two vertex sets of size $\lceil |S|/2 \rceil$ and $\lfloor |S|/2 \rfloor$ is a maximum cut of $G[S]$. Suppose we fix the partitions of the at most $\lceil |S|/2 \rceil$ external vertices of S. Then the at least $\lfloor |S|/2 \rfloor$ internal vertices can be assigned to the partitions so they each contain $\lceil |S|/2 \rceil$ and $\lfloor |S|/2 \rfloor$ vertices. Thus, regardless of how $C_{\text{ext}(G)}(S)$ is partitioned, the size of a maximum cut of $G[S]$ remains the same. □

We can exhaustively apply Reduction Rule 1 in $O(|V| \cdot \Delta^2)$ time by scanning over all vertices in the graph. When scanning vertex v, we check whether $N[v]$ induces a clique. This finds all cliques with at least one internal vertex. Checking whether Reduction Rule 1 is applicable is then straightforward by counting the number of vertices with degree higher than the size of the clique.

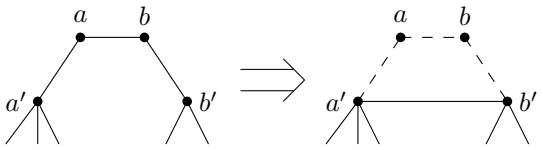


Figure 4.2: Reduction Rule 2. Vertices a', a, b, and b' induce a 3-path with $N(a) = \{a', b\}$ and $N(b) = \{a, b'\}$. Th edges of the path are removed and a new edge between a' and b' is inserted. Since a and b now have degree zero, they can also be removed from the graph.

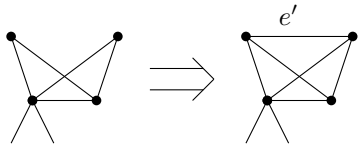


Figure 4.3: Reduction Rule 3. The vertices shown induce a near-clique with three internal vertices and a missing edge between two internal vertices. The missing edge e' is added to the graph.

Reduction Rule 2. Let (a', a, b, b') be an induced 3-path in a graph G with $N(a) = \{a', b\}$ and $N(b) = \{a, b'\}$. Construct G' from G by adding a new edge $\{a', b'\}$ and removing the vertices a and b. Then $\beta(G) = \beta(G') + 2$. Figure 4.2 shows and example of this reduction rule.

Proof. Let $S = \{a', a, b, b'\}$ and let $\delta : V \to \{0, 1\}$ be an assignment of vertices to the partitions of a cut in G. We distinguish two cases:

- Case $\delta(a') = \delta(b')$: If $\delta(a) = \delta(b) = \delta(a')$, then no edges of $G[S]$ are cut. Notice that this cut is not maximum since moving b between partitions increases the cut size by two. If $\delta(a) \neq \delta(a')$ or $\delta(b) \neq \delta(b')$, then exactly two edges in $G[S]$ are cut.
- Case $\delta(a') \neq \delta(b')$: By choosing $\delta(a) = \delta(b')$ and $\delta(b) = \delta(a')$, all three edges in $G[S]$ are cut. In G', the edge between a' and b' is cut, so $\beta(G) = \beta(G') + 2$.

□

Reduction Rule 3. Let G be a graph and let $S \subseteq V(G)$ induce a near-clique in G where the missing edge e' connects two internal vertices. Let G' be the graph obtained from G by adding e' so that S induces a clique in G'. If $|S|$ is odd or $|C_{\text{int}(G)}(S)| > 2$, then $\beta(G) = \beta(G')$. See Figure 4.3 for an example.

Proof. Let $e' = (u, v)$ be the edge added to the graph and δ any 2-coloring of $C_{\text{ext}(G)}(S)$. We show that a maximum cut of G' exists such that u and v are in the same partition. As G has one less edge than G', this means that $\beta_\delta(G[S]) = \beta_\delta(G'[S])$, which implies that $\beta(G) = \beta(G')$.

Define $V_c = \{x \in C_{\text{ext}(G')}(S) \mid \delta(x) = c\}$ for $c \in \{0, 1\}$. Without loss of generality, assume $|V_0| \geq |V_1|$. Note that, given the partition for $C_{\text{ext}(G')}(S)$, maximizing the cut of S means minimizing $||V_0| - |V_1||$. We distinguish three cases:

- $|V_0| - |V_1| \geq 2$: By adding u and v to V_0, $||V_0| - |V_1||$ decreases. The rest of the internal vertices have to be distributed among V_0 and V_1 such that $||V_0| - |V_1||$ is minimized.

- $|V_0| - |V_1| = 1$: By adding u and v to V_0, $||V_0| - |V_1||$ stays 1. If $|S|$ is odd, then 1 is the minimal value possible and $|C_{\text{int}(G)}(S)|$ is even. So the remaining internal vertices can be distributed evenly between V_0 and V_1. If S is even, then an odd number of internal vertices are left (and at least one by the definition of the rule) which can be distributed to balance V_0 and V_1.

- $|V_0| = |V_1|$: By adding u and v to V_0, $||V_0| - |V_1||$ becomes 2. If $|S|$ is odd, then an odd number of internal vertices is left to assign to such that $||V_0| - |V_1||$ becomes 1. If $|S|$ is even then there is an even number of internal vertices left which can be distributed to balance V_0 and V_1.

Since some cliques are irreducible by currently known rules, it may be beneficial to also apply Reduction Rule 3 'in reverse'. Although this 'reverse' reduction neither reduces the vertex set nor (as our experiments suggest) lead to applications of other rules, it can undo unfruitful additions of edges made by Reduction Rule 3 and may remove other edges from the graph.

Reduction Rule 4. Let G be a graph and let $S \subseteq V(G)$ induce a clique in G. If $|S|$ is odd or $|C_{\text{int}(G)}(S)| > 2$, an edge between two vertices of $C_{\text{int}(G)}(S)$ is removable. That is, $\beta(G) = \beta(G')$ for $G' = (V, E \setminus \{e\})$, $e \in E(G[C_{\text{int}(G)}(S)])$.

Proof. Follows from the correctness of Reduction Rule 3. □

The following reduction rule is closely related to the upcoming generalization of Reduction Rule 8 by Crowston et al. [Cro+13]. It is able to further reduce the case where $|X| = |N(X)|$ for a clique X of G. In comparison, the generalization of Reduction Rule 8 from [Cro+13] is able to handle the case $|X| > |N(X)|$. Due to the degree by which these rules are similar, they are also merged together in our implementation, as the techniques to handle both are the same.

Reduction Rule 5. Let $X \subseteq V$ induce a clique in a graph G, where $|X| = |N(X)| \geq 1$ and $N(X) = N(x) \setminus X$ for all $x \in X$. Create G' from G by removing an arbitrary vertex of X. Then $\beta(G) = \beta(G') + |X|$. Figure 4.4 shows an example for $|X| = 2$.

Proof. Let $S := X \cup N_G(X)$ and δ be any 2-coloring of $N_G(X)$. Note that $C_{\text{ext}(G)}(S) \subseteq N_G(X)$, i.e., the removal of $N_G(X)$ disconnects X from the remainder of the graph.

Define $V_c = \{x \in N_G(X) \mid \delta(x) = c\}$ and $z_c := |V_c|$ for $c \in \{0, 1\}$. We distribute the vertices in X among V_0 and V_1 such that $E(V_0, V_1)$ is maximized. Notice that every vertex in X is connected to all other vertices in S. The size of any cut is

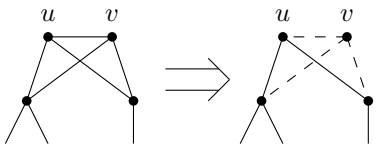


Figure 4.4: Reduction Rule 5. Vertices u and v induce a clique and have the same neighborhood of two vertices outside the clique. We remove v along with all its edges from the graph.

therefore $p(c_0, c_1) = c_0z_1 + c_1z_0 + c_0c_1 + |E(V_0, V_1)|$, where c_0 and c_1 denote the number of vertices from X that we want to insert into V_0 and V_1, respectively. This can be rewritten as $p(c_0, c_1) = (z_0 + c_0) \cdot (z_1 + c_1) - z_0z_1 + |E(V_0, V_1)|$. As all other parts are constant, this reduces to maximizing $(z_0 + c_0) \cdot (z_1 + c_1)$. As $z_0 + c_0 + z_1 + c_1$ is constant, $(z_0 + c_0) \cdot (z_1 + c_1)$ is maximized when $|(z_0 + c_0) - (z_1 + c_1)|$ is minimized.

Because $|X| = |N_G(X)|$, it is always possible to distribute the vertices of X such that $z_0 + c_0 = z_1 + c_1 = |X|$, which then maximizes $p(c_0, c_1)$. Removing any vertex $x \in X$ from G will change the cut by $-|X|$: without loss of generality, let $x \in V_0$. Then $|X| + |N_G(X)|$ is odd and $|(z_0 + (c_0 - 1)) - (z_1 + c_1)| = 1$, which maximizes the cut. Then, $p(c_0 - 1, c_1) = p(c_0, c_1) - |X|$. □

The following algorithm identifies all candidates of Reduction Rule 5 in linear time. First, we order the adjacencies of all vertices. That is, for every vertex $v \in V$, the vertices in $N(v)$ are sorted according to a numeric identifier assigned to every vertex. For this, we create an auxiliary array of empty lists of size $|V(G)|$. We then traverse the vertices $w \in N(v)$ for every vertex $v \in V(G)$ and insert each pair (v, w) in a list identified by indexing the auxiliary array with w. We then iterate once over the array from the lowest identifier to the highest and recreate the graph with sorted adjacencies. In total, this process takes $O(|V| + |E|)$ time.

For any clique X of G, we have to check if for all pairs (x_1, x_2) of vertices from X that $N[x_1] = N[x_2]$ holds (neighborhood condition). Our algorithm uses tries [Fre60; De 59] to find all candidates. A trie supports two operations, INSERT(KEY,VAL) and RETRIEVE(KEY). The KEY parameter is an array of integers and VAL is a single integer. Function RETRIEVE returns all inserted values by INSERT that have the same key. Internally, a trie stores the inserted elements as a tree, where every node corresponds to one integer of the key and every prefix is stored only once. That means that two keys sharing a prefix share the same path through the trie until the position where they differ.

For each vertex $v \in V$, we use the ordered set $N(v) \cup \{v\}$ as KEY and v as the VAL parameter. Notice that $N(v)$ is already sorted. The key $N(v) \cup \{v\}$ can be then computed through an insertion of v into the sequence $N(v)$ in time $O(|N(v)|)$. After INSERT($N(v) \cup \{v\}$,v) is done for every vertex $v \in V$, each trie leaf contains all vertices that satisfy the condition of Reduction Rule 5. Meaning, for every vertex pair (x_1, x_2) of a trie leaf, the neighborhood condition is met. We then verify

whether the vertex set X of a leaf is a clique, in $O(|X|^2)$ time. Since every vertex is only contained one leaf and each leaf forms a clique due to the neighborhood condition, this takes $O(|V| + |E|)$ time in total. As a last step, we check whether $|X| > |N(X)| \geq 1$ by using the observation that $\forall x \in X : |N(X)| = \deg(x) - |X|$. In Section 4.4, we describe a timestamping system that assists the above procedure in not having to repeatedly check the same structures after any amount of vertices and edges are added or removed from G. However, in those later applicability checks, we disregard sorting the adjacencies of all vertices in linear time again. Rather we simply use a comparison based sort on the adjacencies.

The next reduction rule is our only rule whose application turns unweighted instances into instances of Weighted Max Cut. Our experiments show that this can reduce the kernel size substantially. This is noteworthy, given that existing solvers for `Max Cut` usually support weighted instances.

Reduction Rule 6. Let G be a graph, $w : E \to \mathbb{Z}$ a weight function, and (a, b, a') be an induced 2-path with $N(b) = \{a, a'\}$. Let e_1 be the edge between vertex a and b; let e_2 be the one between b and a'. Construct G' from G by deleting vertex b and adding a new edge $\{a, a'\}$ with $w'(\{a, a'\}) = \max\{w(e_1), w(e_2)\} - \max\{0, w(e_1) + w(e_2)\}$. Then $\beta(G, w) = \beta(G', w) + \max\{0, w(e_1) + w(e_2)\}$.

Proof. Let δ be a maximum cut of G and consider the following two cases:

- $\delta(a) = \delta(a')$: If $w(e_1) + w(e_2) > 0$, then cutting both edges increases the cut, so $\delta(b) \neq \delta(a)$. Otherwise, $\delta(b) = \delta(a)$. In total, the path contributes $\max\{0, w(e_1) + w(e_2)\}$ to the cut. In G', the edge between a and a' is not cut, so $\beta(G, w) = \beta(G', w') + \max\{0, w(e_1) + w(e_2)\}$.

- $\delta(a) \neq \delta(a')$: If $w(e_1) > w(e_2)$, then cutting e_1 increases the cut by more than e_2, so $\delta(b) = \delta(a')$. Otherwise, $\delta(b) = \delta(a)$. In total, the path contributes $\max\{w(e_1), w(e_2)\}$ to the cut. In G', the edge between a and a' is cut and contributes $w'(\{a, a'\}) = \max\{w(e_1), w(e_2)\} - \max\{0, w(e_1) + w(e_2)\}$ to the cut, so again $\beta(G, w) = \beta(G', w') + \max\{0, w(e_1) + w(e_2)\}$. □

Our next two rules (Reduction Rules $7_{w=1}$ and 7) generalize Reduction Rule 8 by Crowston et al. [Cro+13], which we restate for completeness.

Reduction Rule 8. ([Cro+13], Reduction Rule 8)
Let (G, l) be a signed graph, $S \subseteq V$ a set of vertices such that $G[V \setminus S]$ is a clique forest, and C a block in $G[V \setminus S]$. If there is a $X \subseteq C_{\text{int}(G[V \setminus S])}(C)$ such that $|X| > \frac{|C|+|N(X)\cap S|}{2} \geq 1$, $N_l^+(x) \cap S = N_l^+(X) \cap S$ and $N_l^-(x) \cap S = N_l^-(X) \cap S$ for all $x \in X$. Construct the graph G' from G by removing any two vertices $x_1, x_2 \in X$, then $\beta(G') - EE(G') = \beta(G) - EE(G)$.

Note that, for unsigned graphs, $N_l^+(x) = \varnothing$ and $N_l^-(x) = N(x)$ for every vertex x.

Here, different choices of S lead to different applications of this rule. Our generalizations do not require such a set anymore and can find *all* possible applications for any choice of S.

Reduction Rule $7_{w=1}$. Let X be the vertex set of a clique in G with $|X| > \max\{|N(X)|, 1\}$ and $N(X) = N(x) \setminus X$ for all $x \in X$. Construct the graph G' by deleting two arbitrary vertices $x_1, x_2 \in X$ from G. Then $\beta(G) = \beta(G') + |N(x_1)|$. See Figure 4.5 for an example.

We show the correctness of Reduction Rule $7_{w=1}$ by reducing it to Reduction Rule 8 by Crowston et al. [Cro+13].

Proof. Let $S = V \setminus X$ and $C = X$. Since X is a clique, $G[V \setminus S]$ is a clique forest. From $|X| > \max\{|N(X)|, 1\}$ it follows that $|X| > \frac{|X|+|N(X)|}{2} = \frac{|C|+|N(X)\cap S|}{2} \geq 1$. Also, $N(x) \setminus X = N(x) \cap S$ and $N(X) \cap S = N(X)$, so all conditions for Reduction Rule 8 are satisfied.

It remains to show that $\beta(G) = \beta(G') + |N(x_1)|$. Note that $|E(G')| = |E(G)| - |N_G(x_1)| - (|N_G(x_2)| - 1)$ and $|V(G')| = |V(G)| - 2$. By Reduction Rule 8, we know that $\beta(G') - EE(G') = \beta(G) - EE(G)$, therefore we have that

$$
\begin{aligned}
&\beta(G) - \beta(G') && (4.1)\\
&= EE(G) - EE(G')\\
&= \frac{|E(G)|}{2} + \frac{|V(G)|-1}{4} - \left(\frac{|E(G')|}{2} + \frac{|V(G')|-1}{4}\right)\\
&= \frac{|E(G)|}{2} + \frac{|V(G)|-1}{4} - \left(\frac{|E(G)| - |N_G(x_1)| - (|N_G(x_2)|-1)}{2} + \frac{(|V(G)|-2)-1}{4}\right)\\
&= \frac{(|V(G)|-1) - |V(G)| + 2 + 1}{4} - \frac{-|N_G(x_1)| - (|N_G(x_2)|-1)}{2}\\
&= \frac{2}{4} - \frac{-|N_G(x_1)| - |N_G(x_1)| + 1}{2} && (4.2)\\
&= \frac{2}{4} - \frac{-2|N_G(x_1)| + 1}{2}\\
&= \frac{2}{4} - \frac{1}{2} + |N_G(x_1)|\\
&= |N_G(x_1)|.
\end{aligned}
$$

Where (4.2) follows from $N_G(x_1) = N_G(x_2)$. □

Reduction Rule 7. Let $X \subseteq V$ induce a clique in a signed graph (G, l) such that $\forall e \in E(X) : l(e) =$ "-" and $|X| > \max\{|N(X)|, 1\}$, $N_l^+(X) = N_l^+(x) \setminus X$, and $N_l^-(X) = N_l^-(x) \setminus X$ for all $x \in X$. Construct G' by deleting two arbitrary vertices $x_1, x_2 \in X$ from G. Then $\beta(G) = \beta(G') + |N(x_1)|$.

Proof (Sketch). The proof for this rule is almost identical to the proof of Reduction Rule $7_{w=1}$. □

Using an almost equivalent approach as we did for Reduction Rule 5, we can find all candidates of this reduction rule in linear time.

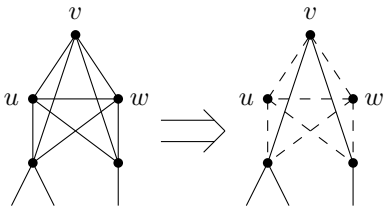


Figure 4.5: Reduction Rule $7_{w=1}$. Vertices u, v and w induce a clique and have the same neighborhood of two vertices outside the clique. We remove u and w along with all their edges from the graph.

In order to also reduce weighted instances to some degree, we use a simple weighted scaling of two reduction rules. That is, we extend their applicability from an unweighted subgraph to a subgraph where all edges have the same weight $c \in \mathbb{R}$. We do this for Reduction Rules 1 and 3.

Reduction Rule $1_{w=c}$. Let (G, ω) be a weighted graph and let $S \subseteq V(G)$ induce a clique with $\omega(e) = c$ for every edge $e \in E(G[S])$ for some constant $c \in \mathbb{R}$. Let $G' = (V(G) \setminus C_{\text{int}(G)}(S), E(G) \setminus E(G[S]))$ with $\omega'(e) = \omega(e)$ for every $e \in E(G')$. If $|C_{\text{ext}(G)}(S)| \leq \lceil \frac{|S|}{2} \rceil$, then $\beta(G, \omega) = \beta(G', \omega') + c \cdot \beta(K_{|S|})$.

Reduction Rule $3_{w=c}$. Let (G, ω) be a weighted graph and let $S \subseteq V(G)$ induce a near-clique in G. Furthermore, let $\omega(e) = c$ for every edge $e \in E(G[S])$ for some constant $c \in \mathbb{R}$. Let G' be the graph obtained from G by adding the edge e' so that S induces a clique in G'. Set $\omega'(e') = c$, and $\omega'(e) = \omega(e)$ for $e \in E(G)$. If $|S|$ is odd or $|C_{\text{int}(G)}(S)| > 2$, then $\beta(G, \omega) = \beta(G', \omega')$.

4.4 Implementation

4.4.1 Kernelization Framework

We now discuss our *overall* kernelization framework in detail. Our algorithm begins by generating an unweighted instance by replacing every weighted edge by an unweighted subgraph with a specific structure. Afterwards, we apply our full set of unweighted reduction rules: 1, $7_{w=1}$ (together with 5), 2, and 3. We then create a signed instance of the graph by exhaustively executing weighted path compression using Reduction Rule 6 with the restriction that the resulting weights are -1 or $+1$. We then exhaustively apply Reduction Rule 7. Once the signed reductions are done, we apply Reduction Rule 6 to fully compress all paths into weighted edges. This is then succeeded by Reduction Rule $1_{w=c}$ and $3_{w=c}$. We then transform the instance into an unweighted one and apply Reduction Rule 4 in order to avoid cyclic interactions between itself and Reduction Rule 3. Finally, if a weighted solver is to be used on the kernel, we exhaustively perform Reduction Rule 6 to produce a

weighted kernel. Note that different permutations of the order in which reduction rules are applied can lead to different results.

4.4.2 Timestamping

Next, we describe how to avoid unnecessary checks for the applicability of reduction rules. For this purpose, let the time of the most recent change in the neighborhood of a vertex be $T : V(G) \to \mathbb{N}_0$ and let the variable $t \in \mathbb{N}$ describe the current time. Initially, $T(v) = 0, \forall v \in V$ and $t = 1$. Every time a reduction rule performs a change on $N(v)$, set $T(v) = t$ and increment t. For each individual Reduction Rule r, we also maintain a timestamp $t_r \in \mathbb{N}_0$ (initialized with 0), indicating the upper bound up to which all vertices have already been processes. Hence, all vertices $v \in V$ with $T(v) \leq t_r$ do not need to be checked again by Reduction Rule r. Note that timestamping only works for "local" reduction rules—the rules whose applicability can be determined by investigating the neighborhood of a vertex. Therefore, we only use this technique for Reduction Rules 1 and 7.

4.5 Experimental Evaluation

4.5.1 Methodology and Setup

All of our experiments were run on a machine with four Octa-Core Intel Xeon E5-4640 processors running at 2.40 GHz CPUs with 512 GB of main memory. The machine runs Ubuntu 18.04. All algorithms were implemented in C++ and compiled using gcc version 7.3.0 with optimization flag `-O3`. We use the following state-of-the-art Weighted Max Cut solvers for comparisons: the exact solvers LocalSolver [Ben+11] (heuristically finds a large cut, and can then verify if it is maximum), Biq Mac [RRW10] as well as the heuristic solver MqLib [DGS18]. MqLib is unable to determine on its own when it reaches a maximum cut and always exhausts the given time limit. We also evaluated an implementation of the reduction rules used by Etscheid and Mnich [EM18]; however, preliminary experiments indicated that our new reduction rules produce smaller kernels in less time. In the following, for a graph $G = (V, E)$, G_{ker} denotes the graph after all reductions have been applied exhaustively. For this purpose, we examine the following efficiency metric: we denote the *kernelization efficiency* by $e(G) = 1 - |V(G_{\text{ker}})|/|V(G)|$. Note that $e(G)$ is 1 when all vertices are removed after applying all reduction rules, and 0 if no vertices are removed.

For our experiments we use four different datasets: First, we use random instances from four different graph models that were generated using the KaGen graph generator [Fun+19; SS16]. In particular, we used Erdős-Rényi graphs (GNM), random geometric graphs (RGG2D), random hyperbolic graphs (RHG) and Barabási-Albert graphs (BA). The main purpose of these instances is to study the effectiveness of individual reduction rules for a variety of graph densities and degree distributions.

Table 4.2: Number of vertices $|V|$ and edges $|E|$ of our benchmark graphs.

Rudy instances [Wie18]:		
Graph	$\|V\|$	$\|E\|$
g05_60	60	885
g05_80	80	1 580
g05_100	100	2 475
pm1d_80	80	3 128
pm1d_100	100	4 901
pm1s_80	79	316
pm1s_100	100	495
pw01_100	100	495
pw05_100	100	2 475
pw09_100	100	4 455
w01_100	100	470
w05_100	100	2 356
w09_100	100	4 245

Medium-sized instances:			
Graph	$\|V\|$	$\|E\|$	source
ca-CSphd	1 882	1 740	[RA15]
ego-facebook	2 888	2 981	[RA15]
ENZYMES_g295	123	278	[RA15]
road-euroroad	1 174	1 417	[RA15]
bio-yeast	1458	1 948	[RA15]
rt-twitter-copen	761	1 029	[RA15]
bio-diseasome	516	1 188	[RA15]
ca-netscience	379	914	[RA15]
soc-firm-hi-tech	33	125	[RA15]
g000302	317	476	[DGS18]
g001918	777	1 239	[DGS18]
g000981	110	188	[DGS18]
g001207	84	149	[DGS18]
g000292	212	381	[DGS18]
imgseg_271031	900	1 027	[DGS18]
imgseg_105019	3 548	4 325	[DGS18]
imgseg_35058	1 274	1 806	[DGS18]
imgseg_374020	5 735	4 427	[DGS18]
imgseg_106025	1 565	2 629	[DGS18]

Large-sized instances [RA15]:		
Graph	$\|V\|$	$\|E\|$
inf-road_central	14 081 816	16 933 413
inf-power	4 941	6 594
web-google	1 299	2 773
ca-MathSciNet	332 689	820 644
ca-IMDB	896 305	3 782 463
web-Stanford	281 903	1 992 636
web-it-2004	509 338	7 178 413
ca-coauthors-dblp	540 486	15 245 729

To analyze the practical impact of our algorithm on current-state-of-the-art solvers we use a selection of sparse real-world instances by Rossi and Ahmed [RA15], as well as instances from VLSI design (`g00*`) and image segmentation (`imgseg-*`) by Dunning et al. [DGS18]. Note that the original instances by Dunning et al. [DGS18] use floating-point weights that we scaled to integer weights. We further subdivide these instances into medium- and large-sized instances. Finally, we evaluate denser instances taken from the rudy category of the Biq Mac Library [Wie18]. See Table 4.2 for an overview of our benchmark graphs.

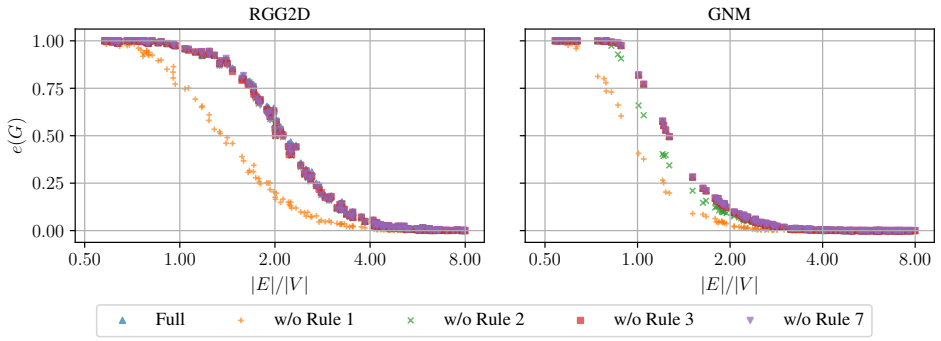


Figure 4.6: Tests consist of 150 synthetic instances. We compare the kernelization efficiency of our full algorithm to the efficiency of our algorithm without a particular reduction rule.

4.5.2 Performance of Individual Rules

To analyze the impact of each individual reduction rule, we measure the size of the kernel our algorithm procedures when removing a single rule. Fig. 4.6 shows our results on RGG2D and GNM graphs with 2048 vertices and varying density. We have settled on those two types of graphs as they represent different ends on the spectrum of kernelization efficiency. In particular, kernelization performs good on instances that are sparse and have a non-uniform degree distribution. Such properties are given by the random geometric graph model used for generating the RGG2D instances. Likewise, kernelization performs poor on the uniform random graphs that make up the GNM instances. We excluded Reduction Rule 4 from these experiments as it only removes edges and thus leads to no difference in the kernelization efficiency.

Looking at Fig. 4.6, we can see that Reduction Rule 1 gives the most substantial reduction in size. Its absence always diminishes the result more than any other rule. In particular, we see a difference in efficiency of up to 0.47 (RGG2D) and 0.41 (GNM) when removing Reduction Rule 1. The second most impactful rule for the RGG2D instances is Reduction Rule 7 with a difference of only up to 0.04. For the GNM instances Reduction Rule 2 is second with a difference of up to 0.17. However, note that Reduction Rules 3 and 7 lead to no difference in efficiency on these instances. Thus, we can conclude that depending on the graph type, different reduction rules have varying importance. Furthermore, our simple Reduction Rule 1 seems to have the most significant impact on the overall kernelization efficiency. Note that this is in line with the theoretical results from Table 4.1, which states that Reduction Rule 1 covers most of the previously published reduction rules and Reduction Rule 2 still covers many but less rules from previous work.

Table 4.3: Impact of kernelization on the computation of a maximum cut by LocalSolver (LS) and Biq Mac (BM). Times are given in seconds. Kernelization is accounted for within the timings for G_{ker}. Values in brackets provide the speedup and are derived from $\frac{T(G)}{T(G_{\text{ker}})}$. Times labeled with "-" exceeded the ten-hour time limit and an "f" indicates the solver crashed.

Name	$\|V\|$	$e(G)$	$T_{\text{LS}}(G)$	$T_{\text{LS}}(G_{\text{ker}})$		$T_{\text{BM}}(G)$	$T_{\text{BM}}(G_{\text{ker}})$	
ca-CSphd	1 882	0.99	24.07	0.32	[75.40]	-	0.06	[∞]
ego-facebook	2 888	1.00	20.09	0.09	[228.91]	-	0.01	[∞]
ENZYMES_g295	123	0.86	1.22	0.33	[3.70]	0.82	0.13	[6.57]
road-euroroad	1 174	0.79	-	-	-	-	-	-
bio-yeast	1458	0.81	-	-	-	-	32 726.75	[∞]
rt-twitter-copen	761	0.85	-	834.71	[∞]	-	1.77	[∞]
bio-diseasome	516	0.93	-	4.91	[∞]	-	0.07	[∞]
ca-netscience	379	0.77	-	956.03	[∞]	-	0.67	[∞]
soc-firm-hi-tech	33	0.36	4.67	1.61	[2.90]	0.09	0.06	[1.41]
g000302	317	0.21	0.58	0.49	[1.17]	1.88	0.74	[2.53]
g001918	777	0.12	1.47	1.41	[1.04]	31.11	17.45	[1.78]
g000981	110	0.28	10.73	4.73	[2.27]	531.47	21.53	[24.68]
g001207	84	0.19	1.10	0.16	[6.88]	53.20	0.06	[962.38]
g000292	212	0.03	0.45	0.45	[1.01]	0.43	0.37	[1.14]
imgseg_271031	900	0.99	10.66	0.19	[55.94]	-	0.17	[∞]
imgseg_105019	3 548	0.93	234.01	22.68	[10.32]	f	13 748.62	[∞]
imgseg_35058	1 274	0.37	34.93	24.71	[1.41]	-	-	-
imgseg_374020	5 735	0.82	1 739.11	72.23	[24.08]	f	-	-
imgseg_106025	1 565	0.68	159.31	34.05	[4.68]	-	-	-

4.5.3 Exactly Computing a Maximum Cut

To examine the improvements kernelization brings for medium-sized instances, we compare the time required to obtain a maximum cut for both the kernelized and the original instance. We performed these experiments using both LocalSolver and Biq Mac. Note that we did not use MQLib as it is not able to verify the optimality of the cut it computes. The results of our experiments for our set of real-world instances are given in Table 4.3 (with weighted path compression) and Table 4.4 (without weighted path compression). Since the image segmentation instances are already weighted, they are omitted from Table 4.4. It is noteworthy that we do not include the results for the rudy instances from the Biq Mac library. These instances feature a uniform edge distribution and an overall average degree of at least 3.5. Our preliminary experiments indicated that kernelization provides little to no reduction in size for these instances. Therefore, we omit them from further evaluation and focus on more sparse graphs.

First, we notice that kernelization is able to provide moderate to substantial speedups for all instances that we have tested. In particular, we achieve a speedup

Table 4.4: Impact of kernelization on the computation of a maximum cut by LocalSolver (LS) and Biq Mac (BM). Times are given in seconds. Kernelization time is included in the solving times for G_{ker}. Values in brackets provide the speedup and are derived from $\frac{T(G)}{T(G_{\text{ker}})}$. Times labeled with "-" exceeded the ten-hour time limit. Weighted path compression by Reduction Rule 6 *is not* used at the end—the kernel is unweighted.

Name	$\|V\|$	$e(G)$	$T_{\text{LS}}(G)$	$T_{\text{LS}}(G_{\text{ker}})$		$T_{\text{BM}}(G)$	$T_{\text{BM}}(G_{\text{ker}})$	
`ca-CSphd`	1 882	0.98	24.79	1.12	[22.23]	-	0.32	[∞]
`ego-facebook`	2 888	0.93	20.39	1.72	[11.83]	967.99	1.42	[682.04]
`ENZYMES_g295`	123	0.82	1.83	0.36	[5.09]	0.96	0.37	[2.60]
`road-euroroad`	1 174	0.69	-	-	-	-	-	-
`bio-yeast`	1 458	0.72	-	-	-	-	-	-
`rt-twitter-copen`	761	0.80	-	409.47	[∞]	-	101.14	[∞]
`bio-diseasome`	516	0.93	-	6.66	[∞]	-	0.35	[∞]
`ca-netscience`	379	0.67	-	4 116.61	[∞]	-	2.10	[∞]
`soc-firm-hi-tech`	33	0.30	4.92	2.34	[2.10]	0.29	0.31	[0.94]
`g000302`	317	0.10	0.71	0.50	[1.41]	1.28	0.89	[1.44]
`g001918`	777	0.06	1.67	1.51	[1.10]	14.90	11.69	[1.27]
`g000981`	110	0.22	11.32	1.97	[5.74]	0.98	0.44	[2.23]
`g001207`	84	0.17	1.56	0.15	[10.11]	0.47	0.37	[1.28]
`g000292`	212	0.01	0.69	0.51	[1.35]	0.56	0.62	[0.91]

between 1.04 and 228.91 for instances that were previously solvable by LocalSolver. Likewise, for the instances that Biq Mac is able to process, we achieve a speedup of up to three orders of magnitude. Furthermore, we allow these solvers to now compute a maximum cut for a majority of instances that have previously been infeasible in less than 17 minutes.

To examine the impact when allowing a weighted kernel, we now compare the performance our algorithm using weighted path compression (Table 4.3) with the unweighted version (Table 4.4). We can see that by including weighted path compression we can achieve substantially better speedups, especially for the sparse real-world instances by Rossi and Ahmed [RA15]. For example, on `ego-facebook` we achieve a speedup of 228.91 with compression and 11.83 without.

Finally, it is also noteworthy that we get substantial improvements for the weighted instances from VLSI design and image segmentation. By examining the performance of each individual reduction rule, we can see that this is solely due to Reduction Rule $1_{w=c}$. These findings could improve the work by de Sousa et al. [dSHK13], which also affects the work by Dunning et al. [DGS18]. In conclusion, our novel reduction rules give us a simple but powerful tool for speeding up existing state-of-the-art solvers for computing maximum cuts. Moreover, as mentioned previously, even our simple weighted path compression by itself is able to have a significant impact.

Table 4.5: Evaluation of large graph instances. A three-hour time limit was used and five iterations were performed. The columns Δ_{LS} and Δ_{MQ} indicate the percentage by which the size of the largest computed cut is larger on the kernelized graph compared to the non-kernelized one, for LocalSolver and MqLib, respectively.

Name	$\|V\|$	$\deg_{\text{avg}}$	$e(G)$	$T_{\text{ker}}(G)$	Δ_{LS}	Δ_{MQ}
`inf-road_central`	14 081 816	1.20	0.59	362.32	∞%	2.70%
`inf-power`	4 941	1.33	0.62	0.04	1.64%	0.45%
`web-google`	1 299	2.13	0.79	0.01	0.69%	0.19%
`ca-MathSciNet`	332 689	2.47	0.63	8.02	1.33%	0.55%
`ca-IMDB`	896 305	4.22	0.42	27.55	0.97%	0.32%
`web-Stanford`	281 903	7.07	0.18	105.17	0.34%	0.30%
`web-it-2004`	509 338	14.09	0.91	22.10	0.08%	0.02%
`ca-coauthors-dblp`	540 486	28.20	0.25	72.39	0.05%	0.04%

4.5.4 Analysis on Large Instances

We now examine the performance of our kernelization framework and its impact on existing solvers for large graph instances with up to millions of vertices. For this purpose, we compared the cut size over time achieved by LocalSolver and MqLib with and without our kernelization. Note that we did not use Biq Mac as it was not able to handle instances with more than 3 000 vertices. Our results using a three-hour time limit for each solver are given in Table 4.5. Furthermore, we present convergence plots in Fig. 4.7.

First, we note that the time to compute the actual kernel is relatively small. In particular, we are able to compute a kernel for a graph with 14 million vertices and edges in just over six minutes. Furthermore, we achieve an efficiency between 0.18 and 0.91 across all tested instances. When looking at the convergence plots (Fig. 4.7) we can observe that the additional preprocessing time of kernelization is quickly compensated by a significantly steeper increase in cut size compared to the unkernelized version. Furthermore, for instances where a kernel can be computed very quickly, such as `web-google`, we find a better solution almost instantaneously. In general, the results achieved by kernelization followed by the local search heuristic are *always* better than just using the local search heuristic alone. However, the final improvement on the size of the largest cut found by LocalSolver and MqLib is generally small for the given time limit of three hours.

4.6 Conclusions

We engineered new efficient data reduction rules for Max Cut that subsume most existing rules. Our experiments show that kernelization has a substantial impact in

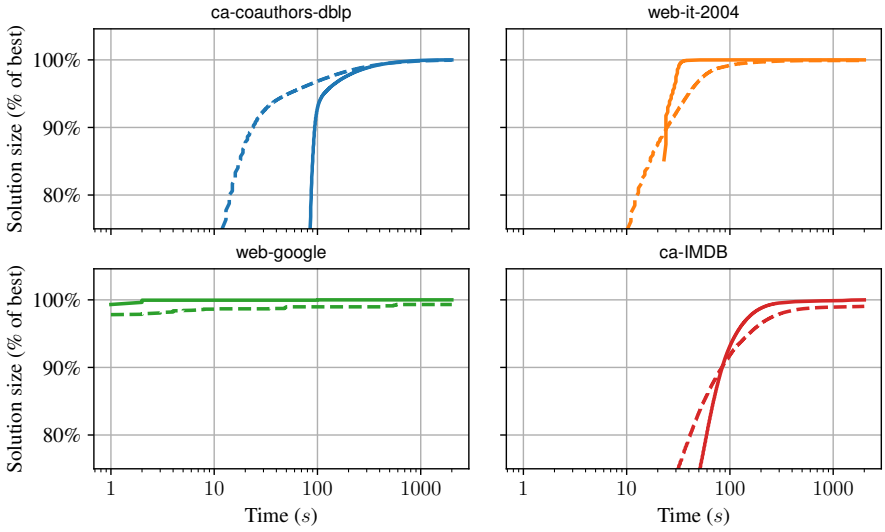


Figure 4.7: Convergence of LocalSolver on large instances. The dashed line represents the size of the cut for the non-kernelized graph, while the full line does so for the kernelized graph.

practice. In particular, our experiments reveal that current state-of-the-art solvers can be sped up by up to *multiple orders of magnitude* when combined with our data reduction rules.

Developing new reduction rules is an important direction for future research. Of particular interest are reduction rules for Weighted Max Cut, where reduction rules yield a weighted kernel.

5 Route Planning in Road Networks

Abstract. *A highly successful approach to speed up query times for route planning in networks (particularly road networks) is to identify a hierarchy in the network that allows faster queries after some preprocessing that inserts additional "shortcut"-edges into a graph. In the past there has been a succession of techniques that infer a more and more fine grained hierarchy enabling increasingly more efficient queries. This appeared to culminate in contraction hierarchies that assign one hierarchy level to each* vertex*.*

In this chapter we identify an even more fine grained hierarchy that assigns one level to each edge *of the network. Our findings indicate that this can lead to considerably smaller search spaces in terms of visited edges. Currently, this leads to comparable query times to contraction hierarchies and even improved query times on some specialized inputs. It remains an open question whether edge hierarchies can be improved to give consistently better performance. We believe that the technique as such is a noteworthy enrichment of the portfolio of available techniques that might prove useful in the future.*

References and Attribution. This chapter is based on the conference paper [HS19a]. The author of this thesis is the main author of this paper with editing done by Peter Sanders. The author made major contributions to all parts of this paper. The implementation and experiments were done by the author of this thesis. Large parts of this chapter were copied verbatim from the conference paper or the corresponding technical report [HS19b].

5.1 Introduction

Computing shortest, fastest, or otherwise optimal routes in networks is a fundamental problem that needs to be solved in many applications. For road networks alone there are multiple important applications, e.g., car navigation, traffic simulation, planning in logistics, etc. An important approach to fast route planning is to preprocess the network in such a way that subsequent queries are accelerated. In this chapter we focus on point-to-point queries in road networks but note that other types of queries or networks might also be supported in a way analogous to previous applications of contraction hierarchies [Gei+12; Bas+16].

A particularly successful class of preprocessing techniques for road networks is to exploit hierarchies inherent to the network. An informal way to describe this is that "usually" the farther away we are from source or destination, the more important are the roads we use. Hierarchical route planning techniques have a history in becoming more aggressive in exploiting the hierarchy resulting in smaller and smaller search spaces. This began with early heuristics based on road categories [Ish+91; JSQ02] and later used exact techniques that insert *shortcut edges*. Shortcuts encode that certain subpaths are important and, together with an appropriate query algorithm, ensure that optimal paths can be found using hierarchical routing techniques. Such techniques include overlay graphs [SWZ02; Del+15], reach based routing [Gut04], highway hierarchies [SS05] and highway node routing [SS07]—so far culminating in contraction hierarchies (CHs) [Gei+08; Gei+12; DSW16].

CHs order the *vertices* of the network by importance, i.e., we conceptually have n levels of hierarchy in a network with n vertices. By inserting appropriate shortcuts, CHs ensure that there exists an *up-down path* between any pair of vertices that is a shortest path. An up-down path progresses from the source vertex to more important vertices and then descends to less important vertices until reaching the destination. CHs are widely used since they are simple, allow fast preprocessing using little space and lead to very small search spaces.

In this chapter we introduce *edge hierarchies* (EHs) as an even more fine grained way to define hierarchy in the network. EHs order *edges* rather than vertices by importance. They keep the concept of up-down paths resulting in a very simple query algorithm. Intuitively, this should further reduce search space sizes. EHs[1]—in contrast to CHs—only have to explore edges out of a vertex v that are more important than the edge leading to v in the current query. Also note that EHs are very close to the informal definition of hierarchical routing that we gave above.

After introducing basic terms and techniques in Section 5.2, and discussing further related work in Section 5.3, we describe EHs in detail in Section 5.4. While the basic query algorithm is simple by design, a preprocessing algorithm finding the "right" shortcuts turns out to be much more complicated. We also discuss some techniques for pruning the query search space.

In Section 5.5 we perform an experimental evaluation using large real world road networks and different cost functions. It turns out that EHs relax significantly fewer edges than CHs in particular for cost functions that are known to be difficult for CHs: with distance as the main optimization criterion and/or explicit modeling of turn penalties. Unfortunately, the overall query time is usually slightly worse than on CHs and preprocessing time is considerably larger. Overall, EHs are thus an intriguing concept with considerable potential but they need further research to find out whether they will eventually be useful in some applications. In Section 5.6 we discuss possible research in this direction.

[1] Technically we are talking about EH queries here, but we will often refer to the algorithms associated with EHs or CHs like this.

5.2 Preliminaries

For general definitions and notations for graphs, see Section 2.1. Here we give additional definitions specific to shortest paths and the techniques used in our algorithm. Note that in this chapter, we consider *directed*, *weighted* graphs with positive edge weights, i. e., $\omega : E \to \mathbb{R}_{\geq 0}$.

The classical algorithm for finding shortest paths is Dijkstra's algorithm [Dij59]. It maintains a *distance label* (dist) for each vertex and repeatedly *settles* the vertex u with the currently smallest distance label among all unsettled vertices. It then *relaxes* all outgoing edges (u, v) by setting $\text{dist}(v) \leftarrow \min(\text{dist}(v), \text{dist}(u) + \omega(u, v))$. In the *bidirectional* version of Dijkstra's algorithm, the *forward* search from s is complemented by a *backward* search from t that only considers incoming edges of the settled vertices. This can lead to faster running times than a plain Dijkstra search because we can stop once a vertex is settled in both searches and it is essential for the query algorithm in both EHs and CHs.

A *shortcut* is an edge whose length corresponds to the length of some nontrivial path in the graph. For example, for edges $e_1 = (u, v)$ and $e_2 = (v, w)$, a shortcut $e_s = (u, w)$ with $\omega(e_s) = \omega(e_1) + \omega(e_2)$ can be added to the graph. Note that adding shortcuts does not change the distance for any pair of vertices in the graph. Also, by storing skipped vertices, we can recursively *unpack* shortcuts, e.g., by replacing e_s with e_1 and e_2 to find the corresponding path that only uses original (non-shortcut) edges.

Contraction Hierarchies [Gei+08; Gei+12; DSW16] use shortcuts to build a hierarchy where every vertex is on its own level. Vertices are repeatedly removed from the graph in order of a measure of importance. If for any pair of incoming and outgoing neighbors u, w the removed vertex v is on the *only* shortest path (u, v, w), then a shortcut (u, w) is added. Whether this shortcut is necessary is determined by a so-called *witness search* that runs a shortest path search starting at u on the *overlay graph*. The overlay graph consists of all vertices not yet removed and all edges incident to these vertices. The witness search can be restricted to stop after settling a small amount of vertices. This might add unnecessary shortcuts but does not affect correctness, while having the potential to speed up the algorithm. Vertex importance is usually determined by a combination of different measures. Metrics successfully implemented in previous work (and used in the implementation we compare against in our evaluation) are the amount of shortcuts added if the vertex were to be removed next, the number of *hops* represented by these shortcuts and an additional *level* metric that helps removing vertices uniformly throughout the graph. These numbers have in common that they only change when a neighbor of a vertex is removed from the graph. The algorithm therefore maintains all vertices in a priority queue with their importance as key. When a vertex is removed, the importance of its neighbors is updated. The query algorithm is a bidirectional Dijkstra search that only relaxes edges that connect a vertex to a more (less) important vertex in the forward (backward) search. Due to this, edges only need to be stored at the end point that is removed first.

5.3 More Related Work

There has been a lot of work on route planning. Refer to [Bas+16] for a recent overview. Here we only give selected references to describe the place of EHs in the wider context of route planning techniques. Besides hierarchical route planning techniques there are also techniques which direct the shortest path search towards the goal (e.g., landmarks [GKW06], precomputed cluster distances [MSM09], arc flags [Möh+05]). On road networks goal directed techniques are usually inferior to hierarchical ones since they need considerably more query or preprocessing time. However, combining goal directed and hierarchical route planning is a useful approach [GKW06; Bau+10]. We expect that this is also possible for EHs using the same techniques as used before. Other techniques allow very fast queries by building shortest paths directly from two (hub labeling [Abr+11]) or three (transit node routing [Bas+07; ALS13]) precomputed shortcuts without requiring a graph search. However, these methods require considerably more space than EHs.

5.4 Edge Hierarchies

The main idea of EHs is to provide a precomputed data structure that allows queries similar to those of CHs: All shortest paths can be found by a bidirectional Dijkstra search that only searches "upwards". In contrast to CHs, which build a hierarchy of vertices, EHs build a hierarchy of edges. Let $r(u, v)$ denote the *rank* assigned to the edge (u, v). Then, paths found by an EH query have the form $(s = v_0, \ldots, v_m, \ldots, v_n = t)$ with $r(v_{i-1}, v_i) \leq r(v_i, v_{i+1})$ for $0 < i \leq m$ and $r(v_{i-1}, v_i) \geq r(v_i, v_{i+1})$ for $m < i < n$ (allowing $s = v_m$ or $t = v_m$). In line with the terminology from CHs, we call such paths *up-down paths*.

The EH query is a modified version of the bidirectional variant of Dijkstra's algorithm: In addition to the distance label dist, we maintain a rank label r at every node, set to 0 for s and t. When settling a vertex u, only edges with $r(u, v) \geq r(u)$ are relaxed. Whenever $\text{dist}(v)$ is updated while relaxing an edge (u, v), $r(v)$ is set to $r(u, v)$. For a stopping condition, the algorithm maintains an upper bound $\overline{d}$ for $\text{dist}(s, t)$ (initially ∞) which is updated whenever a vertex is settled that has already been settled from the other direction. No edges leaving vertices with $\text{dist}(v) > \overline{d}$ are relaxed. Figure 5.1 illustrates the search space of an Edge Hierarchy Query. Note how the edges ranked 2 and 3 are not in the search space of the backward search, even though their target vertex is settled.

Algorithm 5.1 shows an algorithm template for constructing an EH. Initially, all edges are unranked (which we will treat as rank ∞). In iteration i, we pick an unranked edge (u, v) and set its rank to i . We then iterate over all unranked edges (u', u) and (v, v') and test whether (u', u, v, v') is a shortest path. If yes, we add either (u', v) or (u, v') as a shortcut. If either of these two edges already exists, we instead adjust its weight and reset its rank to ∞, if it was already ranked before.

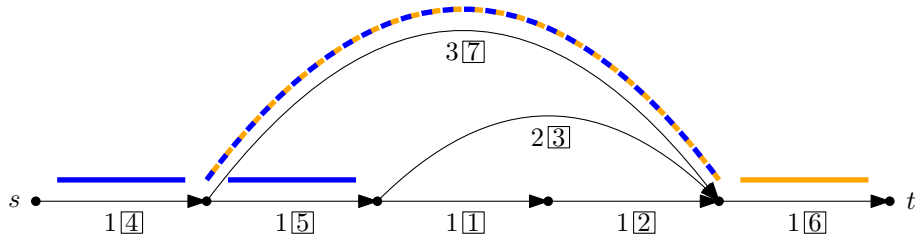


Figure 5.1: Search space of an EH Query. Blue edges are in the search space of the forward search, orange edges are in the search space of the backward search. Boxed numbers are edge ranks, unboxed numbers are edge weights.

Algorithm 5.1 : BuildEdgeHierarchy

```
1 currentRank ← 0
2 while Unranked edges remain do
3     Pick unranked edge (u, v)
4     r(u, v) ← currentRank++
5     for all unranked edges (u', u) do
6         for all unranked edges (v, v') do
7             if dist(u', v') = w(u', u) + w(u, v) + w(v, v') then
8                 Add shortcut (u', v) or (u, v')  — Or adjust weight + unset rank
```

Theorem 5.1

For every pair of vertices s and t, such that there is a path from s to t in the input graph, Algorithm 5.1 assigns ranks and adds shortcuts such that there is a shortest up-down path from s to t.

Proof. We prove this by showing the following: If at the beginning of iteration i, there is a shortest path from s to t that only uses unranked edges, then in iteration $j > i$, there exists an up-down-path p from s to t that only uses edges of rank $\geq i$. As at the beginning of the first iteration, all edges are unranked, this proves the theorem.

In iteration i, an edge e gets ranked. Let p be a shortest path from s to t consisting only of unranked edges. If e is not part of p, then p is still a shortest path that only uses unranked (rank ∞) edges (which is an up-down path by definition).

If e is at neither end of p, then a shortcut is inserted that replaces two edges of p, so there still is a shortest path only using unranked edges from s to t.

If $e = (s, v)$ (the case $e = (v, t)$ is analogous) we distinguish two cases:

(i) There still exists a shortest path of unranked edges from s to v: Then there is also a shortest path of unranked edges from s to t.

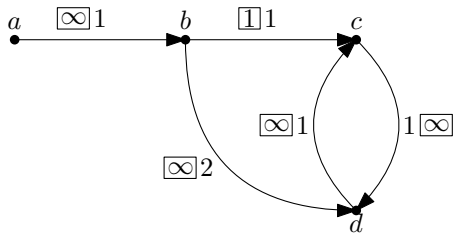


Figure 5.2: Example showing that EH construction needs to calculate distances on the complete graph. Boxed numbers are edge ranks, unboxed numbers are edge weights.

(ii) There is no shortest path of unranked edges from s to v: Then (s, v) gets assigned rank i and can never change its rank (note for this, that edges can only be inserted or assigned to a different rank if there is a shortest path of unranked edges between their endpoints). Furthermore, there is a shortest path of unranked edges from v to t. By induction, in every iteration $j > i$, there will be an up-down-path from v to t that uses only edges of rank $\geq i$. By adding the edge (s, v) to the beginning of that path, we get an up-down path from s to t.

As the induction basis, note that at the end of the algorithm, no edges are unranked, so the claim holds trivially. □

Note that from the induction in the proof above, it follows that we can use the EH query for the distance calculation in Algorithm 5.1.

The algorithm can also be slightly altered by only adding a shortcut if (u', u, v, v') is the *only* remaining unranked shortest path from u' to v'. However, preliminary experiments showed that the version presented here yields better results.

An important difference to CH construction is that Algorithm 5.1 has to calculate distances in the complete graph, whereas CH construction only has to query the overlay graph. See Figure 5.2 for an example why using the overlay graph does not suffice for EHs: If (b, d) is assigned rank 2, we need to check whether $p = (a, b, d, c)$ is a shortest path. If we use only the overlay graph for the distance calculation, then we would falsely assume that p is a shortest path and add a shortcut.

5.4.1 Shortcut Selection

The choice of the shortcut that is added in the inner loop of Algorithm 5.1 can make a significant impact on the total number of shortcuts added. For example, in Figure 5.3, we could either add the shortcut (u, v') or *all* of the shortcuts (u'_i, v) (assuming (u'_i, u, v, v') is a shortest path for all u'_i). In contrast, in CHs there is no choice of which shortcut to add. We minimize the number of shortcuts added using a solution to a minimum bipartite vertex cover problem for every iteration of the outer while-loop of Algorithm 5.1.

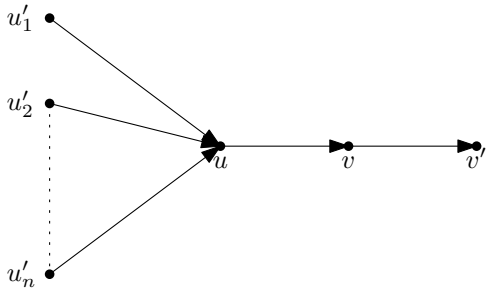


Figure 5.3: When ranking (u, v), we could either add *all* shortcuts (u'_i, v) or just (u, v').

The problem $(U \cup V, E)$ is constructed as follows: Instead of directly adding one of the two possible shortcuts, we add the vertices u', v' to U, V respectively (if they have not been added before) and an edge between them.

After all shortcut candidates for an iteration of the outer loop have been added to the bipartite graph, we compute a minimum Vertex Cover C. Note that this can be done in polynomial time via maximum cardinality bipartite matching using König's Theorem. We then add the shortcuts (u', v) for every $u' \in U \cap C$ and (u, v') for every $v' \in V \cap C$. It is easy to verify that for every pair of candidate shortcuts, one is added. Also, every set of shortcuts added implies a Vertex Cover for the graph above, so finding a *minimum* Vertex Cover minimizes the number of shortcuts added in every iteration of the construction algorithm, given the edge that is assigned a rank.

To further minimize the number of shortcuts added, we always prefer edges already present in the graph: if (u', v) or (u, v') is already in the graph (ranked or unranked), we change its weight accordingly and reset its rank. The pair (u', v') is then not added to the minimum Vertex Cover problem described above.

5.4.2 Edge Selection

In every iteration of Algorithm 5.1, we need to choose an edge to assign a rank to next. Our heuristic to select these edges is guided by two goals: Adding a small number of shortcut edges to the graph, and ranking edges uniformly throughout the graph. Here, we present the version that produced the best results in our preliminary experiments. Other versions that resemble the vertex selection strategies used for CHs resulted in worse preprocessing and query times.

Our heuristic works in rounds: in the beginning of each round, a set of edges to rank is selected and fixed. Then we assign ranks to all selected edges in any order. Only when all edges selected are ranked, a new round is started and a new set of edges is selected. Edges are selected by counting for each unranked edge e the number of new shortcuts that would be added if e was ranked. This is done by

simulating an iteration of the outer while-loop of Algorithm 5.1 without actually adding any shortcuts to the graph and resetting $r(u, v)$ to ∞ afterwards. Then, we select all edges that cause the minimum number of shortcuts among all their incident edges.

5.4.3 Stalling

A technique that significantly reduces query times for CHs is called *Stall on Demand*. The idea is to stall the search at vertices that do not lie on a shortest path from s to t by checking whether a shorter path can be found via incoming (outgoing) downward edges in the forward (backward) search. This can happen because CHs only guarantee shortest up-down paths between any pairs of vertices. The same is true for EHs. We present two stalling techniques that can be used with EHs.

Stall on Demand. In EHs, any edge can be a downward or an upward edge depending on the rank of the edges leading to the source vertex of that edge. Stall on Demand checks *all* incoming (outgoing) edges in the forward (backward) search.

Stall in Advance. Stall on Demand may relax every edge twice: Once when settling the source (target) vertex and once for stalling when settling the target (source) vertex in the forward (backward) search. *Stall in Advance* relaxes every edge at most once: when settling a vertex u, we not only relax all outgoing (incoming) edges that are ranked higher than the path to u, but also all edges (u, v) that are ranked lower. However, we do not update $\text{dist}(v)$ with the distance computed via the low ranked edges. Instead, we store it in a separate $\text{stallDist}(v)$ label. To check whether we can stall the search at vertex v, we compare $\text{dist}(v)$ with $\text{stallDist}(v)$. If stallDist is smaller, we can stall at v.

5.5 Experimental Evaluation

We implement EHs in C++ and compile with gcc 7.4.0 using full optimizations (`-O3`). Our implementation of the construction algorithm is relatively straight forward without much emphasis on optimizations. For queries, we use adjacency arrays for incoming and outgoing edges and sort all edges incident to a vertex in descending order of their rank. This way we can stop iterating over a vertex's neighborhood once we find an edge with a lower rank than allowed for the current path. Additionally, we reorder the vertices in depth-first-search-preorder for better memory locality. The EH *construction* algorithm uses CH queries to find the distance between two vertices. The source code is available on GitHub[2].

[2] `https://github.com/Hespian/EdgeHierarchies`

For comparison with CHs, we use the implementation from RoutingKit[3] [DSW16] where queries use Stall on Demand.

The machine used for all experiments is equipped with 4 x Intel Xeon E5-4640 at 2.4 GHz and 512 GiB DDR3-PC1600 RAM but only a single core is utilized.

5.5.1 Data Sets

We evaluate EHs on two benchmark graphs from the DIMACS Challenge on Shortest Paths [DGJ09]: The road network of Western Europe from PTV AG with 18 million vertices and 42 million directed edges, and the TIGER/USA road network with 23 million vertices and 29 million undirected edges (resulting in 58 million directed edges), as well as smaller subsets of the TIGER/USA graph. Both graphs are available with edge weights corresponding to travel times or geographic distance.

In addition to these graphs, we also evaluate the performance on graphs that model the cost for taking turns at a crossing. We follow the approach used in [Del+15; Bas+16] to define simple turn costs that reportedly yield performance characteristics similar to truly realistic values: For the travel time metric, we assign costs of 100 seconds for U-turns (meaning an edge pair $(u, v), (v, u)$) and 0 for all other turns. For the distance metric, all turns are free. We explicitly model turns into our graphs. This can be done by splitting every vertex v into a number of vertices equal to its degree and connecting each new vertex to one of v's incident edges. Then, edges between the new vertices are added: For each vertex incident to one of v's incoming edges, an edge is added to each of the vertices incident to one of v's outgoing edges. The weights of these new edges are set to the turn costs. We use a more compact representation of the same concept: We only split a vertex into a number of vertices equal to its outgoing degree and connect incoming edges directly to these new vertices, adding the turn costs to the edge weights. Figure 5.4 shows an example for travel times. Table 5.1 lists all instances and their sizes used in our evaluation.

The distance metric as well as adding turn information are cases in which CHs were shown to perform significantly worse than with the travel time metric and without turn information (e.g., [Del+15]).

Table 5.1: Instances used in our evaluation. *With turns* are original instances with added turns.

	Original		With turns	
Graph	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$
USA.BAY	321 270	794 830	794 830	2 279 208
USA.W	6 262 104	15 119 284	15 119 284	41 815 474
USA.CTR	14 081 816	33 866 826	33 866 826	93 609 832
USA	23 947 347	57 708 624	57 708 624	159 734 066
EUROPE	18 010 173	42 188 664	42 188 664	113 953 602

[3] `https://github.com/RoutingKit/RoutingKit`

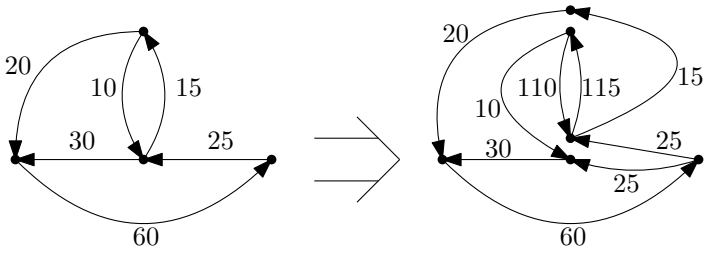


Figure 5.4: Left: Original graph. Right: Graph with added turns. 100 seconds are added to the edges corresponding to a U-turn.

5.5.2 Choosing the Right Stalling Technique

In this section we evaluate the stalling techniques explained in Section 5.4.3. To get some insight in how stalling performs for other techniques, we compare to Stall on Demand for CHs. Tables 5.2 and 5.3 compare the query times, number of vertices settled and edges relaxed for different stalling techniques averaged over 100 000 random queries. The number of edges actually relaxed and the number of edges "relaxed" to check whether the search can be stalled are shown separately. We also count the number of vertices that are settled at their actual distance to the source vertex (min. vertices). This gives an insight into how many vertices would be settled with a *perfect* stalling technique. For the travel time metric, EHs with both Stall on Demand and Stall in Advance perform more stall checks than CHs, outweighing the savings in number of vertices settled and leading to longer query times than without any stalling. For the distance metric, Stall on Demand reduces the number of vertices settled for EHs to less than for CHs. The total of number of edges touched is also smaller for EHs. However, running times are still faster without stalling because fewer edges are relaxed (or considered for stalling) and thus fewer distance labels are touched. Due to the additional distance label, Stall in Advance significantly increases query times. The last column also shows that stalling holds more potential for CHs than for EHs. However, we also see that EHs already perform relatively well without stalling: CHs on the travel time metric would have to settle more than twice as many vertices as EHs if no stalling was used. Even when not counting the stall checks, CHs with Stall on Demand relax more edges than EHs. For the distance metric, this is even more severe: Here, the search space for CHs without Stall on Demand increases so much that query times increase to over 3 ms. EHs already settle a reasonably small number of vertices without stalling.

These experiments show that the increased number of edges touched outweighs the decreased number of vertices settled. Thus, a stalling technique that only touches *some* more edges might lead to improved running times if it successfully stalls at enough vertices. Figure 5.5 shows the performance when only a fraction

Table 5.2: Query results for different stalling techniques for Edge Hierarchies and Contraction Hierarchies on the EUROPE road network with the **travel time** metric and **turns**.

Algo.	Stalling	time [μs]	settled	relaxed	stall checks	min. vertices
EH	-	199	906	1734	-	361
	S. on Demand	250	604	958	11920	
	S. in Advance	471	614	982	10563	
CH	S. on Demand	130	533	1969	2888	253
	-	338	1996	15500	-	

Table 5.3: Query results for different stalling techniques for Edge Hierarchies and Contraction Hierarchies on the EUROPE road network with the **distance** metric and **turns**.

Algo.	Stalling	time [μs]	settled	relaxed	stall checks	min. vertices
EH	-	608	2573	5586	-	638
	S. on Demand	642	1368	2276	29192	
	S. in Advance	1387	1439	2442	26959	
CH	S. on Demand	634	1943	16849	25007	704
	-	3403	12320	300758	-	

of the edges incident to a vertex are considered for Stall on Demand—going from high ranked edges to low ranked edges (note that this can be done efficiently in our implementation as edges are stored ordered by their rank). We are going to refer to this as *partial stalling* from here on. We see a slight increase in running time due to the associated calculations (see the data point for $x = 0.0$) but all configurations shown benefit from partial stalling for some fraction of edges (10% for travel times and 30% for distances).

5.5.3 Main Results

As EHs share similarities with CHs, both using similar query algorithms, we compare the two with respect to their preprocessing and query times as well as the number of vertices settled and edges relaxed during queries. Another interesting property is the number of edges in the hierarchy. Note however, that CHs only store each edge once, whereas EHs need to store each edge at both endpoints. Tables 5.4 and 5.5 show these numbers averaged over 100 000 random queries. We execute queries without Stall on Demand and with partial stalling in increments of 10%. The numbers reported here are for the best query times among these stalling configurations as indicated by the last column. In a real-world system the optimal configuration could be found as a part of the preprocessing step. Checking whether the search can be stalled at a vertex is essentially an edge relaxation (minus priority queue operations), so we combine these numbers here. We can see that EHs suffer less from adding turns to the graphs than CHs. While the number of shortcuts added is comparable for EHs and CHs on the original graphs (with CHs even adding slightly

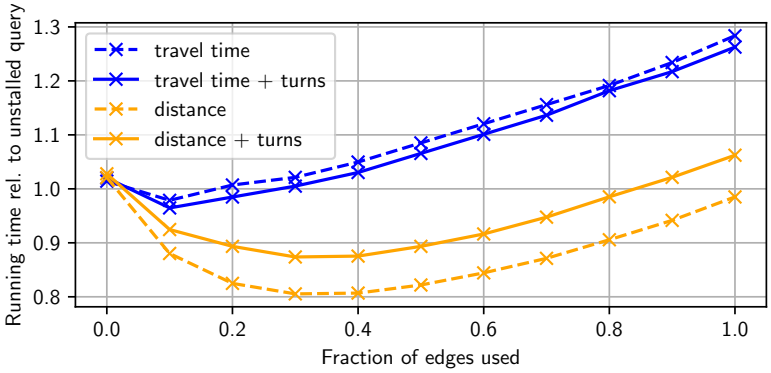


Figure 5.5: Running time of query with partial stalling relative to unstalled query with different fractions of edges used for stalling. Times were measured on the EUROPE road network.

fewer), CHs add substantially more when turns are added. This can also be seen in the number of edges relaxed: The number of edges relaxed with and without turns are very similar for EHs. For the distance metric, EHs perform even better when adding turns than on the original inputs. With turns, EHs almost always relax less than half as many edges as CHs. This shows that the intuition behind EHs—ranking *roads* (edges) rather than *junctions* (vertices)—helps to better prune roads that are irrelevant for the query. However, CHs usually settle between 2 and 3 times fewer vertices (except for the distance metric with turns where EHs often settle fewer vertices than CHs). Overall this leads to longer query times for EHs in most cases. For the distance metric with turns, query times for EHs are close to CHs—for the EUROPE instance EHs even achieve faster queries. The preprocessing step is much faster for CHs, partially due to our unoptimized implementation, but the CH vertex ranking also only updates the neighbors of a vertex after it was ranked. The edge ranking we use, on the other hand, simulates the ranking of every edge for each round of edge selection. The CH implementation in RoutingKit also limits the number of steps done for the witness search, giving additional speed up. As EHs have to find witnesses and (depending on the edge ranking technique) calculate importance values for every *edge*, compared to CHs having to do the same for every *vertex*, longer preprocessing times are to be expected.

The random queries used for the experiments above are long-ranged on average. However, real-world queries tend to be short-ranged. For this reason, Sanders and Schultes [SS05] introduce an evaluation methodology using *Dijkstra Ranks*. When running a Dijkstra query starting at some vertex in the graph, the ith vertex removed from the priority queue is assigned Dijkstra Rank i. In other words, vertices are ordered by their distance from the source vertex. Figures 5.6 and 5.7 show the number of vertices settled, number of edges relaxed, and query times for vertices of

Table 5.4: Running times and search space sizes of Edge Hierarchies and Contraction Hierarchies on different graphs with the **travel time** metric.

	Graph	Prepr. [s]		$\|E\|$ [M]		Query [μs]		settled		relaxed		stall.
		EH	CH	EH	CH	EH	CH	EH	CH	EH	CH	%
Original	USA.BAY	100	6	1.4	1.4	37	16	301	108	710	679	-
	USA.W	1785	153	27.5	27.4	96	37	538	193	1299	1386	-
	USA.CTR	4389	482	61.5	61.1	140	53	612	254	3132	2136	10
	USA	7145	674	104.5	104.0	153	60	643	271	3320	2253	10
	EUROPE	3171	453	70.3	70.3	138	75	607	356	2443	2967	10
With turns	USA.BAY	634	156	4.0	6.0	79	67	511	362	929	3253	-
	USA.W	9403	2730	69.9	105.1	165	124	748	564	1365	4810	-
	USA.CTR	25084	7316	159.3	239.2	240	172	885	700	3126	6530	10
	USA	45904	15462	270.3	404.3	250	186	900	737	3217	6792	10
	EUROPE	17822	4743	194.0	249.1	191	130	726	533	2662	4857	10

Table 5.5: Running times and search space sizes of Edge Hierarchies and Contraction Hierarchies on different graphs with the **distance** metric.

	Graph	Prepr. [s]		$\|E\|$ [M]		Query [μs]		settled		relaxed		stall.
		EH	CH	EH	CH	EH	CH	EH	CH	EH	CH	%
Original	USA.BAY	166	9	1.5	1.5	73	30	560	180	1440	1686	-
	USA.W	3435	243	28.6	28.5	254	96	1002	446	8183	6045	20
	USA.CTR	13062	1157	65.7	65.5	526	216	1697	832	20041	15561	30
	USA	21041	1537	110.8	110.7	573	235	1769	897	21461	16787	30
	EUROPE	14487	2152	79.6	79.6	538	355	1756	1179	19793	27807	30
With turns	USA.BAY	476	158	3.6	5.7	95	92	623	470	1149	4979	-
	USA.W	8452	3338	64.9	102.3	278	250	1289	993	2564	13402	-
	USA.CTR	30313	13629	148.5	235.7	556	537	1477	1743	15286	31629	40
	USA	58025	30869	251.1	398.3	604	580	1605	1849	13712	33436	30
	EUROPE	24757	13266	172.3	267.2	533	634	1543	1943	13355	41856	30

Dijkstra Ranks $2^6, \ldots, 2^{\lfloor \log |V| \rfloor}$ from 1 000 random starting vertices. This way, the performance of algorithms can be observed for both short-ranged and long-ranged queries (and everything in between). EHs use 10% and 30% partial stalling for travel times and distances, respectively. The comparison between number of vertices settled and query time shows that the algorithm that settles fewer vertices has the faster query time and edge relaxations play a less important role. This is likely due to vertex accesses causing more cache misses than accesses to the edges of a single vertex. If one would improve the cache efficiency by better node orderings or other improvements, it seems possible that the decreased number of relaxed edges in EH queries can outweigh the increased number of settled vertices.

5.6 Future Work

For CHs there is a lot of experience with configuring the preprocessing phase. The additional complications of EH preprocessing make it likely that much better versions are possible also for EHs. Trying different ways of cleaning up the distance labels for new queries might lead to some improvements as preliminary experiments showed some effect here. Due to EHs being less cache-efficient than CHs right now,

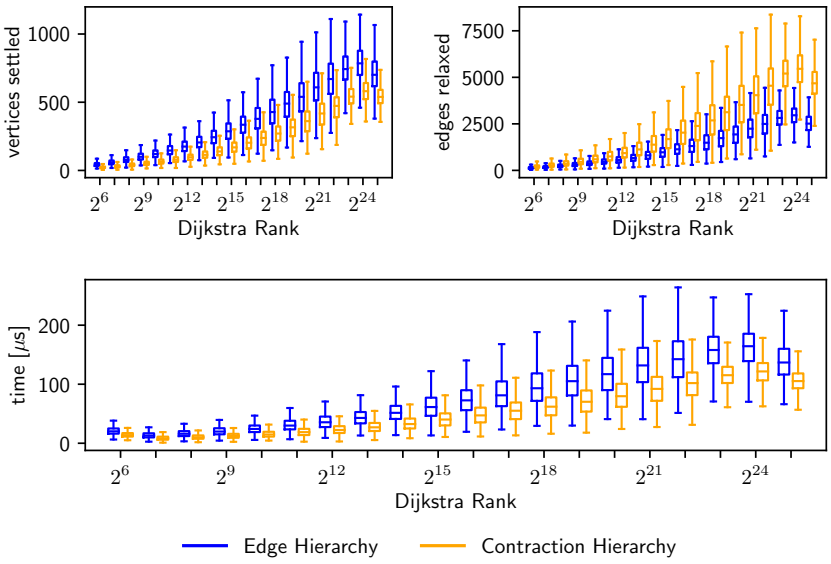


Figure 5.6: Number of vertices settled and edges relaxed, and query times for different Dijkstra Ranks on EUROPE with the **travel time** metric and **turns**.

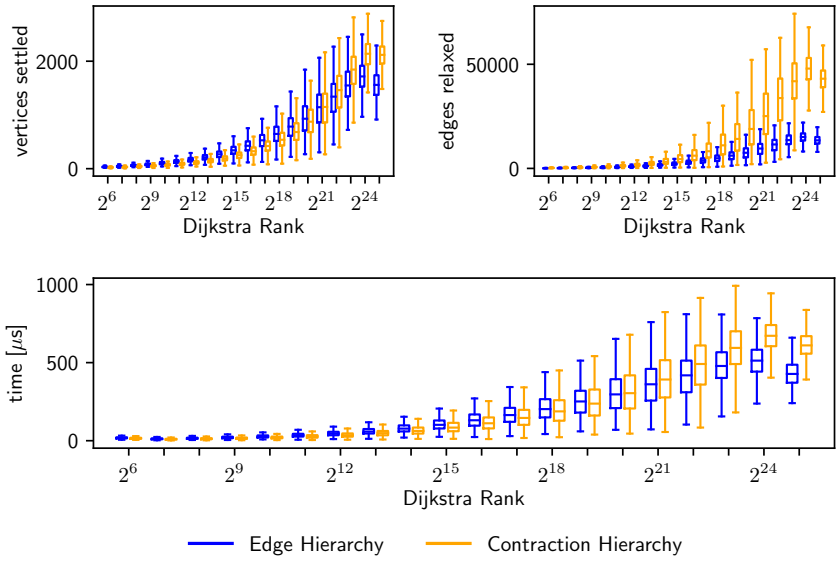


Figure 5.7: Number of vertices settled and edges relaxed, and query times for different Dijkstra Ranks on EUROPE with the **distance** metric and **turns**.

we expect them to profit more from such changes. On the application side, we can look for networks with different characteristics where EHs might have advantages. For road networks, we might harvest the advantage in the number of relaxed edges by looking at generalizations of static shortest path search where edge relaxations are expensive, e.g., time-dependent edge costs [Bat+13; KWZ16] or multicriteria shortest paths. Since first publishing EHs, some first results already found them to achieve competitive query times for stochastic route planning [RR21].

Part II

Data Recovery for Fault-Tolerant MPI Applications

Chapter 6

6 Introduction

When scaling applications and algorithms to massive inputs, we often have to use distributed systems like high performance computing (HPC) clusters; either to finish the computation in a reasonable amount of time or to even fit the input and working data into memory. With the increasing number of processors in HPC clusters, the probability that some processors fail during a computation rises. Handling such failures constitutes a major challenge for future exascale systems [SDM10]. For example ORNL's Jaguar Titan Cray XK7 system had on average 2.33 failures/day between August 2008 and 2010 [Gam+14]. In upcoming systems, we expect a hardware failure to occur every 30 to 60 minutes [Cap+14; DHR15; Sni+14]. In the following two chapters we present techniques for recovering the data lost after such a failure with the use of redundant data storage.

The first result shows a fault tolerance technique for general purpose parallel processing frameworks, which in turn are used to implement parallel algorithms using high-level abstractions. Our technique only has small overhead of about 4% for most benchmark algorithms during fault-free execution. This is achieved by exactly observing which part of the data is already available redundantly and only doing additional work for the small remaining part.

The second result is a library for programs that are not expressed in these data processing frameworks. Here, the application programmer has to explicitly specify the parts of the data that have to be recovered after a failure, which we then redundantly store using an engineered data distribution for fast recovery. Our C++ library ReStore provides an interface to specify data to be redundantly stored and efficiently retrieve it after a failure. ReStore is used to replace the previous recovery method in a widely used bioinformatics application, where recovery times are reduced by more than an order of magnitude.

References and Attribution. This introduction chapter as well as Chapter 8 are based on the conference paper [Hüb+22]. The author of this thesis is one of the main authors. Further information on contributions is provided in Chapter 8. Large segments of Chapters 6 and 8 were copied verbatim from the conference paper or the corresponding technical report [Hes+22].

6.1 Preliminaries

In distributed memory parallel programs using the *Message Passing Interface* (MPI), p processes (or *processing elements* (PEs)) run on multiple machines (or *nodes*) and communicate via messages sent over the network. We consider two important factors for evaluating the running time of such parallel algorithms: The *bottleneck number of messages sent* and *received*, and the *bottleneck communication volume*. The bottleneck number of messages sent and received describes the maximum number of messages sent or received by a single PE. This influences performance, as there is a startup overhead (latency) for establishing a connection associated with each message. As a result, sending or receiving data from one PE to another in a single message is usually faster than splitting that data into many parts and sending each part to a different receiver. The bottleneck communication volume describes the maximum amount of data sent or received by a single PE and represents a point on the critical path of the application.

As faults, we consider the case that one or multiple PEs suddenly stop working and do not contribute to the computation anymore (which we will refer to as *failed*). Following a fault, the application has to redistribute the work formerly performed by a failed PE using either the *substitute* or the *shrink* strategy [AHE18]. Under the *substitute* strategy, a replacement PE takes over the work previously performed by the failed PE. This circumvents the need for re-balancing the workload and simplifies loading the required data. However, reserving idle processors for this purpose constitutes a waste of resources. In the *shrink* strategy, the program's load balancer (re)distributes the work performed by the failed PE among the remaining (or *surviving*) PEs. This strategy does therefore not require spare PEs but requires reloading fractions of the data on many or even all PEs. While the number of failures an algorithm can tolerate using the *substitute* strategy is limited to the number of spare PEs, this limitation does not apply to the *shrink* strategy [AHE18]. Both chapters in this part will therefore focus on the *shrink* strategy.

6.2 Experimental Environment

We ran all our experiments on the SuperMUC-NG super computer.[1] Each node consists of two Intel Skylake Xeon Platinum 8174 processors with 24 cores and 96 GB of memory each, connected via an OmniPath network with a bandwidth of 100 Gbit/s. The operating system is SUSE Linux Enterprise Server 15 SP1 running Linux Kernel version 4.12.14-197.78. Unless otherwise stated, we communicate using OpenMPI version 4.0.4. The recent MPI 4 standard includes mechanisms for detecting failures and re-establishing a consistent environment after a failure. An implementation called "ULFM" is available for OpenMPI [Bla+13]. We verify that our implementations function properly if nodes actually fail and communication is recovered with ULFM as part of our fully automated unit tests. The current version

[1] `https://doku.lrz.de/display/PUBLIC/SuperMUC-NG`

of ULFM at the time of running our experiments, however, was not stable enough to conduct reliable performance benchmark experiments. For example, processes may be reported incorrectly as failed or recovery may result in two separate groups of nodes that each assume that the other group has failed. We reported this behavior on the ULFM mailing list and the authors of ULFM reproduced and confirmed the bug.[2] Additionally, most communication and fault tolerance mechanisms are currently slow (see Hübner et al. [Hüb+21a] for details). We expect these issues in ULFM to be fixed once more resources are allocated to implementing these features. In our performance benchmarks, we thus use OpenMPI and simulate failures by removing processes from the calculation using `MPI_Comm_split` and replacing other required fault recovery steps by functionally similar ones (e.g., replacing `MPIX_Comm_agree` with `MPI_Barrier`s).

6.3 Acknowledgements

We gratefully acknowledge the Gauss Centre for Supercomputing e.V. (`www.gauss-centre.eu`) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (`www.lrz.de`).

[2] George Bosilca. Post `pbSToy94RhI/xUrFBx_1DAAJ` on the ULFM mailing list.

Chapter 7

7 Fault Tolerance for Distributed Processing Frameworks

Abstract. *Parallel processing frameworks like MapReduce, Spark, and Thrill help programmers scale their data analysis tasks to large machines connected by networks. They often also include functionality to recover from failures of a part of the computational resources. However, this usually comes with the drawback of high overheads during normal execution and/or long recovery times. We present a low-overhead technique for enabling fault tolerance in such frameworks that enables fast recovery after a failure. We experimentally evaluate a simplified variant of our technique that shows that the overhead during fault-free execution is below 4% for most benchmark algorithms.*

References and Attribution. The author of this thesis made the main contribution to the general technique presented in this chapter. Detailed algorithms for MapReduce and implementation techniques where developed with Lukas Hübner. The implementation and experiments where done by Charel Mercatoris as part of his master thesis [Mer21].

7.1 Introduction

Big Data processing frameworks like MapReduce [DG08], Spark [Zah+12], or Thrill [Bin+16] enable programmers to easily write programs that are executed in highly parallel environments without the need to implement parallelization themselves. They achieve this by restricting the programmer to a small set of operations which are parallelized by the framework developer. Conceptually the parallelization is usually implemented by splitting up the computation into rounds of communication between worker processes and local work in between these round. These frameworks usually also include features for being able to recover from node failures. However, they often come with drawbacks like having to use a distributed filesystem [DG08] or having to redo large parts of the computation [Zah+12]. In this chapter we present an approach to quickly recover from single node failures in the Thrill and MapReduce frameworks. We achieve this by locally storing all messages sent through the network on the sending and receiving side until the next round of global communication plus an additional (hopefully small) communication overhead for data that cannot be secured that way. We implement a simplified version of our approach for the more restricted MapReduce framework. Our experiments show

that the additional overhead during normal execution is small (under 4% for most benchmark algorithms) and enables reasonably fast recovery after a failure (about 30% of the running time of the original operation).

7.2 Related Work

Here we give an overview of Big Data processing frameworks and their fault tolerance techniques.

MapReduce [DG08] was originally developed by Google and inspired more complex tools like Spark and Thrill. Its most commonly used implementation is Apache Hadoop [Apa] with its fault tolerant file system *Hadoop Distributed File System* (HDFS) [Shv+10]. MapReduce processes data using only two functions: *Map* and *Reduce*. *Map* locally applies a user-defined function to every input item and outputs key-value-pairs. *Reduce* gathers all items grouped by their key and applies a user-defined reduction function which takes all items with the same key as input. Fault tolerance is achieved by storing the result of every *Reduce* operation to a fault tolerant file system and a master process that reschedules failed tasks. This has the benefit of only having to re-execute the last failed operation but comes at the cost of frequent reads and writes to the fault tolerant file system which forms a performance bottleneck. There are numerous extensions and alternative implementations of MapReduce, but to the best of our knowledge they all either do not support fault tolerance [PD11; Gao+17] or rely on a distributed fault tolerant file system [Con+10; Eka+10; Guo+15].

The dominant way of theoretically analyzing algorithms in the MapReduce framework is the *MapReduce Class* (MRC) complexity class and its corresponding computational model [KSV10]. This is, however, rather coarse grained as it only counts the number of MapReduce rounds required while constraining the user-defined functions rather loosely in their time and space complexity. Sanders [San20] gives a more detailed model for analyzing MapReduce algorithms using (among others) the bulk-synchronous parallel (BSP) model [Val90; McC95]: Let m be the total data volume, $\hat{m}$ the maximum size of an object produced by a user-defined function, and w and $\hat{w}$ the total and maximum local work for evaluating the user-defined functions, respectively. Then Sanders shows how to implement MapReduce in the BSP model using $\mathcal{O}(\hat{w}\hat{o}(\frac{w}{\hat{w}}, p))$ local work and $\mathcal{O}(\hat{m}\hat{o}(\frac{m}{\hat{m}}, p))$ communication volume, where $\hat{o}(b, p)$ is the expected maximum number of balls in a bin when randomly assigning $\lceil b \rceil$ balls to p bins. If $w/\hat{w}$ and $m/\hat{m}$ are in $\Omega(p \log p)$, this becomes $\mathcal{O}(\frac{w}{p})$ and $\mathcal{O}(\frac{m}{p})$, respectively. The same bounds can be achieved without large fractions $w/\hat{w}$ and $m/\hat{m}$ when using more sophisticated load balancing techniques.

Apache Spark [Zah+12] is a distributed processing framework with a rich set of instructions like sorting or database-like joins. It runs in the Java virtual machine and supports fault tolerance by tracking the *lineage* of their *Resilient Distributed Datasets* (RDDs): If a PE fails, the parts of an RDD that were stored on that

PE are lost. Spark then re-executes all previous operations leading to that RDD but only on the data required to compute the lost data. This has the advantage of having virtually no overhead during fault-free execution and works well if the lost data is only dependent on a small (contiguous) fraction of preceding RDDs, as for example in operations like mapping or filtering that require no communication. However, if large parts of a preceding RDD A are required to recompute the lost part of an RDD B, and A is no longer kept in memory, then Spark has to recompute the entire RDD A in order to recover a small part of B.

7.2.1 Thrill

We now describe the C++ framework Thrill [Bin+16] which we will use as a basis for our fault tolerance techniques. We also point out which parts of Thrill need special attention for fault tolerance. While our discussion will focus on Thrill, our techniques could similarly be applied to other frameworks. In fact, we begin by describing a simplified version for MapReduce in Section 7.3.

Thrill's *Distributed Immutable Arrays (DIAs)* are *ordered sequences* of data that can be transformed by Thrill's set of supported operations (like mapping, grouping, or sorting) to form new DIAs. DIAs are similar to Sparks RDDs with the difference that DIAs are always ordered, hence the name *array* in Thrill as opposed to *set* in Spark. The main operations can be distinguished into two categories: Local Operations (LOps) that require no communication between PEs and Distributed Operations (DOps) that require communication. In addition, there are sources that create a DIA from external sources, and actions that generate an output. Thrill works in a bulk-synchronous manner: The computation is split up into local computation phases and communication phases. In the local computation phase, all LOps between two DOps as well as the local parts of the surrounding DOps are evaluated in parallel on all PEs. Thrill optimizes this by compiling all local computations into one operation by heavy use of C++'s template meta-programming. Introducing fault tolerance in between LOps would require some kind of remote backup of the produced data resulting in large additional communication overhead. In addition, it would most likely break Thrill's optimized way of compiling all LOps into a single binary code. We thus focus on the communication phase of DOps for fault tolerance: Here, communication is required and we insert fault tolerance mechanisms at operations that induce large communication volumes. As communication is usually the bottleneck in large distributed applications, loosing progress in the local parts is acceptable as long as only little progress in communication-heavy parts is lost.

7.3 A Simple Version for MapReduce

In this section, we start with a simple version of our fault tolerance technique for MapReduce [DG08]. In Section 7.4 we extend this to the more powerful Thrill framework [Bin+16].

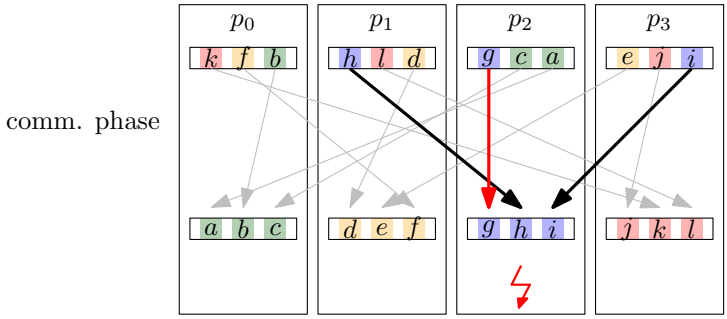


Figure 7.1: Data flow and stored messages for MapReduce. Colors indicate destinations of data elements. Arrows indicate messages sent during the communication phase of a *Reduce* operation. After failure of PE p_2, we can reconstruct the data held by p_2 using the messages sent by other PEs. Black arrows correspond to messages available on the sending PE. The red self-message is the only message not available on another PE and has to be additionally communicated for backup.

We now explain our technique to tolerate single PE failures. While this technique cannot tolerate multiple failures *at once*, it can be extended to tolerate failures of all machines powered by one power supply, an entire rack or any other predefined (non-intersecting) grouping of the PEs (Section 7.5.1). The idea is to create *recovery points* every time data is shuffled through the network in an all-to-all manner, which happens in every *Reduce* step. These recovery points are build in a way that enable fast recovery and balanced workloads after a failure.

In an all-to-all communication, we call the messages sent by a PE that are received by that same PE *self-messages*. We use the communication phase of each *Reduce* operation—which sends the majority of the data through the network—as *recovery point* by sending the self-messages of each PE p_i to a different PE p_j for backup. In the following we will use $b(i)$ to denote the PE that we send p_i's self-message to. This results in $\mathcal{O}(\frac{n}{p^2})$ additional bottleneck communication volume if all messages have the same size because each PE holds data of size $\mathcal{O}(\frac{n}{p})$ which is split up into p messages. Every PE then keeps all messages sent to other PEs for backup in memory (or spills them to local disks if the volatile memory is too small). Upon failure of PE p_i, all other processors send the data that they originally sent to p_i to $b(i)$, so that $b(i)$ has all the data that p_i received from the all-to-all communication. PE $b(i)$ can then recompute p_i's result by applying the reduce and map functions locally. See Figure 7.1 for an illustration. Note that this only works if the state of a processor is fully described by the data that is sent and global data that is the same among all processors. For pure MapReduce, as described in Section 7.2, this holds true.

When only using one PE $b(i)$ as backup for each PE p_i, then $b(i)$ hold twice as much data as all the other PEs after p_i failed which results in imbalances. Instead of sending the self-message to only one other PE, we divide each self-message into $p-1$ parts. Each part is then sent to a different processor, so we can do the recomputation in parallel on all remaining processors after a failure. This also helps with load balancing of the next *Map* operation when continuing normal computation on $p-1$ processors.

Data destinations in the *Reduce* phase are usually implemented by hashing each key and splitting up the range of hash values evenly between the PEs. During recovery, the range of hash-values previously sent to F is split up evenly among the surviving PEs and all messages sent to F (including the backed up self-message) are redistributed according to this new distribution. The newly received items are then merged into the existing data after applying the reduction function and the following *Map* step. At this point, the system is in a state from which it can continue normal operation on the remaining PEs.

Because only the messages of the last *Reduce* operation are required for recovery, we can delete all previously stored messages after the next *Reduce* step is successfully finished.

If the overhead for replicating the self-messages of every *Reduce* step is too large, some of them can also be changed to non-recovery-points by storing just the messages sent without backing up self-messages. Recovery then need to recompute the data lost starting at the last recovery point. The last *Reduce* step that *did* include a recovery point is recovered as before. Following *Reduce* steps without a recovery point use that recovered data as during normal execution: On each PE, the data previously sent to the failed PE F along with the recovered data is used to re-execute the operations. During the communication phase we have to discard any parts of the recovered data that lies outside of F's hash-range, i.e., not send it to the destination PE as it was already sent before the failure. Figure 7.2 illustrates this change. In fact, this could be taken to the extreme by never backing up self-messages. In that case, after a failure, the data originally obtained by F from the data source (or a checkpoint created in between) would have to be re-read (usually from a fault tolerant file system like HDFS) and all operations would have to be re-executed on the lost data. This would cause virtually zero running time overhead during fault-free execution but would substantially increase recovery times and the memory overhead for storing all messages sent to other PEs. It would also enable recovering from any number of simultaneous PE failures because all data of all failed PEs can be recovered.

7.3.1 Analysis

We now analyze the running time of our technique following the BSP-based analysis by Sanders [San20]. As Sanders considers all data produced by the *Map* step to be communicated, sending the self-message to the other PEs does not change the bottleneck communication volume asymptotically. So it stays at $\mathcal{O}(\hat{m}\hat{o}(\frac{m}{\hat{m}}, p))$,

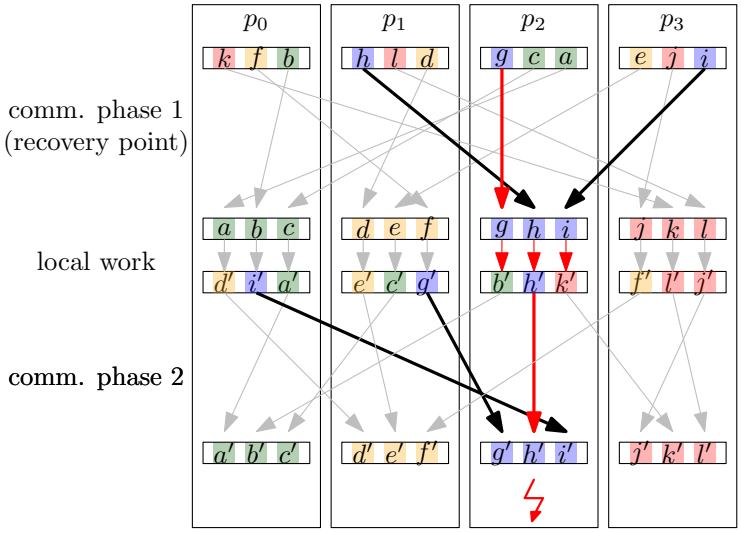


Figure 7.2: Messages lost when the communication phase of some *Reduce* operations is not used as a recovery point, i.e., self-messages are not backed up at other PEs. After failure of p_2 we need the data from all red arrows for recovery. Self-messages from communication phase 1 are backed up so we can recompute data elements b', h', and k' from the stored messages. These recomputed data elements in combination with the stored messages from the other PEs are used for recovery of communication phase 2. During recovery, we do not re-send data elements b' and k' because they were not sent to p_2 before.

where m is the total data volume, $\hat{m}$ is the maximum size of an object produced by a user-defined function, and $\hat{o}(b, p)$ is the expected maximum number of balls in a bin when randomly assigning $\lceil b \rceil$ balls to p bins. One addition to note here is that Sanders sends every item produced by a *Reduce* function to a random other PE to ensure load balancing and randomization of the input to the next *Map* step (actually, the first item produced by the *Reduce* for every key is kept on the same PE but this does not change the asymptotic communication volume in the analysis). Fault tolerance for this step can be implemented analogously to the communication phase before applying the *Reduce* function and the same analysis of the communication volume applies. Because during normal execution, the data is only sent and received, but not processed further, local work stays unchanged at $\mathcal{O}(\hat{w}\hat{o}(\frac{w}{\hat{w}}, p))$, where w and $\hat{w}$ are the total and maximum local work for evaluating the user-defined functions, respectively.

During recovery, the data previously held by the failed PE has to be re-processed. This PE received p out of the p^2 total messages sent during the last communication phase. Just as in [San20], a balls-into-bins argument thus gives us that each of these messages has size $\mathcal{O}(\hat{m}\hat{o}(\frac{m}{\hat{m}}, p^2))$. As each PE holds one of these messages

as well as a part of the self-message of the failed PE, which is in total of the same size, we have total bottleneck communication volume $\mathcal{O}(\hat{m}\hat{o}(\frac{m}{\hat{m}}, p^2))$. The same argument applies to the local work of the failed PE that has to be re-done by the surviving PEs, giving us $\mathcal{O}(\hat{w}\hat{o}(\frac{w}{\hat{w}}, p^2))$ local work.

In summary, the running time does not change asymptotically during normal execution and recovery takes approximately the time of processing a fraction $1/p$ of the data on p PEs.

7.4 The General Framework

We now explain how to extend the simple version of our technique to the more powerful Thrill framework [Bin+16]. As explained in Section 7.2, we will focus on DOps as LOps can be recomputed locally without any communication. Additionally, in contrast to MapReduce, where we expect every *Reduce* step to send the majority of the data through the network, Thrill has DOps that only need to communicate small parts of a DIA. Introducing a recovery point at these DOps would increase the communication volume substantially. We thus only introduce recovery points on *shuffling* DOps, which do send the majority of a DIA to other PEs.

Because Thrill supports operations like sorting or applying functions to sliding windows, DIAs are ordered, i.e., DIAs are ordered sequences where the item with index i logically stands before the item with index $i+1$. This is implemented by having an order of the PEs and having the data ordered on each PE. For our fault tolerance approach, we relax this requirement by allowing to divide the data further into ordered blocks. Every PE holds some of these blocks, but not necessarily blocks that are consecutive in their order. As handling multiple, possibly non-consecutive blocks comes with a performance penalty due to additional bookkeeping as well as communication overhead when requiring information from a neighboring DIA entry, every PE initially holds one block. Upon failure, the data held by the failed PE is split up into c blocks and ownership for each of these c blocks is assigned to a new PE. This means that after a failure, some PEs hold two blocks: their original block and a smaller block that holds a part of the data that was owned by the failed PE. An extreme case would be to set $c = p - 1$ (and adjusting it to the number of alive PEs minus one after every failure). However, this might lead to increased communication for non-shuffling DOps.

As this approach causes some performance penalties after a failure has occurred due to the data being split up into more than p blocks, we merge the blocks into one block per PE on each shuffling operation during normal operation (i.e., while not recovering from a failure). See Figure 7.3 for an illustration.

As for MapReduce, we discard the stored messages once they are no longer needed. Whenever a shuffling operation β is performed, the data backed up from the previous shuffling operation α can be discarded once the backup for β is complete. At this point we can always recover to operation β so no backup-capability for α is needed. An exception to this is when there are two DIAs B and C are derived from

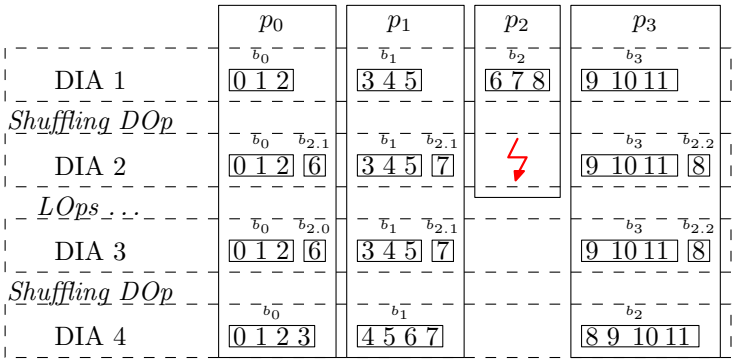


Figure 7.3: Illustration of our fault tolerance technique with 4 PEs and $c = 3$. Numbers indicate indexes within a DIA. DIAs are shown split up into blocks b_i distributed over the PEs p_i. DIA 1 is transformed into DIA 2 by a shuffling DOp. After completing that DOp, p_2 fails. All data previously located on p_2 is split up into $c = 3$ parts and distributed over the surviving PEs. When transforming DIA 2 into DIA 3 via LOps, these blocks are kept. During the next shuffling DOp, that transforms DIA 3 into DIA 4, the blocks are merged to obtain only one block per surviving PE.

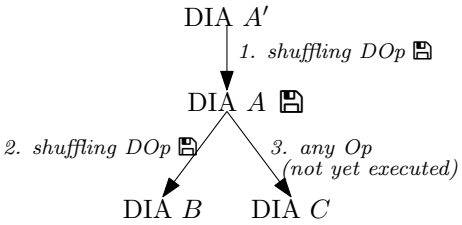


Figure 7.4: DIAs B and C are obtained from operations on DIA A. DIAs A', A, and B are already computed; DIA C has not been computed yet. In order to quickly compute C when it is needed, the framework decides to keep A in memory (shown by the floppy disk symbol). In order to enable having A in memory after a failure, we keep the self-messages of the shuffling DOp that created A from A' until the next shuffling DOp after C is executed. As DIA B has been obtained from A via a shuffling DOp, we also keep the self-messages of this operation.

the same parent DIA A. Any framework supporting this needs some mechanism to decide whether to keep A in memory after computing B but before C is computed. So whenever A is kept in memory, we keep the backup data for the last shuffling operation preceding A so that A can be recomputed after a failure. See Figure 7.4 for an example.

7.4.1 Details on Operations

We now show how every DOp of Thrill can be modified to support single PE failures using our technique. Note that we only consider DOps because LOps only require the data available locally and can thus easily be recomputed in parallel from the data recovered from the previous DOp.

We briefly explain every DOp from the Thrill paper [Bin+16] and the key aspects needed for fault tolerance. For this, we always assume that the operation used as a recovery point has been successfully completed once. If not, we discard its output and start recovery at the last recovery point that was fully completed. We can thus assume that all data on surviving PEs is correct. We call the PE that failed F and the c processors that hold the c new blocks after a failure $p_1, \ldots, p_c$.

ReduceByKey and GroupByKey both take a key extraction function and a combination function as input. The key extraction function is applied to every DIA entry to extract a key. Then all items with the same key are gathered and combined using the combination function. Here, *ReduceByKey* and *GroupByKey* differ slightly: In the case of *ReduceByKey*, the combination function is called a *reduction* function and specifies how to combine *two* items into one. This is done locally until only one item per key remains. After that all items with the same key are gathered on one PE and further reduced until only one item per key remains globally. *GroupByKey* is similar to MapReduce's *Reduce* function. Here, the combination function is called a *group* function and takes *all* items with the same key as input and produces one output, so here no local reduction is applied. Instead, all items with the same key are gathered on one PE and the group operation is run for each key. In addition, there is also a *ToIndex* variant of both of these functions where instead of a key extraction function, the user can specify the desired index in the output DIA. Both are implemented by passing the extracted keys through a hash function and splitting up the hash-range evenly between the PEs.

Because of the pseudo-random distribution of the data due to the hash-function, *Reduce* and *Group* are shuffling, i.e., they sent the majority of the data through the network. We utilize this for our fault tolerance approach by storing all messages sent during the communication phase as well as backing up self-messages at c different PEs. As mentioned above, we assume that the operation successfully completed once before a fault, so when a PE F dies, all others know F's hash-range which can then be further split up into c sub-ranges to send to PEs p_1 to p_c. All messages previously sent to F, including

the c self-message parts, are then split up and sent to the corresponding new PE. A further possible optimization is to already split up the c self-message parts correctly during normal operation so they don't need to be re-sent during recovery. By keeping all DIA entries ordered by their keys hash-value, this recovery does not change the logical order of the DIA.

We can also discard any blocks from previous recoveries here: When running a *Reduce* or *Group* operation during normal operation (i.e., while not recovering from a fault), we merge all blocks on each PE into one before executing the actual operation.

Sort uses a distributed SampleSort implementation to sort a DIA according to a user-defined comparison function. As this shuffles the data through the network (at least for unsorted inputs), we store all messages sent and store the c self-message parts at $p_1, \ldots, p_c$. After a failure, all PEs participate in a new sorting operation on the data originally sent to F (including the self-messages) with a change in the algorithm to send the data only to the c backup-blocks on $p_1, \ldots, p_c$.

Again, as for *Group* and *Reduce*, we can merge all blocks back into one block per PE before sorting.

Merge combines multiple sorted DIAs using a user-defined comparison function. This is implemented by first determining global splitter points by sampling all input DIAs on all PEs. After that, the DIA entries are sent to the target PE which is determined by the previously obtained splitter points. Depending on the distribution of the input DIAs, this is likely to only send small parts of the data through the network, so replicating self-messages would be costly here. Sent messages and the computed splitters still need to be stored for re-execution on data recovered by a previous shuffling operation, though. After a failure, a merge operation is executed on all messages sent to F as well as the recovered data from all input DIAs. However, the splitters from the previous execution are used to filter out any items outside of F's range that may be part of the recovered data of the input DIAs.

If a lot of data is shuffled due to unfavorable input data distribution, we can also back up self-messages in *Merge* and use it as a recovery point.

Concat takes multiple DIAs as input and concatenates them, preserving order. This is implemented by summing up the number of items in each input DIA and assigning an equal amount of items in the output DIA to each PE. As this will (except for corner cases with very distorted data distributions) shuffle most of the data through the network and many self-messages will even be empty, we can afford to back up self-messages (and of course keep sent messages in memory). Recovery after a failure then needs to assign the messages previously sent to F among p_1 to p_c analogously to the failure-free case.

During normal execution we can also discard any additional blocks from previous recoveries by only using one block per PE on the receiving side.

PrefixSum uses a user-defined associative function to compute a partial sum. This is implemented by adding up the local items and computing a global partial sum over these local sums. This is then used as initial value to compute the partial sum of the local elements. As this does not communicate any DIA items, we do not use it as a recovery point and only need to store the *right-most* local value of each PE (or block if there was a previous failure). During recovery, we only send the right-most value of PE $F-1$ to p_1 and use it as initial value for a new PrefixSum-computation on $p_1, \ldots, p_c$.

Zip combines multiple DIAs of equal size index-wise using a user-defined zip function. The item with index i of each input DIA is passed to the zip function which outputs the item at position i in the output DIA. This is implemented by using a global partial sum to determine the amount of items on each PE in the output DIA. The input DIAs are then redistributed according to this distribution. For most data distributions this requires only little communication volume, so we do not use *Zip* as a recovery point. Recovery is analogous to *Merge*: We zip the data sent to F as well as the recovered part of the DIAs on $p_1, \ldots, p_c$ while discarding any recovered data that is outside of F's bounds. For this, $p_1, \ldots p_c$ need the indices of the data they hold which can be obtained from the neighboring surviving PEs.

ZipWithIndex applies a user-defined function to each item of the input DIA together with its index in the DIA. The only communication here is a partial sum of the local sizes which is required to compute the global indices. Recovery is similar to *PartialSum*: We compute a partial sum of the local sizes of the recovered DIA parts with the index of the *left* neighbor item of the block stored on p_1 as starting value. We thus have to store the *right-most* index of each block during normal execution.

Window and FlatWindow take a window size k and apply a user-defined function on sliding windows of size k to the ordered elements of the DIA. *Window* outputs a single item per window, and *FlatWindow* can output an arbitrary number of items. This is implemented by sending the $k-1$ *right-most* items of each PE (or block) to the next PE (or block). If a PE (or block) holds less than $k-1$ items, it waits until it receives the remaining ones from its other *neighbor* PE (or block). As this operation does usually not send the majority of a DIA through the network, we do not create a recovery point here. We still need to ensure that it can be re-executed on data recovered from a previous shuffling operation, though. To ensure this, for each block, we store the $k-1$ items sent to the *right* neighbor block. After a failure, the c recovered blocks contain almost all data required to run a *Window* operation on the recovered data. Only F's *left* neighbor needs to re-send the $k-1$ missing items.

Backup can be introduced as a new operation to create a manual recovery point. Functionally, this is just the identity function but recovery can return to this operation instead of the last shuffling DOp preceding it. Internally this could be implemented by each PE simply backing-up its part of the DIA on c other PEs and sending any small additional blocks it holds from previous recoveries to another PE for back up. Alternatively, we could use some other checkpointing mechanism (for example using our tool ReStore presented in Chapter 8). Recovery then simply continues with the backed-up data on $p_1, \ldots, p_c$.

Asymptotically, the running time of the shuffling operations does not change *for randomly distributed inputs* because the self-messages are only a fraction of the size of the already communicated volume (on average a fraction $1/p$). During recovery, the running time of the recovered shuffling DOp is roughly that of running the operation on an input of size n/p on c PEs. If we assume the non-shuffling DOps to have communication volume independent of the input size (which is always true for *PrefixSum*, *ZipWithIndex*, and *(Flat)Window* for constant k; and true for evenly distributed inputs in *Zip* and *Merge*), then after a failure these require local work for an input of size $\frac{n}{p} + \frac{n}{pc}$, and communication volume as well as communication startup costs as if running on $p + c$ PEs.

As for MapReduce in Section 7.3, all operations described here as recovery points can also be changed to non-recovery-points by storing just the messages sent without backing up self-messages. Recovery is then analogous to *Merge* and *Zip* where the data previously sent to F along with the recovered DIA parts is used to re-execute the operation on the recovered data. Like in *Merge* and *Zip* we have to discard any parts of the recovered data that lies outside of F's output-range. This can reduce communication overhead in cases where only a small part of the data is shuffled through the networks, i.e., self-messages are large. For example, that could happen when sorting an already (almost) sorted DIA.

7.5 Supporting Multiple Failures

The raw technique presented in Sections 7.3 and 7.4 can only support a single failure between two shuffling DOps. For the case that more than one PE fails at a time, our algorithm does not work. Consider two PEs p_i, p_j failing at the same time. Even if p_i is not used to backup any self-messages of blocks on p_j (and vice-versa), the messages sent from p_i to p_j are lost and cannot be recovered. However, we can extend the technique to support failures of predefined groups of PEs like all CPUs on the same power supply (Section 7.5.1) and multiple single failures in between recovery points (Section 7.5.2).

7.5.1 Supporting Failure of Predefined Sets of PEs

While supporting arbitrary combinations of PEs failing at once is not supported by our technique, we can relatively easily extend it to support recovering from the failure of predefined non-intersecting sets of PEs. These sets could, for example, be all PEs located on the same multi-threaded CPU or those being powered by the same power supply which are more likely to fail at once than unrelated PEs [Bau+11]. By logically treating them as one PE and treating all messages sent between them as self-messages, the same technique described in Sections 7.3 and 7.4 still applies.

7.5.2 Supporting Multiple Single-PE Failures

We can extend our system to support another failure after a failure, if the system had enough time to finish recovery from the first failure. To do this, every PE also needs to store all incoming messages during DOps. After a fault of PE F, the messages received from F are sent to another PE which acts as F in case of another failure. Every PE that sent part of its self-message to F also sends that part to another PE for backup. Additionally, $p_1, \dots p_c$ send the self-messages that they backed up for F to another PE (without splitting them up further) that will be the new backup PE for these blocks. During recovery, when the operations are re-executed on the recovered data, every block also stores the messages that it *would* have sent during normal execution. So if another failure happens (say, of PE F') before the next shuffling DOp, these stored messages can be used to run the recovery operations described in Section 7.4.1 on the c blocks recovered from F's original self-messages as well as any blocks that F' held from previous failures.

7.6 Experiments

7.6.1 Experimental Setup

We implement a simpler version of our fault tolerance techniques for MapReduce from Section 7.3 with support for multi-PE-failures as described in Section 7.5.1 in a prototype MapReduce-Implementation based on the Message Passing Interface (MPI). The implementation is configured to recover from the failure of an entire compute node at once, i.e., all messages sent between processes on the same compute node are treated as self-messages. In contrast to the theoretical description above, we only send the self-messages of each PE to one other PE instead of splitting it up among the other $p-1$ PEs. This does not change the bottleneck communication volume but leads to some imbalance during recovery. We also use the same hash function for every *Reduce* step and do not distribute the outputs of the *Reduce* step as used for the analysis. This can help utilize locality in benchmark algorithms but can in turn lead to larger self-messages. A detailed description of the implementation can be found in Charel Mercatoris' master thesis [Mer21].

Experiments were run on the SuperMUC-NG cluster with OpenMPI as described in Section 6.2. Our code is written in C++ and compiled using GCC 8.4.0 with full optimizations (`-O3`).

In our failure simulation experiments we simulate the failure of 10% of the nodes used, always failing one node (48 PEs) at a time. These failures are distributed uniformly across the MapReduce iterations (consisting of one *Map* and one *Reduce* step). Failures are simulated during the message exchange phase of the *Reduce* step which is also the point where any failures would be noticed in a real failure setting. We use the average running time over five repetitions of the same configuration with different random seeds.

7.6.2 Experimental Results

We evaluate the overhead caused by our fault tolerance technique for four MapReduce benchmark algorithms:

Word Count is a popular benchmark algorithm for MapReduce-implementations. Its input is a text which is split up into words in the *Map* phase. In the *Reduce* phase it gathers each occurrence of a word and counts the number of times it occurs in the text. This is not an iterative algorithm, so there are no iterations to insert faults into. It is still shown here due to its popularity. As an input we use 2 GB of text per node generated by picking words uniformly at random from the 4 436 574 distinct words in Project Gutenberg [Gut].

R-Mat, or Recursive Matrix Model [CZF04], is a random graph generator. Given a number of vertices and edges, R-MAT generates each edge based on a probability distribution in the adjacency matrix. It is commonly used to generate graphs with a community structure and is used in the Graph 500 benchmark [Mur+10]. We use a MapReduce implementation of the R-Mat generator introduced by Plimpton and Devine [PD11]. The input is a set of edges produced by the probability distribution that may contain duplicate edges. The *Map* phase has no functional role here. In the *Reduce* phase, we collect all edges with the same two endpoints and generate new ones for every duplicate. This is repeated until all edges are unique. We generate graphs with 2^{18} vertices per node and an average degree of 30 using the same randomization parameters used in the Graph 500 benchmark.

Connected Components are maximal connected subgraphs and a basic tool for graph analysis and a building block of many graph algorithms. We implement an algorithm by Kiveris et al. [Kiv+14] which consists of two phases that are repeated until convergence. Both phases output edges in a specific vertex order (i.e., directing the edge from vertex u to v or from v to u) during the *Map* phase and then group edges by the first endpoint during the *Reduce* phase where they are redirected to their connected components representative. This is repeated until convergence. We compute connected components of random graphs [ER60] with 2^{18} vertices per compute node and an average degree of 0.25. This leads to graphs with many small connected components [ER60].

PageRank is a network analysis algorithm developed by Google [Pag+99] and is also commonly used as a benchmark algorithm for parallel processing frameworks. Each vertex of a graph starts with the same score with the sum of scores adding up to 1. The algorithm then works in rounds where each vertex's current score is split up and transferred in equal parts to each neighbor. Additionally, each vertex gets the same small amount of score every round. These two components are always normalized to keep the global sum at 1. We implement this in MapReduce by outputting for each neighbor of a vertex, the score transferred to that neighbor in the *Map* phase. In the *Reduce* phase, the scores sent to each vertex are gathered, summed and the constant score is added. In order to keep all required information across rounds, we additionally emit the neighborhood of each vertex in the *Map* phase. This neighborhood is then also output during the *Reduce* phase together with the score. We run the PageRank algorithm for 100 iterations on random graphs [ER60] with 2^{18} vertices per node and an average degree of 38.

Figure 7.5 shows the overhead caused by our fault tolerance technique. "Fault Tolerance Overhead" shows the overhead caused in a fault-free scenario, i.e., the running time when enabling our fault tolerance technique (but without any failures occurring) relative to the running time without fault tolerance enabled. "Recovery Overhead" shows the time taken to recover from a failure relative to the time of a single MapReduce round.

After an initial phase for low PE counts where a large fraction of the messages are self-messages, the overhead for enabling fault tolerance reduces substantially. For $6,144$ PEs enabling fault tolerance has an overhead of 2% for Word Count, 3% for Page Rank, 4% for Connected Components and 29% for R-MAT. For all benchmarks except for R-MAT this is already reasonably low but could be improved further by not backing up self-messages in every round at the cost of a slower recovery. For R-MAT the overhead of almost 30% would be deemed impractical. This is due to the large fraction of self-messages: Because the same hash function is used in every iteration, all non-duplicate edges are always (except in the first MapReduce round) in the self-message. This can also be seen by the recovery taking about the same amount of time as a regular (fault-free) round. Because the entire self-message is stored on one compute node, and the self-messages contain the majority of the lost data, this node forms a major bottleneck of the recovery. Recovery performance is further reduced here due the communication phase dominating the running time of the R-MAT recovery because the *Map* and *Reduce* phase essentially only forward their inputs (at least for non-duplicate edges in the case of the *Reduce* phase). This could be alleviated by implementing splitting up the self-message across multiple PEs on different compute nodes leading to a lower bottleneck communication volume. For the other benchmark algorithms recovery takes about 30% of the time of a normal iteration. While this is already substantially faster than re-running the entire round, as a traditional checkpoint-restart scheme would do, it is hindered from better performance by a poorly scaling communication phase. Communication is implemented using the `MPI_Alltoallv` function which has been shown to have poor scalability [Sch13; SU23; SS23]. Improving this would directly improve the

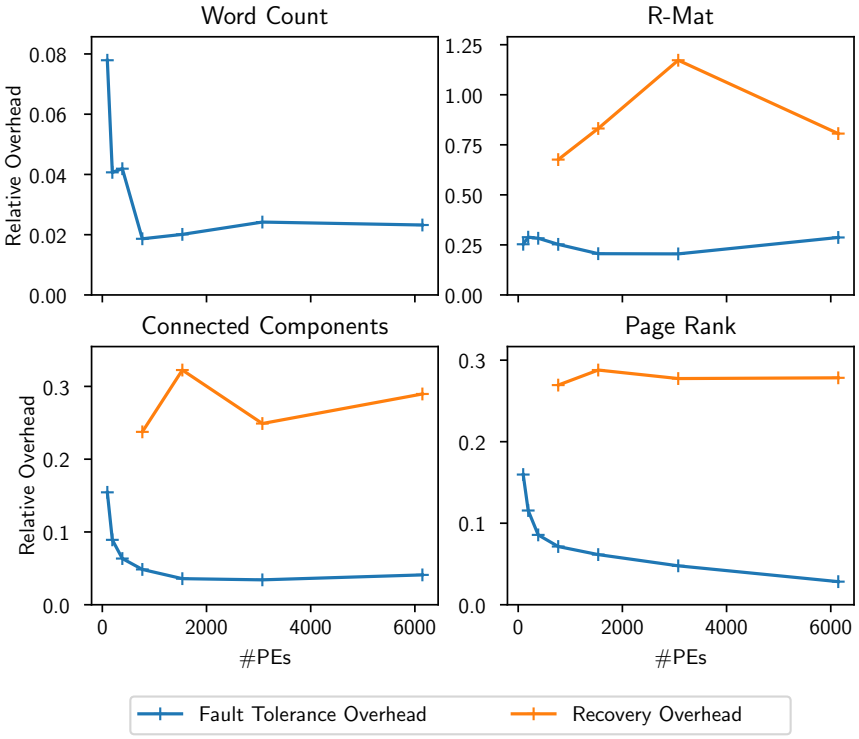


Figure 7.5: Overhead of fault tolerance and failure recovery overhead for different MapReduce benchmark algorithms.

performance and scalability of recovery. Additionally, the node storing the self-message holds on average twice the amount of data (and for these benchmark problems usually even more) causing an imbalance in the amount of data sent per PE during recovery. Splitting up self-messages as described in Sections 7.4 or 7.3 would distribute the load evenly.

7.7 Conclusion and Future Work

We present a framework for low-overhead fault tolerance and fast recovery in a powerful general purpose distributed processing framework. We then adapted this to the simpler MapReduce framework and showed that our techniques cause only low overhead for most MapReduce benchmark algorithms in practice. In the future, we would like to extend our implementation to enable a deeper experimental analysis of our technique. Apart from the obvious long-term goal of implementing fault tolerance for the full set of Thrill's operations, there are other extensions to be

implemented for more efficient fault tolerance in our MapReduce implementation that we already cover in Section 7.3: Creating recovery points less frequently could lower the overhead during fault-free execution, and splitting up self-messages during normal execution would improve load balance during recovery. Many MapReduce algorithms also use additional features often supported by MapReduce implementations. For example, static data that does not change across MapReduce rounds can speed up algorithms by not sending redundant data through the network. This static data could be recovered after a failure using ReStore, the software presented in Chapter 8. ReStore could also be used for the zero-overhead variant briefly explained at the end of Section 7.3. Apart from improvements to the fault tolerance of our implementation, the base MapReduce implementation itself could be extended and improved by adding load balancing (for example the randomized or work-stealing-based versions by Sanders [San20]), local aggregation for reduction functions that allow it, improved all-to-all communication (for example using the 1-factor algorithm [ST02; Sch13] or grid-based approaches [SS23; SU23]), and a hybrid parallelization for better utilization of the multi-core machines in modern HPC clusters.

8 Fast General Purpose Data Recovery

Abstract. *Fault tolerant distributed applications require mechanisms to recover data lost via a process failure. On modern cluster systems it is typically impractical to request replacement resources after such a failure. Therefore, applications have to continue working with the remaining resources. This requires redistributing the workload and that the non-failed processes reload data. We present an algorithmic framework and its C++ library implementation ReStore for MPI programs that enables recovery of data after process failures. By storing all required data in memory via appropriate data distribution and replication, recovery is substantially faster than with standard checkpointing schemes that rely on a parallel file system. As the application developer can specify which data to load, we also support shrinking recovery instead of recovery using spare compute nodes. We evaluate ReStore in both controlled, isolated environments and real applications. Our experiments show loading times of lost input data in the range of milliseconds on up to 24 576 processors and a substantial speedup of the recovery time for the fault tolerant version of a widely used bioinformatics application.*

References and Attribution. The following chapter is based on the conference paper [Hüb+22]. Together with Lukas Hübner, the author of this thesis is a main author of this paper with editing done by Peter Sanders and Alexandros Stamatakis. The design of the data distribution and algorithms was developed in close collaboration with Lukas Hübner. The implementation was done by the author of this thesis and Lukas Hübner where the author had a focus on the recovery after a failure and Lukas Hübner had a focus on storing the data before a failure. The experiments were done by Lukas Hübner. Large parts of this chapter were copied verbatim from the conference paper or the corresponding technical report [Hes+22].

8.1 Introduction

To recover from a failure in a distributed application, an important step is to restore the lost data that the failed processors were working on. To save the current state of a program's data, applications write *checkpoints* which can be reloaded after a process failure. Checkpointing libraries usually write their checkpoints to a parallel file system (PFS) [Aga+04; Bau+11; Sha+19; Nic+19], implying slow

recovery due to low disk access speeds and because many processors simultaneously access the same resources. Many checkpointing libraries also assume failures to leave the machine in a state where the process can simply be started again, or they assume that enough spare resources are kept idle to start a new process for replacing the failed one [Aga+04; Bau+11; Sha+19; Nic+19; TH14; Moo+10; BH14; Gam+14; Lu05; Bar+17]. Under this assumption, a re-spawned process can simply read exactly the data of the failed process. In the case of the new process being located on the same compute node as the failed one, the checkpoint can even be read from a local disk. To the best of our knowledge there exists no general purpose checkpointing solution that allows for in-memory recovery without requiring spare resources.

Contribution and Structure.

We introduce ReStore, an in-memory checkpointing library that is optimized for recovery speed (in contrast to checkpoint creation speed). This is especially important for data which the program never or rarely changes but has to be redistributed after every failure. We do not assume that spare resources are available. Instead, ReStore enables recovery in an application that continues its execution only with the processes that are still alive. While this approach requires a more involved recovery mechanism and strategic data distribution, it saves resources because all available processors can participate in the application's useful computations from the beginning. Keeping the checkpoints in-memory avoids the bottlenecks involved in a PFS and allows high scalability.

The remainder of this paper is structured as follows: We provide an overview of existing checkpointing libraries and other related work in Section 8.2. We explain our general framework and data distribution in Section 8.3. In Section 8.4 and 8.5 we present the implementation of our open source C++ library and the experimental results, respectively. We conclude in Section 8.6 and outline future work.

8.2 Related Work

Scientific applications are increasingly implemented to tolerate faults. Examples include a numeric linear equation and partial equation solver [Ali+16], a plasma simulation [Obe+17], a molecular dynamics simulations [Lag+16], a Fast Fourier Transformation [EG03], and an algorithm for phylogenetic inference [Hüb+21a]. The three main techniques for implementing fault tolerant algorithms are Algorithm-Based Fault Tolerance [VM97; Bos+08], restarting failed sub-jobs [Mem+16], and checkpointing/restart [Koh+19; Hüb+21a]. Checkpointing/restart can be further subdivided into coordinated and uncoordinated checkpointing. In coordinated checkpointing, the program synchronizes before creating the checkpoint in a distributed manner. This ensures that there are no messages in-flight and the program's state is therefore well-defined. Gavaskar and Subbarao recommend coordinated

checkpointing for the high-bandwidth, low-latency interconnections of modern HPC systems [GS13]. Checkpointing libraries can save their checkpoint either to a (possibly network attached) disk or to the compute node's main memory ("diskless") [PLP98]. Checkpointing libraries which save their checkpoints to disk include, for example, the algorithm presented by Agarwal et al. [Aga+04], FTI [Bau+11], CRAFT [Sha+19], SCR [Moo+10], and VeloC [Nic+19]. As the number of nodes per parallel program execution continues to grow, the congestion on the PFS increases—resulting in a bottleneck and reduced checkpointing performance [Gos+21; Hér+19]. Examples for in-memory checkpointing libraries include ftRMA [BH14], Fenix [Gam+14], GPI_CP [Bar+17], and the algorithm described by Lu [Lu05] (Table 8.1). All of these employ the substitute strategy and therefore rely on the availability of replacement nodes, if we want to continue the computation in case of node failure. This implies that some nodes are allocated to the job but not available for computation. Some algorithms additionally designate some compute nodes as pure checkpointing nodes, which are neither participating in the computation nor available as spares. Ashraf et al. [AHE18] describe an implementation of a fault tolerance mechanism for a specific application which is able to checkpoint to memory and recover in a shrinking setting. This is, however, not a general-purpose checkpointing library but application-specific. Erasure codes are often used to reduce the file-size or memory footprint of checkpoints [Lu05; BH14; Bau+11].

Other areas where replication approaches similar to the one presented in this chapter are used are distributed fault tolerant file systems like early versions of the Hadoop Distributed File System [Shv+10] or distributed processing frameworks like Apache Spark [Zah+12]. However, these target very different use cases and sometimes only support very basic replication like storing each PEs data on a single partner PE.

8.2.1 Reproducibility Study

In the following, we describe our attempts to replicate the results of competing tools. We provide a visual summary of these in Table 8.1.

The ftRMA [BH14] tool has not been maintained since 2014 and relies on the Cray-only foMPI library which has also not been further maintained since 2014. The authors confirmed (pers. comm. 30. June 2022) that the current code exclusively works on Cray systems and is no longer being actively maintained. Although the authors suggested that ftRMA could—in principle—be ported to a non-Cray system, taking into account the unmaintained code base comprising 513 calls to foMPI functions, this would incur a prohibitive programming effort with uncertain outcomes. Further, as ULFM currently provides "little support for fault tolerance" with respect to RMA calls [Bou19], deploying ftRMA would be bound to fail using a current fault tolerant MPI implementation.

With respect to the Fenix tool, there has only been a single commit to its repository in 2022. In addition, the author's automated testing on GitHub failed for

Table 8.1: Comparison of checkpointing libraries. See Section 8.2 for details. [1]The program needs to allocate spare nodes, which participate in the computation only in case of a failure. [2]The program needs to allocate spare nodes and nodes used purely for checkpointing. [3]The maintenance state is unclear (Section 8.2.1).

	ftRMA [BH14]	**Fenix** [Gam+14]	**SCR** [Moo+10]	**Lu** [Lu05]	**GPI_CP** [Bar+17]	***ReStore*** **this paper**
Features						
in-memory	✓	✓	✗	✓	✓	✓
substituting rec.	✓	✓	✓	✓	✓	✓
shrinking rec.	✗	✗	✗	✗	✗	✓
all nodes computing	✗[2]	(✓)[1]	(✓)[1]	✗[2]	(✓)[1]	✓
framework	MPI RDMA	MPI	MPI	MPI	PGAS/GPI	MPI
Reproducibility						
source-code avail.	✓	✓	✓	✗	✓	✓
maintained (2022)	✗	?[3]	✓	✗	✗	✓
other issues	Cray-only	author-provided examples segfault			requires libiverbs, GPI-2, password-less ssh-login on all compute nodes	

this commit. We thus denote Fenix's maintenance status as being "unclear" ? in Table 8.1. The author-provided Fenix examples [Gam+14] fail with a segmentation fault. As Fenix does currently not support restoring the data that was checkpointed on a different rank, setting up an experimental comparison is challenging. The authors did not respond to our e-mail requesting assistance.

SCR [Moo+10] has frequent commits to its github repository. Hence, we consider that it is still being maintained. SCR supports *caching* checkpoints on a RAM-disk. These checkpoints, however, have to be transferred to the parallel file system such as to become available upon rank failure. We therefore do not consider SCR to be an in-memory checkpointing library in the context of node failures.

The source code of Lu [Lu05] is not available, and the author can not be contacted, as they did not provide a contact e-mail address on their publications.

The git repository of GPI_CP [Bar+17] has only a single commit from seven years ago; we thus also consider that it is no longer being maintained. As we do not have access to an HPC system where GPI-2 (a library for the Partitioned Global Address Space (PGAS) programming model) is supported, and its dependency `libiverbs` has to be installed by a system administrator, we are unable to compare GPI_CP against ReStore.

8.3 In-Memory Replica for Fast Recovery

ReStore allows application developers to store redundant copies of their data in-memory. In case of a failure, the surviving PEs can invoke a recovery routine to load all or parts of the data lost during the failure.

	PE 0				PE 1				PE 2				PE 3			
copy 1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
copy 2	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7

Figure 8.1: Example showing the data distribution of the copies stored with ReStore for $p = 4$ PEs, $n = 16$ data blocks, and $r = 2$ copies.

The remainder of this section is structured as follows: In Section 8.3.1 we introduce our general framework for maintaining redundant copies of the user-supplied data in memory as well as the algorithm used for recovery. Section 8.3.2 expands on the distribution of copies by adding random permutations that accelerate the recovery algorithm. We analyze the memory usage of our proposed data distribution in Section 8.3.3 and the probability of irrecoverable data loss in Section 8.3.4. In Section 8.3.5 we describe a yet unimplemented approach to restore the level of redundancy after a failure.

8.3.1 General Framework

The main idea of ReStore is to store r copies of the data on different nodes. By storing them such that it is unlikely for all copies of one data element to fail at once, there will most likely (Section 8.3.4 and Section 8.5.2.a)) be copies left to recover from. The application programmer can store data into ReStore using the *submit* function and retrieve data from ReStore using the *load* function. To make the data addressable, we divide it into blocks where each block has a unique identifier.

Let n be the number of data blocks. In its most basic form (Figure 8.1) we store the block with ID x on PEs $L(x,k) = \lfloor \frac{xp}{n} \rfloor + k \cdot \frac{p}{r} \mod p$, for $k \in [0,r)$. Under this distribution, we expect the copies of a block to not be stored on the same physical node/case/rack in most cluster setups. This decreases the probability of loosing all copies of a block, as failures of PEs in the same node/case/rack are more likely to occur than a simultaneous failure of unrelated PEs [Bau+11]. In Section 8.3.2 we explore a change to this basic distribution scheme that allows for faster recovery.

During recovery, when PE i requests to load block j, we choose one of the surviving PEs that hold block j at random to serve the request. If a PE requests multiple successive blocks which are stored on the same set of PEs, we choose one PE to serve all of them. This strategy minimizes the bottleneck number of messages received. Next, we distribute the requested data using a custom sparse all-to-all communication.

8.3.2 Breaking Up Access Patterns for Faster Recovery

The goal of distributing data copies as described in Section 8.3.1 is primarily to preserve the ability to recover from a fault. In the following, we explore how to

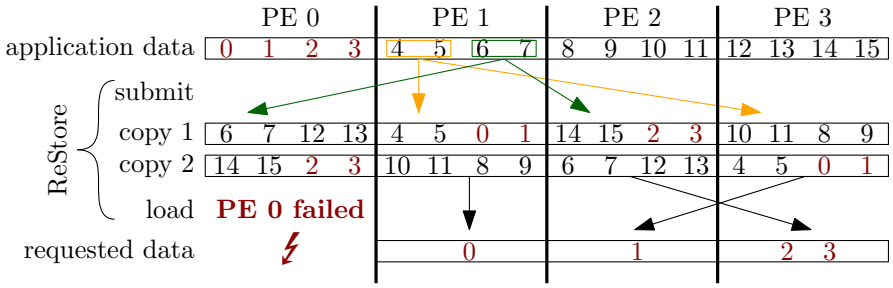


Figure 8.2: Example showing the submit and load operations as well as the data distribution of the copies with a random permutation for $p = 4$ PEs, $n = 16$ data blocks, $r = 2$ copies, and $s_{pr} = 2$ blocks per permutation range. The first row shows the data submitted by the application. As an example, the orange and green arrows show the data ReStore sends from PE 1 to the target PEs which hold copies of the received data. After PE 0 fails, the application requests the data shown in the last row (dark red in all occurrences) which is served by ReStore as shown with the black arrows.

adapt the data distribution such that it accelerates data recovery while preserving the level of failure resilience.

Assume a failed PE i which worked on data blocks $\left[i\frac{n}{p}, (i+1)\frac{n}{p}\right)$, where n is the total number of data blocks submitted to ReStore. If the application were to redistribute the lost data evenly to all surviving PEs, we would ideally want a dedicated sending PE for each receiver.This would result in a bottleneck communication volume of $\frac{n}{p^2}$ and a bottleneck number of messages received of 1. With the data distribution from Section 8.3.1, only the surviving subset of the $r \ll p$ PEs that hold copies of these blocks act as sources, resulting in a bottleneck communication volume of $\frac{n}{pr}$. We can alleviate this issue by evenly distributing the copies of data blocks $\left[i\frac{n}{p}, (i+1)\frac{n}{p}\right)$ among multiple PEs by randomly permuting the block identifiers. For a random permutation π and $k \in [0, r)$, the block x is stored on PEs

$$L(x,k) = \left\lfloor \frac{\pi(x) \cdot p}{n} \right\rfloor + k \cdot \frac{p}{r} \mod p$$

If the user requests blocks $\left[i\frac{n}{p}, (i+1)\frac{n}{p}\right)$, more PEs have parts of the data and can send them to the requesting PE. This approach, however, can lead to a large bottleneck number of messages being sent and received: If a PE requests $\frac{n}{p^2}$ data blocks, these blocks can reside on up to $\min(\frac{n}{p^2}, p)$ different PEs. To mitigate this, we group the data into *permutation ranges* of size s_{pr}. We then apply a random permutation to these permutation ranges instead of individual data blocks (Figure 8.2). If by a fault on PE i, data blocks $\left[i\frac{n}{p}, (i+1)\frac{n}{p}\right)$ need to be redistributed, these correspond to permutation ranges $[\frac{i \cdot n/p}{s_{pr}}, \frac{(i+1) \cdot n/p}{s_{pr}})$. If we request the data to

be evenly distributed among the $p-1$ surviving PEs such that PE j receives blocks $[i\frac{n}{p} + j\frac{n}{p\cdot(p-1)}, i\frac{n}{p} + (j+1)\frac{n}{p\cdot(p-1)})$, only $\frac{n/(p\cdot(p-1))}{s_{pr}}$ PEs send to every receiving PE.

The best choice of s_{pr} and whether to use permutations at all depends on the data distribution, the expected amount of data lost by a fault, and the application's recovery strategy, but also on the frequency of checkpoint creation since submitting data with permutations enabled results in a more dense communication pattern. Section 8.5.2.a) shows how we experimentally chose a good value for s_{pr}.

Note that with this data distribution, we always have sets of $\frac{n}{s_{pr}p}$ permutation ranges that are stored together for all r copies (e.g., blocks $6, 7, 12, 13$ in Figure 8.2 are always stored together—once on PE 0 and once on PE 2). This means that for all copies of any permutation range whose first copy is stored on PE i to become lost, exactly the set of r PEs $i + k \cdot \frac{p}{r}, k \in [0, r)$ has to fail. One could also opt for a different approach—for example, using a distinct permutation for each copy. In this case, no sets of permutation ranges will always be stored together. So in order for any permutation range whose first copy resides on PE i to be lost, it is sufficient if *any* of the $\frac{n}{s_{pr}p}$ sets of PEs of size r fail that hold the copies of one of the permutation ranges. In Section 8.3.4 we analyze the probability of irrecoverable data loss under our proposed data distribution.

8.3.3 Memory Usage

Other fault tolerance libraries [Lu05; BH14; Bau+11] often use erasure coding—for example the Reed-Solomon code [RS60]—to reduce their memory footprint. This works for example by not storing the replicas A' and B' of two blocks A and B but rather the XOR of these blocks $A \oplus B$. We decide against using erasure coding as a means to reduce the memory footprint of our checkpoints, as this would incur additional messages upon checkpoint creation and recovery as well as a substantial computational overhead [CD96]. We therefore trade reduced communication overhead for increased memory consumption.

As in Section 8.3.1, let n be the number of data blocks, r be the number of replicas and p be the number of processes. On each PE ReStore requires main memory to store $\frac{rn}{p}$ data blocks for the replicated storage. The memory requirement is doubled during submission as we require additional space for the send and receive buffers. During recovery, an additional copy of all data being sent and received is stored on each PE. We verified these measures empirically (data not shown). A plethora of applications exist for which the amount of memory for the input data *and* the data that need to be checkpointed fit in memory r times. Examples include RAxML-NG [Koz+19; Hüb+21a], k-means, and page-rank.[1] For example, RAxML-NG is memory bandwidth bound [Koz18]. Hence, using additional cores with their associated larger cache memory capacity can even yield super-linear speedups due to increased cache-efficiency. Such applications can therefore substantially benefit

[1] We implemented fault tolerant version for all three of these using ReStore and show running times for RAxML-NG and k-means in Section 8.5.3.

from reduced communication and computational overhead to create and restore a checkpoint without erasure codes.

8.3.4 Probability of Irrecoverable Data Loss.

Let r be the replication level and p be the number of PEs. In this analysis, we assume that $r|p$ (r divides p). This constitutes a reasonable assumption for current two socket systems that exhibit an even number of cores per socket and $r = 4$. If $r|p$, the PEs are divided into $g = \frac{p}{r}$ groups, with all PEs in a respective group storing the same data. Thus, if and only if all r PEs in a specific group fail, we will not be able to recover a part of the data. We denote such an event as Irrecoverable Data Loss (IDL). Let f be the number of failed PEs. There is exactly one possibility to draw r out of r PEs belonging to a single group. The number of possibilities to draw the remaining $f - r$ failed PEs among the remaining PEs such that they do *not* belong to the given group is $\binom{p-r}{f-r}$. The overall number of possibilities to draw f PEs from the p PEs that are still alive at program start is $\binom{p}{f}$. The probability that, given f failures, all processes of a *given* group fail is thus $1 \cdot \binom{p-r}{f-r} / \binom{p}{f}$. When generalizing this equation to the probability of all processors of *at least one* group failing, we have to apply the inclusion-exclusion principle to avoid counting the same combination multiple times. We thus obtain the following equation for the probability of an IDL at failure f or any failure before:

$$P^{\leq}_{\mathrm{IDL}}(f) = \sum_{j=1}^{g} (-1)^{j+1} \binom{g}{j} \frac{\binom{p-jr}{f-jr}}{\binom{p}{f}}$$

The probability of an IDL at exactly failure f is thus:

$$P^{=}_{\mathrm{IDL}}(f) = P^{\leq}_{\mathrm{IDL}}(f) - P^{\leq}_{\mathrm{IDL}}(f-1)$$

The expected number of failures until an IDL occurs is:

$$E[\text{Failures until IDL}] = \sum_{f=r}^{p} P^{=}_{\mathrm{IDL}}(f) \cdot f$$

For small f, the *approximate* probability of all PEs of any group failing is given by $P^{\text{approx.}}_{\mathrm{IDL}}(f) = g \cdot (f/p)^r$. Setting $P^{\text{approx.}}_{\mathrm{ID}} = 1$ and solving for the fraction of PEs that fail f/p yields $f/p = (r/p)^{(1/r)} \in \mathcal{O}(p^{-1/r})$ for a fixed r.

In Section 8.5.2.a) we simulate node failures using the actual data distribution to verify these formulas.

8.3.5 Recovering Lost Replicas After a Node Failure

To further increase the resilience of our framework, we introduce an approach to restore replicas that were lost upon a failure while keeping all other replicas in place.

That is, we do not need to redistribute any replicas that reside on surviving nodes. As in the previous sections, let n be the number of data blocks, r be the number of replicas per block, and p be the number of nodes.

We draw a different random permutation ρ_x of $[0, p-1]$ (or long, non-repeating random sequences of nodes) for each block x and place the replicas of x on the first r alive nodes of that permutation. When a node dies, we copy all replicas that this node held to the next node in each replicas permutation. We can refine this approach to attain a perfectly balanced initial data distribution and reduce the probability of an IDL (Section 8.3.2): We initially place the first r replicas $(L(x,0), L(x,1), \ldots, L(x,r-1))$ deterministically as described in Section 8.3.1. That is, the data distribution is given by

$$L(x,k) = \begin{cases} \text{as described in Section 8.3.1,} & \text{if } k < r \\ \rho_x(k), & \text{else} \end{cases}$$

Following this data distribution, we can compute the ranks on which we store a given block in $\mathcal{O}(r+f)$ time and $\mathcal{O}(1)$ space where r is the number of replicas per block and f is the number of node failures. In order to keep recovery fast, we can apply this technique on a permutation-range-level rather than on individual blocks as explained in Section 8.3.2.

8.4 Implementation

We implement ReStore as a C++ library[2] using the User Level Failure Mitigation (ULFM) proposal implementation [Bla+13]. Application programmers submit their data to ReStore by writing their serialized data blocks to a memory location supplied by the library or using ReStore's interface for already serialized data. After a failure, they can request data blocks by passing a list of ranges of block identifiers to ReStore. This can be done in two ways: Either by providing the full list of requested block IDs on all PEs or by providing exactly those ID ranges each individual PE needs on exactly that PE. By using the first approach, no communication is required to determine which PE serves which request. When using the second approach, the receiving PE will determine which PE should send each requested data block. Then, a sparse all-to-all communication is performed to issue the requests to the sending PEs. Preliminary experiments showed that the latter method performs substantially better because the full list of requests usually scales with the number of PEs in the application, slowing down the first approach. Thus, for all experiments in Section 8.5, we employ the second approach. As ReStore's implementation currently focuses on fast recovery, it provides only an interface for submitting data once and thus is currently not suitable for repeatedly checkpointing changing data. This is sufficient for many fault tolerant applications—two of which are demonstrated in Section 8.5.3.

[2] `https://github.com/ReStoreCpp/ReStore`

8.5 Experimental Evaluation

In this section we present the results of our experimental evaluation. We present the experimental environment in Section 8.5.1. In Section 8.5.2 we evaluate ReStore's fault resilience and performance in isolation. In Section 8.5.3 we show how ReStore performs when used in two fault tolerant applications: A simple k-means algorithm and a complex bioinformatics application used by thousands of researchers. Finally, in Section 8.5.4 we compare ReStore to reading from a parallel file system (PFS)—which represents a lower bound for checkpointing libraries using the PFS as storage—as well as the reported running times by other checkpointing libraries.

8.5.1 Environment and Experimental Setup

We run our experiments on the SuperMUC-NG cluster using OpenMPI as described in Section 6.2. We compile our benchmark applications using gcc version 10.2.0 with full optimizations enabled (`-O3`) and all assertions disabled. As ReStore currently only supports submitting data once, all experiments shown in this section submit only their input data. This is a restriction in the API, not the underlying algorithm, and will be removed in future work.

All plots show results for 10 repetitions per experiment. Plots depict the mean with error bars for the 10th and 90th percentile.

8.5.2 Isolated Evaluation

In this section we explore ReStore in isolation. We first choose the number of redundant copies (Section 8.3.1). Next, we analyze ReStore's performance and experimentally optimize the size of permutation ranges (Section 8.3.2).

8.5.2.a) Number of Redundant Copies.

Figure 8.3 shows the result of a simulation of our data distribution: We continue simulating the failure of random PEs until at least for one data block no copies remain on the surviving PEs. We can see that even for 2^{25} PEs, more than 1 % of all PEs have to fail until we can no longer recover all data when using $r = 4$ redundant copies. Even in the event of an irrecoverable data loss, the program will not crash, but will have to merely reload the input data from disk. For applications running on fewer PEs, an even smaller number r of redundant copies is sufficient to yield data loss unlikely. For all further experiments we therefore set the number of redundant copies to $r := 4$. We compare the values obtained by applying the formula in Section 8.3.4 with the values obtained by simulation in Figure 8.4 showing that our theoretical formula matches the simulation very closely.

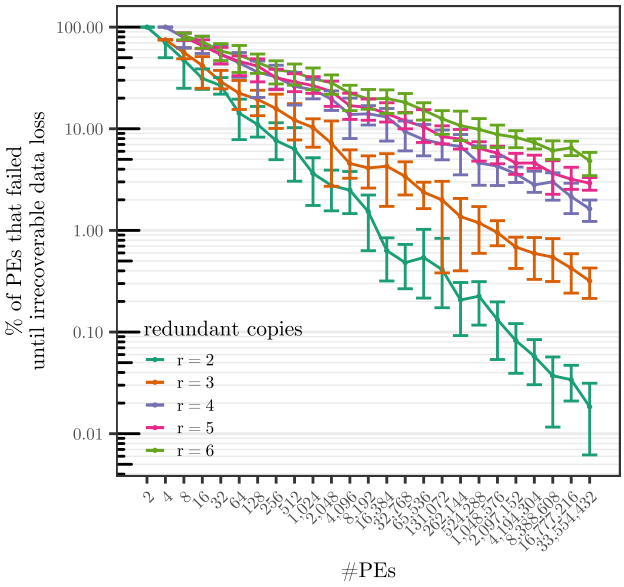


Figure 8.3: Percentage of failed PEs until all redundant copies of one data block are lost. Simulation of the data distribution described in Section 8.3.1. We continue simulating the failure of random PEs until there is at least one data block with no remaining copies on the surviving PEs.

8.5.2.b) Performance.

For all experiments in this section we use data blocks of size 64 B and 16 MiB of data per PE. We show results for three different operations: In the `submit` operation we pass 16 MiB on all PEs to ReStore's submit function. In `load 1 % data` we load the data submitted by 1 % of the PEs with contiguous data block IDs evenly across all PEs. So for n total data blocks, we pick a random starting PE i and request data blocks $i \cdot n/p$ to $(i + 0.01 \cdot p) \cdot n/p$. This simulates the requests expected if 1 % of PEs fail at once. In `load all data` we load all data stored in ReStore evenly distributed across all PEs in a way that no PE loads the same data it originally submitted.

By decreasing the size of permutation ranges (Section 8.3.2) we can control how many PEs can participate in sending requested data: When using smaller permutation ranges, more PEs are able to serve parts of the requested data to the requesting PEs. Small permutation ranges, on the other hand, lead to fragmentation of the data and therefore induce many small messages. In Figure 8.5 we show the number of bytes per permutation range on the x-axis and the running times of `submit` and `load 1 % data` on the y-axis for different numbers of PEs. We do not show results for `load all data` because permutations even have a negative effect

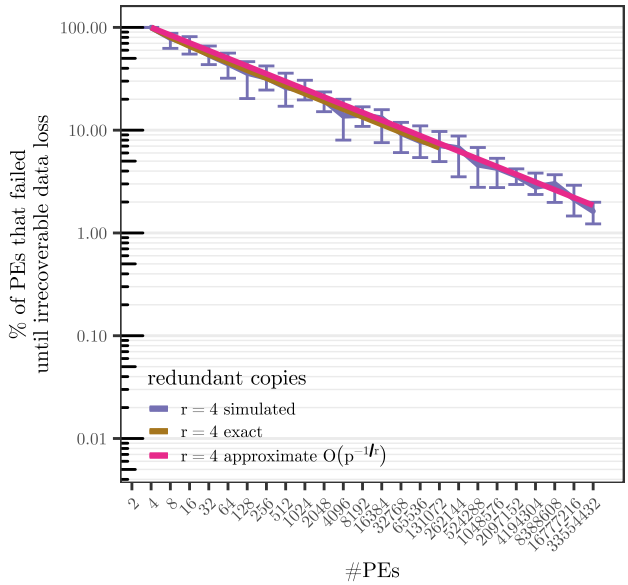


Figure 8.4: Percentage of failed PEs until all redundant copies of one data block are lost. Comparison of the probability given by the equations in Section 8.3.4 and the simulated values from Figure 8.3.

on performance here (Figure 8.6). We therefore recommend turning them off when using a recovery mechanism which loads all data stored in ReStore. We observe that for few bytes per permutation range, both `submit` and `load 1 % data` are slower by up to an order of magnitude than the fastest configuration because of a high bottleneck number of messages. Approaching 16 MiB of data per permutation range, fewer PEs can participate in sending data. Between these two extremes, there is a range of permutation range sizes which yield fast running times. For all further experiments, we thus fix the amount of data per permutation range to 256 KiB (0.65 ms to 2.27 ms for `load 1 % data` on 48 to 6 144 PEs). For 16 MiB of data per PE, every PE receives approximately 164 KiB in `load 1 % data` which results in an average of two PEs requesting the same permutation range and therefore induces a sparse communication pattern. On the sending side, this implies that the data submitted by a single PE is distributed among 64 permutations ranges. With $r = 4$ redundant copies this results in up to $64 \cdot 4 = 256$ PEs that participate in serving this part of the data.

As expected, enabling random permutations speeds up `load 1 % data` and slows down `load all data`, especially for runs on many PEs (Figure 8.6). This is because in `load all data`, even without permutations, every PE sends some part of the data. By enabling permutations, the data requested by a PE is distributed among

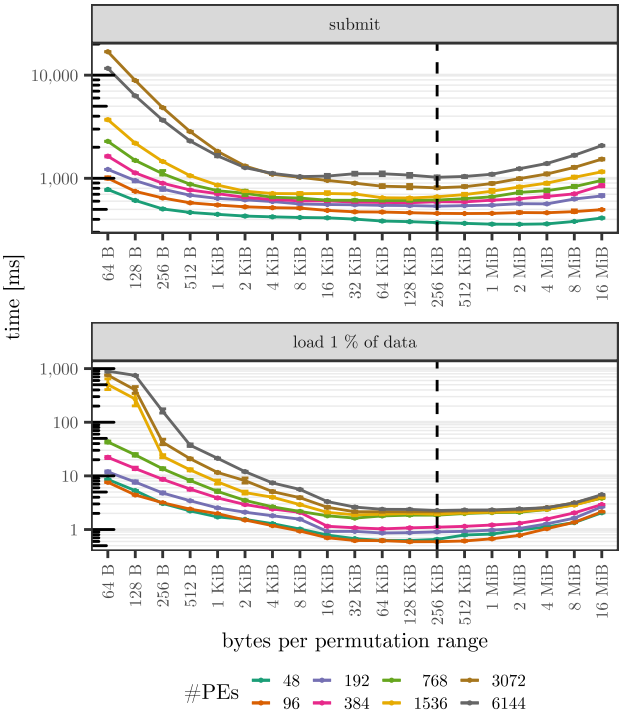


Figure 8.5: Influence of the number of bytes per permutation range on the running time of submitting to and loading from ReStore.

more sending PEs, resulting in a denser communication pattern. We can tolerate an increase in running time of `submit` as it is called only once in the case of only submitting input data. In contrast, a load is issued after every failure.

8.5.3 Applications

To demonstrate realistic use cases of ReStore we use it to restore lost data in two different real-world applications. Figure 8.7 shows running times for a small example application that computes a k-means clustering [Mac67].[3] Each PE holds 65 536 points in a 32-dimensional space as input with 8 byte double precision floating point values per dimension resulting in 16 MiB of input data. All PEs start with the same 20 random starting centers. Iteratively, each PE assigns the nearest center to each of its local points and all PEs collaboratively calculate new centers positions

[3]We ran these experiments with IntelMPI, because its `Group_*`-functions—which we use to determine which PEs failed—perform better than OpenMPI's.

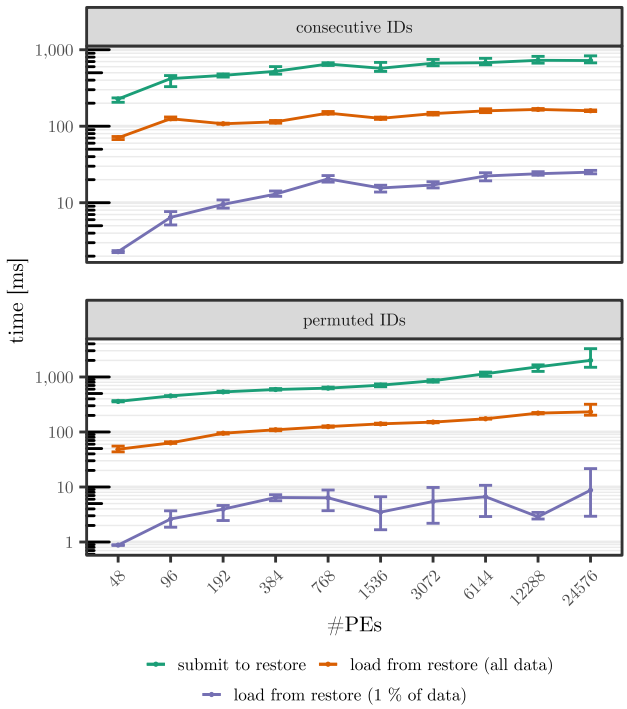


Figure 8.6: Weak scaling experiment (16 MiB per PE) of our three benchmark operations with and without ID randomization. We copy all data over the network—i.e., no rank holds a copy of its requested data in its local part of the ReStore storage.

using an all-reduce-operation over k elements. If a PE fails, the remaining PEs divide the failed PE's data points evenly among them using ReStore and continue with the calculation. We perform 500 iterations of the algorithm and simulate an expected failure of 1 % of all nodes distributed uniformly at random during these iterations. This is done by determining a suitable probability for each PE to fail in each iteration of the algorithm.[4] We find that ReStore accounts for only 1.6 % (median) of the overall running time on up to 24 576 PEs with up to 262 PEs failing. Note that the overall running time increases by more than ReStore's overhead for large PE counts mainly due to MPI operations used to restore a functioning communicator after a PE failure as well as the reduced number of PEs participating in the computation after a failure.

Next, we demonstrate ReStore's performance for the fault tolerant version of the highly complex and widely used phylogenetic tree inference software RAxML-NG

[4] Using a discrete exponential decay with 1 % of failed PEs after 500 iterations.

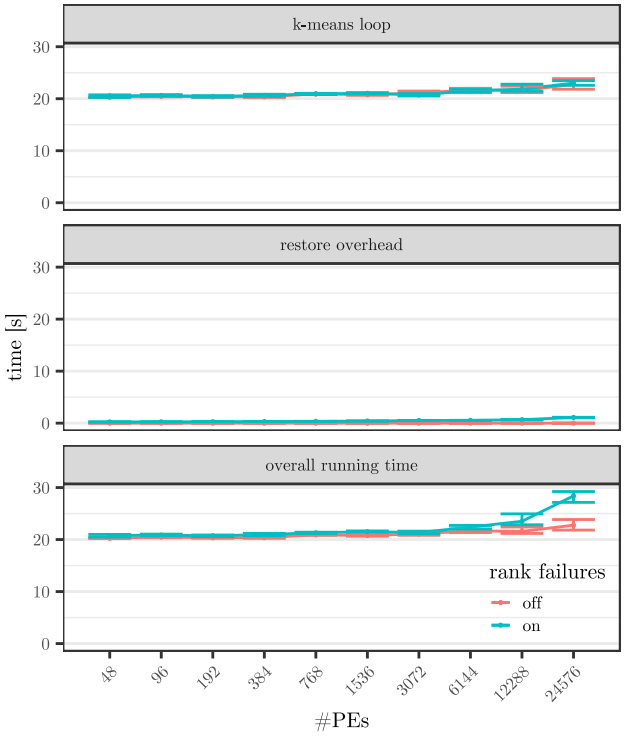
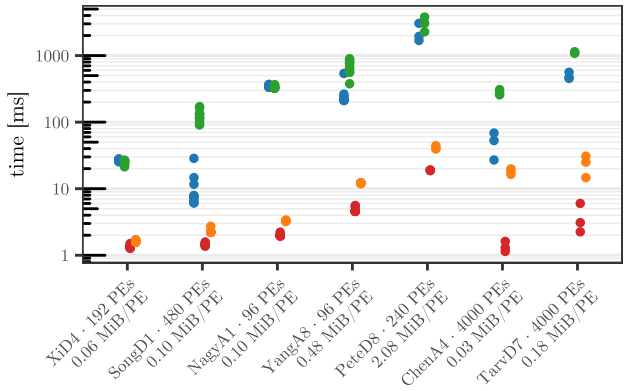


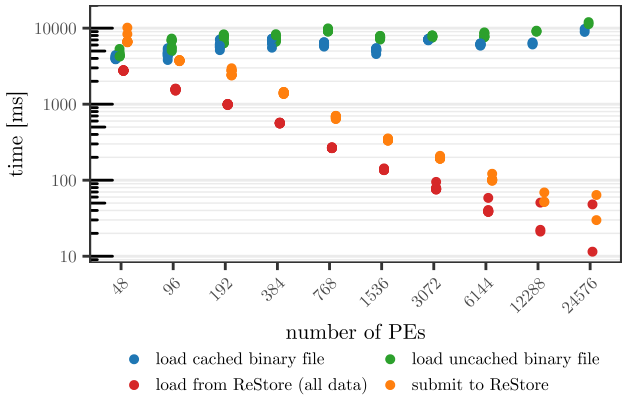
Figure 8.7: Running time of the k-means clustering algorithm with and without failures on 16 MiB of data per PE. *k-means loop:* time spent for the core clustering algorithm. *Restore overhead:* time spent in ReStore's functions. *Overall running time* also includes additional work required for attaining fault tolerance, such as a load balancer to determine how to redistribute data and MPI functions for identifying the failed PEs.

by Kozlov et al. [Koz+19]—called FT-RAxML-NG [Hüb+21a][5]—using the same empirical datasets as in [Hüb+21a] (Figure 8.8 (a)). Additionally, we use a 19.1 GiB synthetic dataset [AKS14] for scaling experiments (Figure 8.8 (b)). FT-RAxML-NG redistributes its input data among all surviving PEs. We therefore deactivate permutation ranges for this application. We compare ReStore's performance against FT-RAxML-NG's currently implemented recovery mechanism: Loading the data from the PFS using RAxML-NG's dedicated binary file format (`RBA`) which enables rapidly reading only the required subset of the input matrix. We distinguish between the input files being uncached by the file system (in the first read) and being cached by previous reads. Both, submitting data to ReStore and loading

[5] `https://github.com/lukashuebner/ft-raxml-ng/tree/restore-paper`



(a) Real world datasets



(b) 19.1 GiB synthetic dataset

Figure 8.8: Performance of data loading after a fault in FT-RAxML-NG. In subplot (a), labels on the x-axis show the name of the data set, the number of PEs used for that data set, and the corresponding amount of input data per PE.

data after a failure, is faster than the original method of loading the data from files—often by more than an order of magnitude. On the synthetic data set, for low PE counts, submitting to ReStore is slower than reloading from a file. However, this is negligible because an actual phylogenetic inference on this dataset requires terabytes of memory for likelihood calculations and would therefore never run on this few nodes. We also want to emphasize that submitting to ReStore has to be done only once in FT-RAxML-NG, while loading has to be conducted after every failure.

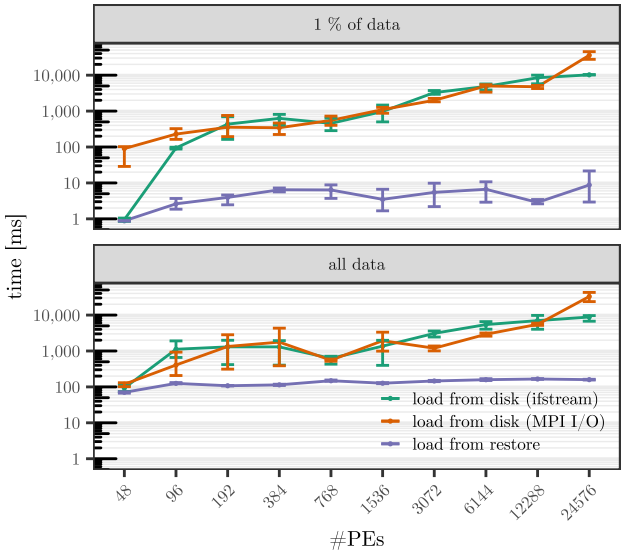


Figure 8.9: Loading performance of ReStore vs. loading from files on the clusters parallel file system, representing the approach of most checkpointing libraries.

8.5.4 Comparison with Other Approaches

We now evaluate ReStore's performance in comparison to other checkpointing approaches. As shown in Section 8.2, most checkpointing libraries store checkpoints on the PFS. Most on-disk and all in-memory checkpointing libraries support only substituting recovery, i.e., no shrinking recovery, yielding a comparison with ReStore challenging. Additionally, as detailed in Section 8.2.1, to the best of our knowledge, there exists no in-memory checkpointing library which is still maintained and working to compare ReStore to.

8.5.4.a) Comparing to Disk-Based Approaches

We compare ReStore against loading a copy of the data stored on the PFS (Figure 8.9). We create this file such that reading is a single consecutive read and therefore as fast as possible. We show running times for reading a separate file for each reading PE using C++'s `ifstream` and reading a single file for all PEs with `MPI_File_read_at_all` (MPI I/O in the plot). This is a lower bound for all checkpointing libraries that have to read their data from disk. ReStore outperforms disk access (`ifstream`) on 24 576 PEs by a factor of 206 (1 % of data; median) and 55 (all data) respectively.

8.5.4.b) Comparing to Reported Measurements

Gamell et al. [Gam+14] measure approximately 115 ms to write a checkpoint with 14.8 MB per rank on 1000 ranks using Fenix. Fenix implements a replication level of $r = 1$. This means, that there exists a single copy of the data in addition to the data the ranks are actively working on. According to our definition (Section 8.3.4), a single rank failure will cause irrecoverable data loss. This works in practice, as long as the data which resided on the failed rank(s) does not need to be restored. To serialize and store 16 MiB per rank on 1536 ranks (32 nodes) with a replication level of $r = 1$ and using consecutive IDs, ReStore needs (126 ± 3) ms ($\mu \pm \sigma$, 10 repeats). Gamell et al. [Gam+14] expect Fenix's recovery time to be the same as its checkpointing time but do not provide experimental results for that claim. ReStore restores the data of a single rank to another single rank in our experiments in (21 ± 2) ms. ReStore additionally offers to restore the data of a single rank scattered to all surviving ranks. This operation requires (20 ± 5) ms in our experiments. In the case that one expects more than one recovery per checkpoint ReStore offers ID permutations to speed up recovery at the cost of slower checkpoint creation (Section 8.3.2). This would for example be the case for static input data, of which multiple ranks need different but small fractions after getting assigned new work following a rank failure. With ID permutations enabled, saving the data to ReStore takes (215 ± 9) ms in our experiments. Restoring the data takes (15 ± 3) ms if restoring all the data to a single rank and (0.9 ± 0.2) ms if restoring the data scattered across the surviving ranks. As the latter evenly distributes the data across the surviving ranks, we expect it to become the more common scenario when working in a shrinking setting.

Bartsch et al. [Bar+17] report GPI_CP to require approximately 1 s to initialize, 200 ms to create a checkpoint and 15 ms to restore data from a checkpoint.

Fenix's performance was measured on a Cray XK7 system with 16 cores per node and a 160 GB s^{-1} network [Cra13]. GPI_CP's performance was measured on an unnamed system with 16 cores per node and a QDR Infiniband network. We measured ReStore's performance on the SuperMUC-NG, which has 48 cores per node and an OmniPath interconnection with 100 Gbit s^{-1} (Section 8.5.1). We choose our experiments such that data is always copied between different nodes and never between two processes running on the same node. Thus, all 48 processes on a single node have to share the same interconnect. Considering that we evaluate ReStore on a slower network than Fenix, we expect an even more favourable comparison when having access to a similar HPC system.

Lu [Lu05] reports checkpoint creation times of 8 s to 20 s for 157 MB to 182 MB on 448 ranks. They report restoration times of 20 s to 48 s. Thus, assuming linear scaling, we expect checkpoint creation times of approximately 1 s and restoration times of approximately 2 s for 16 MiB of data. Lu's algorithm is thus an order of magnitude slower than ReStore and Fenix. We assume this is due to the fact, that Lu's algorithm uses erasure codes (Section 8.3.3).

To summarize, ReStore can be configured to create and restore from checkpoints in the same manner and approximately the same time as existing checkpointing solutions. ReStore additionally has functionality to (a) increase the replication level (b) restore the data in a scattered manner to multiple ranks instead of to one rank and (c) enable ID permutations to decrease time to restore the data by an order of magnitude while doubling the time taken to create a checkpoint. The latter option is for example useful when creating a replicated storage for the input data of a program, which has to be partially reload after a failure.

8.6 Conclusion and Future Work

We show that by using a suitable data distribution strategy, recovery of lost data after a failure is possible in tens to hundreds of milliseconds, depending on the amount of data loaded. We achieve this by using a distribution scheme for redundant copies that ensures a low probability of data loss and a rapid recovery of the data. We also provide the—to the best of our knowledge—first in-memory checkpointing library which supports shrinking recovery, that is ReStore is able to restore the data of the failed PEs scattered to multiple or all surviving ranks instead of to a single respawned or spare PE. This alleviates the need for the application to allocate spare nodes which participate in the computation only in case of a node failure, thus increasing computation efficiency. We supply an analysis of the probability of irrecovable data loss and propose a data distribution to easily restore lost replicas after a failure. Experimental and theoretical evaluation of the proposed data redistribution after a node failure constitutes part of future work. This further decreases the probability to lose all copies of any data. With our C++ library, we were able to improve recovery performance of FT-RAxML-NG [Koz+19; Hüb+21a] by up to two orders of magnitude. By using the proposal implementation of the fault tolerance mechanisms included in the recent MPI 4.0 standard, our library can be used by applications on HPC systems once the new standard is implemented. We also plan on evaluating ReStore for checkpointing of dynamic program state and extend its API for different data formats (e.g., 2D data).

Appendix

Publications and Supervised Theses

In Conference Proceedings

Demian Hespe, Lukas Hübner, Lorenz Hübschle-Schneider, Peter Sanders, and Dominik Schreiber. "Scalable Discrete Algorithms for Big Data Applications". In: *High Performance Computing in Science and Engineering'21: Transactions of the High Performance Computing Center, Stuttgart (HLRS) 2021*. Springer, 2023, pages 439–449. DOI: 10.1007/978-3-031-17937-2_27

Lukas Hübner, Demian Hespe, Peter Sanders, and Alexandros Stamatakis. "ReStore: In-Memory REplicated STORagE for Rapid Recovery in Fault-Tolerant Algorithms". In: *2022 IEEE/ACM 12th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. 2022, pages 24–35. DOI: 10.1109/FTXS56515.2022.00008

Demian Hespe, Sebastian Lamm, and Christian Schorr. "Targeted Branching for the Maximum Independent Set Problem". In: *19th International Symposium on Experimental Algorithms, SEA 2021, June 7-9, 2021, Nice, France*. 2021, 17:1–17:21. DOI: 10.4230/LIPIcs.SEA.2021.17

Demian Hespe, Sebastian Lamm, Christian Schulz, and Darren Strash. "WeGotYouCovered: The Winning Solver from the PACE 2019 Challenge, Vertex Cover Track". In: *Proceedings of the SIAM Workshop on Combinatorial Scientific Computing, CSC 2020, Seattle, USA, February 11-13, 2020*. 2020, pages 1–11. DOI: 10.1137/1.9781611976229.1

Damir Ferizovic, Demian Hespe, Sebastian Lamm, Matthias Mnich, Christian Schulz, and Darren Strash. "Engineering Kernelization for Maximum Cut". In: *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*. 2020, pages 27–41. DOI: 10.1137/1.9781611976007.3

Demian Hespe and Peter Sanders. "More Hierarchy in Route Planning Using Edge Hierarchies". In: *19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems, ATMOS 2019, September 12-13, 2019, Munich, Germany*. 2019, 10:1–10:14. DOI: 10.4230/OASIcs.ATMOS.2019.10

Demian Hespe, Christian Schulz, and Darren Strash. "Scalable Kernelization for Maximum Independent Sets". In: *Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments, ALENEX 2018, New Orleans, LA, USA, January 7-8, 2018*. 2018, pages 223–237. DOI: 10.1137/1.9781611975055.19

Journal Articles

Lukas Hübner, Alexey M. Kozlov, Demian Hespe, Peter Sanders, and Alexandros Stamatakis. "Exploring Parallel MPI Fault Tolerance Mechanisms for Phylogenetic Inference with RAxML-NG". in: *Bioinformatics* 37.22 (2021), pages 4056–4063. DOI: 10.1093/bioinformatics/btab399

Demian Hespe, Christian Schulz, and Darren Strash. "Scalable Kernelization for Maximum Independent Sets". In: *ACM Journal of Experimental Algorithmics* 24.1 (2019), 1.16:1–1.16:22. DOI: 10.1145/3355502

Technical Reports

Demian Hespe, Lukas Hübner, Peter Sanders, and Alexandros Stamatakis. "ReStore: In-Memory REplicated STORagE for Rapid Recovery in Fault-Tolerant Algorithms". In: *CoRR* abs/2203.01107 (2022). arXiv: 2203.01107

Demian Hespe, Sebastian Lamm, and Christian Schorr. "Targeted Branching for the Maximum Independent Set Problem". In: *CoRR* abs/2102.01540 (2021). arXiv: 2102.01540

Lukas Hübner, Alexey M. Kozlov, Demian Hespe, Peter Sanders, and Alexandros Stamatakis. "Exploring Parallel Mpi Fault Tolerance Mechanisms for Phylogenetic Inference with RAxML-NG". in: *bioRxiv* (2021). DOI: 10.1101/2021.01.15.426773

Demian Hespe, Sebastian Lamm, Christian Schulz, and Darren Strash. "WeGotYouCovered: The Winning Solver from the PACE 2019 Implementation Challenge, Vertex Cover Track". In: *CoRR* abs/1908.06795 (2019). arXiv: 1908.06795

Demian Hespe and Peter Sanders. "More Hierarchy in Route Planning Using Edge Hierarchies". In: *CoRR* abs/1907.03535 (2019). arXiv: 1907.03535

Damir Ferizovic, Demian Hespe, Sebastian Lamm, Matthias Mnich, Christian Schulz, and Darren Strash. "Engineering Kernelization for Maximum Cut". In: *CoRR* abs/1905.10902 (2019). arXiv: 1905.10902

Demian Hespe, Christian Schulz, and Darren Strash. "Scalable Kernelization for Maximum Independent Sets". In: *CoRR* abs/1708.06151 (2019). arXiv: 1708.06151

Demian Hespe, Martin Weidner, Jonathan Dees, and Peter Sanders. "Fast OLAP Query Execution in Main Memory on Large Data in a Cluster". In: *CoRR* abs/1709.05183 (2017). arXiv: 1709.05183

Theses

Demian Hespe. “Scalable Kernelization for the Maximum Independent Set Problem”. Master’s thesis. Karlsruhe Institute of Technology, 2017

Demian Hespe. “Communication Efficient Algorithms for Distributed Olap Query Execution”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2014

Supervised Theses

Charel Mercatoris. “Scalable Decentralized Fault-Tolerant MapReduce for Iterative Algorithms”. Master’s thesis. Karlsruhe Institute of Technology, 2021

Christian Schorr. “Improved Branching Strategies for Maximum Independent Sets”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2020

Lukas Hübner. “Load-Balance and Fault-Tolerance for Massively Parallel Phylogenetic Inference”. Master’s thesis. Karlsruhe Institute of Technology, 2020

Damir Ferizovic. “A Practical Analysis of Kernelization Techniques for the Maximum Cut Problem”. Master’s Thesis. Karlsruhe Institute of Technology, 2019

Tom George. “Distributed Kernelization for Independent Sets”. Bachelor’s Thesis. Karlsruhe Institute of Technology, 2018

Bibliography

[Abr+11] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. "A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks". In: *10th Symposium on Experimental Algorithms (SEA)*. 2011, pages 230–241. DOI: 10.1007/978-3-642-20662-7_20. [see page 72]

[Abu+07] Faisal N. Abu-Khzam, Michael R. Fellows, Michael A. Langston, and W. Henry Suters. "Crown Structures for Vertex Cover Kernelization". In: *Theory of Computing Systems* 41.3 (2007), pages 411–430. DOI: 10.1007/s00224-007-1328-0. [see pages 12, 17, 52]

[AC20] Maram Alsahafy and Lijun Chang. "Computing Maximum Independent Sets Over Large Sparse Graphs". In: *International Conference on Web Information Systems Engineering*. Springer. 2020, pages 711–727. [see pages 28, 30]

[Aga+04] Saurabh Agarwal, Rahul Garg, Meeta Sharma Gupta, and José E. Moreira. "Adaptive Incremental Checkpointing for Massively Parallel Systems". In: *Proceedings of the 18th Annual International Conference on Supercomputing, ICS 2004, Saint Malo, France, June 26 - July 01, 2004*. 2004, pages 277–286. DOI: 10.1145/1006209.1006248. [see pages 109–111]

[AHE18] Rizwan A. Ashraf, Saurabh Hukerikar, and Christian Engelmann. "Shrink or Substitute: Handling Process Failures in HPC Systems Using In-Situ Recovery". In: *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2018, Cambridge, United Kingdom, March 21-23, 2018*. 2018, pages 178–185. DOI: 10.1109/PDP2018.2018.00032. [see pages 88, 111]

[AI16] Takuya Akiba and Yoichi Iwata. "Branch-And-Reduce Exponential/FPT Algorithms in Practice: A Case Study of Vertex Cover". In: *Theoretical Computer Science* 609 (2016), pages 211–225. DOI: 10.1016/j.tcs.2015.09.023. [see pages 9, 12–14, 16–19, 28–30, 34–37]

[AKS14] Andre J. Aberer, Kassian Kobert, and Alexandros Stamatakis. "ExaBayes: Massively Parallel Bayesian Tree Inference for the Whole-Genome Era". In: *Molecular Biology and Evolution* 31.10 (2014), pages 2553–2556: Oxford University Press. DOI: 10.1093/molbev/msu236. [see page 123]

[Ali+16] Md. Mohsin Ali, Peter E. Strazdins, Brendan Harding, and Markus Hegland. "Complex Scientific Applications Made Fault-Tolerant with the Sparse Grid Combination Technique". In: *International Journal of High Performance Computing Applications* 30.3 (2016), pages 335–359. DOI: 10.1177/1094342015628056. [see page 110]

[ALS13] Julian Arz, Dennis Luxen, and Peter Sanders. "Transit Node Routing Reconsidered". In: *12th International Symposium on Experimental Algorithms (SEA)*. 2013, pages 55–66. [see page 72]

[AO09] Emely Arráiz and Oswaldo Olivo. "Competitive Simulated Annealing and Tabu Search Algorithms for the Max-Cut Problem". In: *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*. 2009, pages 1797–1798. DOI: 10.1145/1569901.1570167. [see page 52]

[Apa] Apache. *Apache Hadoop*. online. URL: https://hadoop.apache.org. [see page 92]

[ARW12] Diogo V. Andrade, Mauricio G.C. Resende, and Renato F. Werneck. "Fast Local Search for the Maximum Independent Set Problem". In: *Journal of Heuristics* 18.4 (2012), pages 525–547: Springer. DOI: 10.1007/s10732-012-9196-4. [see pages 16, 18]

[Bar+17] Valeria Bartsch, Rui Machado, Dirk Merten, Mirko Rahn, and Franz-Josef Pfreundt. "GASPI/GPI In-Memory Checkpointing Library". In: *Euro-Par 2017: Parallel Processing: 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1, 2017, Proceedings*. 2017, pages 497–508. DOI: 10.1007/978-3-319-64203-1_36. [see pages 110–112, 126]

[Bar+88] Francisco Barahona, Martin Grötschel, Michael Jünger, and Gerhard Reinelt. "An Application of Combinatorial Optimization to Statistical Physics and Circuit Layout Design". In: *Operations Research* 36.3 (1988), pages 493–513. DOI: 10.1287/opre.36.3.493. [see pages 51, 52]

[Bar82] Francisco Barahona. "On the Computational Complexity of Ising Spin Glass Models". In: *Journal of Physics A: Mathematical and General* 15.10 (1982), page 3241: IOP Publishing. DOI: 10.1088/0305-4470/15/10/028. [see pages 51, 52]

[Bar96] Francisco Barahona. "Network Design Using Cut Inequalities". In: *SIAM Journal on Optimization* 6.3 (1996), pages 823–837: SIAM. DOI: 10.1137/S1052623494279134. [see pages 51, 52]

[Bas+07] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. "Fast Routing in Road Networks with Transit Nodes". In: *Science* 316.5824 (2007), page 566 . [see page 72]

[Bas+16] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. "Route Planning in Transportation Networks". In: *Algorithm Engineering*. Springer, 2016, pages 19–80. [see pages 1, 7, 69, 72, 77]

[Bat+13] Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. "Minimum Time-Dependent Travel Times with Contraction Hierarchies". In: *ACM Journal of Experimental Algorithmics* 18 (2013). DOI: 10.1145/2444016.2444020. [see page 83]

[Bat+14] M. Batsyn, B. Goldengorin, E. Maslov, and P. Pardalos. "Improvements to MCS Algorithm for the Maximum Clique Problem". English. In: *Journal of Combinatorial Optimization* 27.2 (2014), pages 397–416: Springer US. DOI: 10.1007/s10878-012-9592-6. [see pages 11, 18]

[Bau+10] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. "Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm". In: *ACM Journal of Experimental Algorithmics* 15 (2010). DOI: 10.1145/1671970.1671976. [see page 72]

[Bau+11] Leonardo Arturo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. "FTI: High Performance Fault Tolerance Interface for Hybrid Systems". In: *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*. 2011, 32:1–32:32. DOI: 10.1145/2063384.2063427. [see pages 103, 109–111, 113, 115]

[Ben+11] Thierry Benoist, Bertrand Estellon, Frédéric Gardi, Romain Megel, and Karim Nouioua. "Localsolver 1.x: A Black-Box Local-Search Solver for 0-1 Programming". In: *4OR. A Quarterly Journal of Operations Research* 9.3 (2011), page 299: Springer. [used in this work: Localsolver 8.0]. DOI: 10.1007/s10288-011-0165-9. URL: https://www.localsolver.com/. [see pages 52, 62]

[BH13] Una Benlic and Jin-Kao Hao. "Breakout Local Search for the Max-Cut Problem". In: *Engineering Applications of Artificial Intelligence* 26.3 (2013), pages 1162–1173. DOI: 10.1016/j.engappai.2012.09.001. [see page 52]

[BH14] Maciej Besta and Torsten Hoefler. "Fault Tolerance for Remote Memory Access Programming Models". In: *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*. 2014, pages 37–48. DOI: 10.1145/2600212.2600224. [see pages 110–112, 115]

[Bin+16] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. "Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++". In: *2016 IEEE International Conference on Big Data (IEEE BigData 2016), Washington DC, USA, December 5–8, 2016*. 2016, pages 172–183. DOI: 10.1109/BigData.2016.7840603. [see pages 3, 91, 93, 97, 99]

[Bla+13] Wesley Bland, Aurélien Bouteiller, Thomas Hérault, George Bosilca, and Jack J. Dongarra. "Post-Failure Recovery of MPI Communication Capability: Design and Rationale". In: *International Journal of High Performance Computing Applications* 27.3 (2013), pages 244–254. DOI: 10.1177/1094342013488238. [see pages 88, 117]

[Bos+08] George Bosilca, Remi Delmas, Jack J. Dongarra, and Julien Langou. "Algorithmic Based Fault Tolerance Applied to High Performance Computing". In: *CoRR* abs/0806.3121 (2008). arXiv: 0806.3121. [see page 110]

[Bou+12] Nicolas Bourgeois, Bruno Escoffier, Vangelis Th. Paschos, and Johan M. M. van Rooij. "Fast Algorithms for Max Independent Set". In: *Algorithmica* 62.1-2 (2012), pages 382–415. DOI: 10.1007/s00453-010-9460-7. [see page 29]

[Bou19] Aurelien Bouteiller. *ULFM 4.0.2u1 Release Notes*. online. Nov. 2019. URL: https://fault-tolerance.org/2019/11/18/ulfm-4-0-2u1/. [see page 111]

[BT07] Sergiy Butenko and Svyatoslav Trukhanov. "Using Critical Sets to Solve the Maximum Independent Set Problem". In: *Operations Research Letters* 35.4 (2007), pages 519–524. DOI: 10.1016/j.orl.2006.07.004. [see page 12]

[But+02] S. Butenko, P. Pardalos, I. Sergienko, V. Shylo, and P. Stetsyuk. "Finding Maximum Independent Sets in Graphs Arising from Coding Theory". In: *Proc. 2002 ACM Symposium on Applied Computing (SAC'02)*. 2002, pages 542–546. DOI: 10.1145/508791.508897. [see page 12]

[But+09] Sergiy Butenko, Panos Pardalos, Ivan Sergienko, Vladimir Shylo, and Petro Stetsyuk. "Estimating the Size of Correcting Codes Using Extremal Graph Problems". In: *Optimization*. Volume 32. Springer Optimization and Its Applications. Springer, 2009, pages 227–243. DOI: 10.1007/978-0-387-98096-6_12. [see page 12]

[BW06] Sergiy Butenko and Wilbert E. Wilhelm. "Clique-Detection Models in Computational Biochemistry and Genomics". In: *European Journal of Operational Research* 173.1 (2006), pages 1–17. DOI: 10.1016/j.ejor.2005.05.026. [see page 9]

[Cap+14] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. "Toward Exascale Resilience: 2014 Update". In: *Supercomputing Frontiers and Innovations* 1.1 (2014), pages 5–28. DOI: 10.14529/jsfi140101. [see pages 1, 87]

[CD96] Tzi-cker Chiueh and Peitao Deng. "Evaluation of Checkpoint Mechanisms for Massively Parallel Machines". In: *Digest of Papers: FTCS-26, The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing, Sendai, Japan, June 25-27, 1996*. 1996, pages 370–379. DOI: 10.1109/FTCS.1996.534622. [see page 115]

[CFJ05] Benny Chor, Mike Fellows, and David Juedes. "Linear Kernels in Linear Time, or How to Save K Colors in $o(n^2)$ Steps". In: *Graph-Theoretic Concepts in Computer Science*. Edited by Juraj Hromkovič, Manfred Nagl, and Bernhard Westfechtel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pages 257–269. DOI: 10.1007/978-3-540-30559-0_22. [see page 11]

[Cha19] Lijun Chang. "Efficient Maximum Clique Computation Over Large Sparse Graphs". In: *Proc. 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2019, pages 529–538. DOI: 10.1145/3292500.3330986. [see page 12]

[Che+08] Tammy M. K. Cheng, Yu-En Lu, Michele Vendruscolo, Pietro Liò, and Tom L. Blundell. "Prediction by Graph Theoretic Measures of Structural Effects in Proteins Arising from Non-Synonymous Single Nucleotide Polymorphisms". In: *PLoS Computational Biology* 4.7 (2008). DOI: 10.1371/journal.pcbi.1000135. [see page 9]

[Chi+07] Charles Chiang, Andrew B Kahng, Subarnarekha Sinha, Xu Xu, and Alexander Z Zelikovsky. "Fast and Efficient Bright-Field AAPSM Conflict Detection and Correction". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.1 (2007), pages 115–126. DOI: 10.1109/TCAD.2006.882642. [see pages 51, 52]

[CJM15] Robert Crowston, Mark Jones, and Matthias Mnich. "Max-Cut Parameterized above the Edwards-Erdős Bound". In: *Algorithmica* 72.3 (2015), pages 734–757. DOI: 10.1007/s00453-014-9870-z. [see pages 52, 54, 55]

[CKJ01] Jianer Chen, Iyad A. Kanj, and Weijia Jia. "Vertex Cover: Further Observations and Further Improvements". In: *Journal of Algorithms* 41.2 (2001), pages 280–301. DOI: 10.1006/jagm.2001.1186. [see pages 11, 17]

[CKX10] Jianer Chen, Iyad A. Kanj, and Ge Xia. "Improved Upper Bounds for Vertex Cover". In: *Theoretical Computer Science* 411.40 (2010), pages 3736–3756. DOI: 10.1016/j.tcs.2010.06.026. [see pages 9, 11, 29]

[CLZ17] Lijun Chang, Wei Li, and Wenjie Zhang. "Computing A Near-Maximum Independent Set in Linear Time by Reducing-Peeling". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pages 1181–1196. DOI: 10.1145/3035918.3035939. [see pages 12, 18, 27]

[Con+10] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. "MapReduce Online". In: *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA*. 2010, pages 313–328. URL: http://www.usenix.org/events/nsdi10/tech/full%5C_papers/condie.pdf. [see page 92]

[CP90] Randy Carraghan and Panos M. Pardalos. "An Exact Algorithm for the Maximum Clique Problem". In: *Operations Research Letters* 9.6 (1990), pages 375–382: Elsevier Science Publishers B. V. DOI: 10.1016/0167-6377(90)90057-C. [see page 29]

[Cra13] Cray. *Cray Xk7 Specifications*. online. Jan. 2013. URL: https://web.archive.org/web/20130106091417/http://www.cray.com/Products/Computing/XK7/Specifications.aspx. [see page 126]

[Cro+13] Robert Crowston, Gregory Gutin, Mark Jones, and Gabriele Muciaccia. "Maximum Balanced Subgraph Problem Parameterized above Lower Bound". In: *Theoretical Computer Science* 513 (2013), pages 53–64. DOI: 10.1016/j.tcs.2013.10.026. [see pages 52, 54, 55, 57, 59, 60]

[CZF04] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. "R-MAT: A Recursive Model for Graph Mining". In: *Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004*. 2004, pages 442–446. DOI: 10.1137/1.9781611972740.43. [see page 104]

[Dah+16] Jakob Dahlum, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. "Accelerating Local Search for the Maximum Independent Set Problem". In: *Proc. 15th International Symposium on Experimental Algorithms (SEA 2016)*. Volume 9685. LNCS. Springer, 2016, pages 118–133. DOI: 10.1007/978-3-319-38851-9_9. [see pages 18, 27]

[De 59] Rene De La Briandais. "File Searching Using Variable Length Keys". In: *Papers presented at the the March 3-5, 1959, western joint computer conference*. ACM. 1959, pages 295–298. DOI: 10.1145/1457838.1457895. [see page 58]

[Del+15] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. "Customizable Route Planning in Road Networks". In: *Transportation Science* 51.2 (2015), pages 566–591: INFORMS . [see pages 70, 77]

[DFH19a] M. Ayaz Dzulfikar, Johannes K. Fichte, and Markus Hecher. *Pace2019: Track 1 - Vertex Cover Instances*. Zenodo, July 2019. DOI: 10.5281/zenodo.3368306. [see page 19]

[DFH19b] M. Ayaz Dzulfikar, Johannes K. Fichte, and Markus Hecher. "The PACE 2019 Parameterized Algorithms and Computational Experiments Challenge: The Fourth Iteration (Invited Paper)". In: *14th International Symposium on Parameterized and Exact Computation (IPEC 2019)*. 2019, 25:1–25:23. DOI: 10.4230/LIPIcs.IPEC.2019.25. URL: https://drops.dagstuhl.de/opus/volltexte/2019/11486. [see pages 38, 39]

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Communications of the ACM* 51.1 (2008), pages 107–113. DOI: 10.1145/1327452.1327492. [see pages 3, 91–93]

[DGJ09] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson. *The Shortest Path Problem: Ninth Dimacs Implementation Challenge*. Volume 74. American Mathematical Soc., 2009. [see pages 1, 38, 39, 77]

[DGS18] Iain Dunning, Swati Gupta, and John Silberholz. "What Works Best When? A Systematic Evaluation of Heuristics for Max-Cut and QUBO". In: *INFORMS Journal on Computing* 30.3 (2018), pages 608–624. DOI: 10.1287/ijoc.2017.0798. [see pages 1, 62, 63, 66]

[DHR15] Jack Dongarra, Thomas Herault, and Yves Robert1. *Fault-Tolerance Techniques for High-Performance Computing*. Computer Communications and Networks 1. Springer, Cham, 2015. ISBN: 978-3-319-20943-2. DOI: 10.1007/978-3-319-20943-2_1. [see pages 1, 87]

[Dij59] Edsger W Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numerische Mathematik* 1.1 (1959), pages 269–271: Springer . [see pages 1, 7, 71]

[DM02] Elizabeth D Dolan and Jorge J Moré. "Benchmarking Optimization Software with Performance Profiles". In: *Mathematical Programming* 91.2 (2002), pages 201–213: Springer . [see page 38]

[dSHK13] Samuel de Sousa, Yll Haxhimusa, and Walter G Kropatsch. "Estimation of Distribution Algorithm for the Max-Cut Problem". In: *International Workshop on Graph-Based Representations in Pattern Recognition*. Volume 7877. LNCS. Springer, 2013, pages 244–253. DOI: 10.1007/978-3-642-38221-5_26. [see pages 51, 52, 66]

[DSW16] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. "Customizable Contraction Hierarchies". In: *ACM Journal of Experimental Algorithmics* 21 (2016), pages 1–5: ACM . [see pages 70, 71, 77]

[Edw73] Christopher S Edwards. "Some Extremal Properties of Bipartite Subgraphs". In: *Canadian Journal of Mathematics. Journal Canadien de Mathematiques* 25.3 (1973), pages 475–485: Cambridge University Press. DOI: 10.4153/CJM-1973-048-x. [see pages 52, 54]

[Edw75] Christopher S Edwards. "An Improved Lower Bound for the Number of Edges in a Largest Bipartite Subgraph". In: *Proc. Second Czechoslovak Symposium on Graph Theory, Prague*. 1975, pages 167–181. [see pages 52, 54]

[EG03] Christian Engelmann and Al Geist. "A Diskless Checkpointing Algorithm for Super-Scale Architectures Applied to the Fast Fourier Transform". In: *1st International Workshop on Challenges of Large Applications in Distributed Environments, CLADE@HPDC 2003, Seattle, WA, USA, June 21, 2003*. 2003, page 47. DOI: 10.1109/CLADE.2003.1209999. [see page 110]

[Eka+10] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey C. Fox. "Twister: A Runtime for Iterative MapReduce". In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC 2010, Chicago, Illinois, USA, June 21-25, 2010*. 2010, pages 810–818. DOI: 10.1145/1851476.1851593. [see page 92]

[EM18] Michael Etscheid and Matthias Mnich. "Linear Kernels and Linear-Time Algorithms for Finding Large Cuts". In: *Algorithmica* 80.9 (2018), pages 2574–2615. DOI: 10.1007/s00453-017-0388-z. [see pages 52, 54, 55, 62]

[ER60] Paul Erdős and Alfréd Rényi. "On the Evolution of Random Graphs". In: *Publications of the Mathematical Institute of the Hungarian Academy of Sciences* 5.1 (1960), pages 17–60 . [see pages 104, 105]

[Far+17] Luerbio Faria, Sulamita Klein, Ignasi Sau, and Rubens Sucupira. "Improved Kernels for Signed Max Cut Parameterized above Lower Bound on (r, l)-Graphs". In: *Discrete Mathematics & Theoretical Computer Science* 19.1 (2017). DOI: 10.23638/DMTCS-19-1-14. [see pages 54, 55]

[Fel+18] Michael R. Fellows, Lars Jaffke, Aliz Izabella Király, Frances A. Rosamond, and Mathias Weller. "What Is Known about Vertex Cover Kernelization?" In: *Adventures Between Lower Bounds and Higher Altitudes: Essays Dedicated to Juraj Hromkovič on the Occasion of His 60th Birthday*. Edited by Hans-Joachim Böckenhauer, Dennis Komm, and Walter Unger. Springer, 2018, pages 330–356. DOI: 10.1007/978-3-319-98355-4_19. [see page 11]

[Fer+19] Damir Ferizovic, Demian Hespe, Sebastian Lamm, Matthias Mnich, Christian Schulz, and Darren Strash. "Engineering Kernelization for Maximum Cut". In: *CoRR* abs/1905.10902 (2019). arXiv: 1905.10902. [see pages 7, 51, 132]

[Fer+20] Damir Ferizovic, Demian Hespe, Sebastian Lamm, Matthias Mnich, Christian Schulz, and Darren Strash. "Engineering Kernelization for Maximum Cut". In: *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*. 2020, pages 27–41. DOI: 10.1137/1.9781611976007.3. [see pages 7, 51, 131]

[Fer19] Damir Ferizovic. "A Practical Analysis of Kernelization Techniques for the Maximum Cut Problem". Master's Thesis. Karlsruhe Institute of Technology, 2019. [see pages 55, 133]

[FGK09] Fedor V Fomin, Fabrizio Grandoni, and Dieter Kratsch. "A Measure & Conquer Approach for the Analysis of Exact Algorithms". In: *Journal of the ACM* 56.5 (2009), page 25: ACM. DOI: 10.1145/1552285.1552286. [see pages 11, 17, 28]

[FK10] F.V. Fomin and D. Kratsch. *Exact Exponential Algorithms*. Springer, 2010. DOI: 10.1007/978-3-642-16533-7. [see page 11]

[Fre60] Edward Fredkin. "Trie Memory". In: *Communications of the ACM* 3.9 (1960), pages 490–499. DOI: 10.1145/367390.367400. [see page 58]

[Fun+19] Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. "Communication-Free Massively Distributed Graph Generation". In: *Journal of Parallel and Distributed Computing* 131 (2019), pages 200–217. DOI: 10.1016/j.jpdc.2019.03.011. [see page 62]

[Gam+14] Marc Gamell, Daniel S. Katz, Hemanth Kolla, Jacqueline Chen, Scott Klasky, and Manish Parashar. "Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales". In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*. 2014, pages 895–906. DOI: 10.1109/SC.2014.78. [see pages 1, 87, 110–112, 126]

[Gao+17] Tao Gao, Yanfei Guo, Boyu Zhang, Pietro Cicotti, Yutong Lu, Pavan Balaji, and Michela Taufer. "Mimir: Memory-Efficient and Scalable MapReduce for Large Supercomputing Systems". In: *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. 2017, pages 1098–1108. DOI: 10.1109/IPDPS.2017.31. [see page 92]

[Gao+18] Jian Gao, Jiejiang Chen, Minghao Yin, Rong Chen, and Yiyuan Wang. "An Exact Algorithm for Maximum k-Plexes in Massive Graphs". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. 2018, pages 1449–1455. DOI: 10.24963/ijcai.2018/201. [see page 28]

[Gar+14] Frédéric Gardi, Thierry Benoist, Julien Darlay, Bertrand Estellon, and Romain Megel. *Mathematical Programming Solver Based on Local Search*. FOCUS Series in Computer Engineering. ISTE Wiley, 2014, page 112. DOI: 10.1002/9781118966464. [see page 52]

[Gei+08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. "Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks". In: *7th Workshop on Experimental Algorithms (WEA)*. 2008, pages 319–333. [see pages 70, 71]

[Gei+12] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. "Exact Routing in Large Road Networks Using Contraction Hierarchies". In: *Transportation Science* 46.3 (2012), pages 388–404: INFORMS . [see pages 69–71]

[Geo18] Tom George. "Distributed Kernelization for Independent Sets". Bachelor's Thesis. Karlsruhe Institute of Technology, 2018. [see page 133]

[Geo73] Alan George. "Nested Dissection of a Regular Finite Element Mesh". In: *SIAM Journal on Numerical Analysis* 10.2 (1973), pages 345–363: SIAM . [see page 32]

[GJS74] M. R. Garey, D. S. Johnson, and L. Stockmeyer. "Some Simplified NP-Complete Problems". In: *Proceedings of the 6th ACM Symposium on Theory of Computing*. 1974, pages 47–63. [see page 9]

[GKW06] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. "Reach for A*: Efficient Point-To-Point Shortest Path Algorithms". In: *8th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2006. [see page 72]

[Gos+21] Mikaila J. Gossman, Bogdan Nicolae, Jon C. Calhoun, Franck Cappello, and Melissa C. Smith. "Towards Aggregated Asynchronous Checkpointing". In: *CoRR* abs/2112.02289 (2021). arXiv: 2112.02289. [see page 111]

[Got+19] Lars Gottesbüren, Michael Hamann, Tim Niklas Uhl, and Dorothea Wagner. "Faster and Better Nested Dissection Orders for Customizable Contraction Hierarchies". In: *Algorithms* 12.9 (2019), page 196: Multidisciplinary Digital Publishing Institute . [see page 38]

[GS13] Sunil P. Gavaskar and Ch D. V. Subbarao. "A Survey of Distributed Fault Tolerance Strategies". In: *International Journal of Advanced Research in Computer and Communication Engineering* 2.11 (Nov. 2013) . [see page 111]

[GT88] Andrew V Goldberg and Robert E Tarjan. "A New Approach to the Maximum-Flow Problem". In: *Journal of the ACM (JACM)* 35.4 (1988), pages 921–940: ACM New York, NY, USA . [see page 31]

[Guo+15] Yanfei Guo, Wesley Bland, Pavan Balaji, and Xiaobo Zhou. "Fault Tolerant MapReduce-Mpi for HPC Clusters". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*. 2015, 34:1–34:12. DOI: 10.1145/2807591.2807617. [see page 92]

[Gut] Project Gutenberg. *Project Gutenberg Is a Library of Over 60,000 Free Ebooks*. online. [Online; accessed 8-August-2021]. URL: https://www.gutenberg.org/. [see page 104]

[Gut04] Ronald J. Gutman. "Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks". In: *6th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2004, pages 100–111. [see page 70]

[Har59] Frank Harary. "On the Measurement of Structural Balance". In: *Behavioral Sciences* 4.4 (1959), pages 316–323. DOI: 10.1002/bs.3830040405. [see pages 51, 52]

[Hér+19] Thomas Hérault, Yves Robert, Aurélien Bouteiller, Dorian C. Arnold, Kurt B. Ferreira, George Bosilca, and Jack J. Dongarra. "Checkpointing Strategies for Shared High-Performance Computing Platforms". In: *International Journal of Networking and Computing* 9.1 (2019), pages 28–52. URL: http://www.ijnc.org/index.php/ijnc/article/view/195. [see page 111]

[Hes+17] Demian Hespe, Martin Weidner, Jonathan Dees, and Peter Sanders. "Fast OLAP Query Execution in Main Memory on Large Data in a Cluster". In: *CoRR* abs/1709.05183 (2017). arXiv: 1709.05183. [see page 132]

[Hes+19a] Demian Hespe, Sebastian Lamm, Christian Schulz, and Darren Strash. *WeGotYouCovered*. May 2019. DOI: 10.5281/zenodo.2816116. [see page 20]

[Hes+19b] Demian Hespe, Sebastian Lamm, Christian Schulz, and Darren Strash. "WeGotYouCovered: The Winning Solver from the PACE 2019 Implementation Challenge, Vertex Cover Track". In: *CoRR* abs/1908.06795 (2019). arXiv: 1908.06795. [see pages 7, 9, 16, 132]

[Hes+20] Demian Hespe, Sebastian Lamm, Christian Schulz, and Darren Strash. "WeGotYouCovered: The Winning Solver from the PACE 2019 Challenge, Vertex Cover Track". In: *Proceedings of the SIAM Workshop on Combinatorial Scientific Computing, CSC 2020, Seattle, USA, February 11-13, 2020*. 2020, pages 1–11. DOI: 10.1137/1.9781611976229.1. [see pages 7, 9, 16, 131]

[Hes+22] Demian Hespe, Lukas Hübner, Peter Sanders, and Alexandros Stamatakis. "ReStore: In-Memory REplicated STORagE for Rapid Recovery in Fault-Tolerant Algorithms". In: *CoRR* abs/2203.01107 (2022). arXiv: 2203.01107. [see pages 87, 109, 132]

[Hes+23] Demian Hespe, Lukas Hübner, Lorenz Hübschle-Schneider, Peter Sanders, and Dominik Schreiber. "Scalable Discrete Algorithms for Big Data Applications". In: *High Performance Computing in Science and Engineering'21: Transactions of the High Performance Computing Center, Stuttgart (HLRS) 2021*. Springer, 2023, pages 439–449. DOI: 10.1007/978-3-031-17937-2_27. [see page 131]

[Hes14] Demian Hespe. "Communication Efficient Algorithms for Distributed Olap Query Execution". Bachelor's thesis. Karlsruhe Institute of Technology, 2014. [see page 133]

[Hes17] Demian Hespe. "Scalable Kernelization for the Maximum Independent Set Problem". Master's thesis. Karlsruhe Institute of Technology, 2017. [see page 133]

[HK73] John E Hopcroft and Richard M Karp. "An N^5/2 Algorithm for Maximum Matchings in Bipartite Graphs". In: *SIAM Journal on Computing* 2.4 (1973), pages 225–231: SIAM . [see page 31]

[HLS21a] Demian Hespe, Sebastian Lamm, and Christian Schorr. "Targeted Branching for the Maximum Independent Set Problem". In: *19th International Symposium on Experimental Algorithms, SEA 2021, June 7-9, 2021, Nice, France*. 2021, 17:1–17:21. DOI: 10.4230/LIPIcs.SEA.2021.17. [see pages 7, 9, 27, 131]

[HLS21b] Demian Hespe, Sebastian Lamm, and Christian Schorr. "Targeted Branching for the Maximum Independent Set Problem". In: *CoRR* abs/2102.01540 (2021). arXiv: 2102.01540. [see pages 7, 9, 27, 132]

[HLW02] Frank Harary, Meng-Hiot Lim, and Donald C Wunsch. "Signed Graphs for Portfolio Analysis in Risk Management". In: *IMA J. Mgmt. Math.* 13.3 (2002), pages 201–210: Oxford University Press. DOI: 10.1093/imaman/13.3.201. [see pages 51, 52]

[HS19a] Demian Hespe and Peter Sanders. "More Hierarchy in Route Planning Using Edge Hierarchies". In: *19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems, ATMOS 2019, September 12-13, 2019, Munich, Germany*. 2019, 10:1–10:14. DOI: 10.4230/OASIcs.ATMOS.2019.10. [see pages 7, 69, 131]

[HS19b] Demian Hespe and Peter Sanders. "More Hierarchy in Route Planning Using Edge Hierarchies". In: *CoRR* abs/1907.03535 (2019). arXiv: 1907.03535. [see pages 7, 69, 132]

[HSS18] Demian Hespe, Christian Schulz, and Darren Strash. "Scalable Kernelization for Maximum Independent Sets". In: *Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments, ALENEX 2018, New Orleans, LA, USA, January 7-8, 2018*. 2018, pages 223–237. DOI: 10.1137/1.9781611975055.19. [see pages 12, 52, 131]

[HSS19a] Demian Hespe, Christian Schulz, and Darren Strash. "Scalable Kernelization for Maximum Independent Sets". In: *ACM Journal of Experimental Algorithmics* 24.1 (2019), 1.16:1–1.16:22. DOI: 10.1145/3355502. [see pages 28, 132]

[HSS19b] Demian Hespe, Christian Schulz, and Darren Strash. "Scalable Kernelization for Maximum Independent Sets". In: *CoRR* abs/1708.06151 (2019). arXiv: 1708.06151. [see page 132]

[HT73] John Hopcroft and Robert Tarjan. "Algorithm 447: Efficient Algorithms for Graph Manipulation". In: *Communications of the ACM* 16.6 (1973), pages 372–378: ACM New York, NY, USA . [see page 30]

[Hüb+21a] Lukas Hübner, Alexey M. Kozlov, Demian Hespe, Peter Sanders, and Alexandros Stamatakis. "Exploring Parallel MPI Fault Tolerance Mechanisms for Phylogenetic Inference with RAxML-NG". In: *Bioinformatics* 37.22 (2021), pages 4056–4063. DOI: 10.1093/bioinformatics/btab399. [see pages 89, 110, 115, 123, 127, 132]

[Hüb+21b] Lukas Hübner, Alexey M. Kozlov, Demian Hespe, Peter Sanders, and Alexandros Stamatakis. "Exploring Parallel Mpi Fault Tolerance Mechanisms for Phylogenetic Inference with RAxML-NG". In: *bioRxiv* (2021). DOI: 10.1101/2021.01.15.426773. [see page 132]

[Hüb+22] Lukas Hübner, Demian Hespe, Peter Sanders, and Alexandros Stamatakis. "ReStore: In-Memory REplicated STORagE for Rapid Recovery in Fault-Tolerant Algorithms". In: *2022 IEEE/ACM 12th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. 2022, pages 24–35. DOI: 10.1109/FTXS56515.2022.00008. [see pages 87, 109, 131]

[Hüb20] Lukas Hübner. "Load-Balance and Fault-Tolerance for Massively Parallel Phylogenetic Inference". Master's thesis. Karlsruhe Institute of Technology, 2020. [see page 133]

[IOY14] Y. Iwata, K. Oka, and Y. Yoshida. "Linear-Time FPT Algorithms Via Network Flow". In: *Proc. 25th ACM-SIAM Symposium on Discrete Algorithms*. 2014, pages 1749–1761. DOI: 10.1137/1.9781611973402.127. [see pages 12, 14, 17]

[Ish+91] Kunihiro Ishikawa, Michima Ogawa, Shigetoshi Azuma, and Tooru Ito. "Map Navigation Software of the Electro-Multivision of The'91 Toyota Soarer". In: *Vehicle Navigation and Information Systems Conference*. IEEE. 1991, pages 463–473. [see page 70]

[Joh93] David S Johnson. “Cliques, Coloring, and Satisfiability: Second Dimacs Implementation Challenge”. In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 26 (1993), pages 11–13 . [see pages 38, 39]

[JSQ02] George R. Jagadeesh, Thambipillai Srikanthan, and K. H. Quek. “Heuristic Techniques for Accelerating Hierarchical Routing on Road Networks”. In: *IEEE Transactions on Intelligent Transportation Systems* 3.4 (2002), pages 301–309: IEEE . [see page 70]

[Kar72] Richard M Karp. “Reducibility among Combinatorial Problems”. In: *Complexity of Computer Computations*. The IBM Research Symposia Series. Springer, 1972, pages 85–103. DOI: 10.1007/978-1-4684-2001-2_9. [see page 51]

[Kie+10] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. “Distributed Time-Dependent Contraction Hierarchies”. In: *Experimental Algorithms, 9th International Symposium*. 2010, pages 83–93. DOI: 10.1007/978-3-642-13193-6_8. [see page 9]

[Kiv+14] Raimondas Kiveris, Silvio Lattanzi, Vahab S. Mirrokni, Vibhor Rastogi, and Sergei Vassilvitskii. “Connected Components in MapReduce and Beyond”. In: *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 3-5, 2014*. 2014, 18:1–18:13. DOI: 10.1145/2670979.2670997. [see page 104]

[KLR09] Joachim Kneis, Alexander Langer, and Peter Rossmanith. “A Fine-grained Analysis of a Simple Independent Set Algorithm”. In: *Proc. 29th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009)*. 2009, pages 287–298. DOI: 10.4230/LIPIcs.FSTTCS.2009.2326. [see page 11]

[Koc+13] Gary A. Kochenberger, Jin-Kao Hao, Zhipeng Lü, Haibo Wang, and Fred Glover. “Solving Large Scale Max Cut Problems Via Tabu Search”. In: *Journal of Heuristics* 19.4 (Aug. 2013), pages 565–571. DOI: 10.1007/s10732-011-9189-8. [see page 52]

[Koh+19] Nils Kohl, Johannes Hötzer, Florian Schornbaum, Martin Bauer, Christian Godenschwager, Harald Köstler, Britta Nestler, and Ulrich Rüde. “A Scalable and Extensible Checkpointing Scheme for Massively Parallel Simulations”. In: *International Journal of High Performance Computing Applications* 33.4 (2019). DOI: 10.1177/1094342018767736. [see page 110]

[Koz+19] Alexey M. Kozlov, Diego Darriba, Tomás Flouri, Benoit Morel, and Alexandros Stamatakis. “RAxML-NG: A Fast, Scalable and User-Friendly Tool for Maximum Likelihood Phylogenetic Inference”. In: *Bioinformatics* 35.21 (2019), pages 4453–4455. DOI: 10.1093/bioinformatics/btz305. [see pages 115, 123, 127]

[Koz18] Alexey Kozlov. “Models, Optimizations, and Tools Forlarge-Scale Phylogenetic Inference,handling Sequence Uncertainty,and Taxonomic Validation”. PhD thesis. Karlsruher Institut für Technologie (KIT), Jan. 2018. [see page 115]

[KSV10] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. “A Model of Computation for MapReduce”. In: *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*. 2010, pages 938–948. DOI: 10.1137/1.9781611973075.76. [see page 92]

[KWZ16] Spyros C. Kontogiannis, Dorothea Wagner, and Christos D. Zaroliagis. “Hierarchical Time-Dependent Oracles”. In: *27th International Symposium on Algorithms and Computation (ISAAC)*. 2016, 47:1–47:13. [see page 83]

[Lag+16] Ignacio Laguna, David F. Richards, Todd Gamblin, Martin Schulz, Bronis R. de Supinski, Kathryn Mohror, and Howard Pritchard. “Evaluating and Extending User-Level Fault Tolerance in MPI Applications”. In: *International Journal of High Performance Computing Applications* 30.3 (2016), pages 305–319. DOI: 10.1177/1094342015623623. [see page 110]

[Lam+17] Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. “Finding Near-Optimal Independent Sets at Scale”. In: *Journal of Heuristics* 23.4 (Aug. 2017), pages 207–229. DOI: 10.1007/s10732-017-9337-x. [see page 52]

[Lar07] Craig E Larson. “A note on critical independence reductions”. In: *Bulletin of the Institute of Combinatorics and its Applications* 5 (2007), pages 34–46 . [see page 12]

[Lei23] Leibnitz Computing Centre of the Bavarian Academy of Sciences and Humanities. *Supermuc-Ng*. online. [Online; accessed 10-May-2023]. 2023. URL: https://doku.lrz.de/display/PUBLIC/SuperMUC-NG. [see page 1]

[LFX13] Chu-Min Li, Zhiwen Fang, and Ke Xu. “Combining MaxSAT Reasoning and Incremental Upper Bound for the Maximum Clique Problem”. In: *Proceedings of 25th International Conference on Tools with Artificial Intelligence (ICTAI)*. Nov. 2013, pages 939–946. DOI: 10.1109/ICTAI.2013.143. [see pages 11, 29]

[LJM17] Chu-Min Li, Hua Jiang, and Felip Manyà. “On Minimization of the Number of Branches in Branch-And-Bound Algorithms for the Maximum Clique Problem”. In: *Computers & Operations Research* 84 (2017), pages 1–15. DOI: 10.1016/j.cor.2017.02.017. [see pages 11, 16, 18–20, 29]

[LJX15] Chu-Min Li, Hua Jiang, and Ruchu Xu. “Incremental MaxSAT Reasoning to Reduce Branches in a Branch-And-Bound Algorithm for MaxClique”. In: *Learning and Intelligent Optimization - 9th International Conference*. 2015, pages 268–274. DOI: 10.1007/978-3-319-19084-6_26. [see page 29]

[LK14] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014. [see pages 1, 38, 39]

[LQ10] Chu Min Li and Zhe Quan. “An Efficient Branch-And-Bound Algorithm Based on MaxSAT for the Maximum Clique Problem”. In: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*. 2010. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1611. [see pages 21, 29]

[Lu05] Charng-Da Lu. “Scalable Diskless Checkpointing for Large Parallel Systems”. PhD thesis. University of Illinois at Urbana-Champaign, 2005. [see pages 110–112, 115, 126]

[Mac67] James MacQueen. “Some Methods for Classification and Analysis of Multivariate Observations”. In: *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*. Oakland, CA, USA. 1967, pages 281–297. [see page 121]

[McC95] William F. McColl. “Scalable Computing”. In: *Computer Science Today: Recent Trends and Developments*. Volume 1000. Lecture Notes in Computer Science. Springer, 1995, pages 46–61. DOI: 10.1007/BFb0015236. [see page 92]

[Mem+16] Bunjamin Memishi, Shadi Ibrahim, María S. Pérez, and Gabriel Antoniu. “Fault Tolerance in MapReduce: A Survey”. In: *Resource Management for Big Data Platforms - Algorithms, Modelling, and High-Performance Computing Techniques*. Edited by Florin Pop, Joanna Kolodziej, and Beniamino Di Martino. Computer Communications and Networks. Springer, 2016, pages 205–240. DOI: 10.1007/978-3-319-44881-7_11. [see page 110]

[Mer21] Charel Mercatoris. “Scalable Decentralized Fault-Tolerant MapReduce for Iterative Algorithms”. Master’s thesis. Karlsruhe Institute of Technology, 2021. [see pages 91, 103, 133]

[Möh+05] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. “Partitioning Graphs to Speed up Dijkstra’s Algorithm”. In: *4th Workshop on Efficient and Experimental Algorithms (WEA)*. 2005, pages 189–202. [see page 72]

[Moo+10] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. "Design, Modeling, and Evaluation of a Scalable Multi-Level Checkpointing System". In: *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*. 2010, pages 1–11. DOI: 10.1109/SC.2010.18. [see pages 110–112]

[MSM09] Jens Maue, Peter Sanders, and Domagoj Matijevic. "Goal Directed Shortest Path Queries Using Precomputed Cluster Distances". In: *ACM Journal of Experimental Algorithmics* 14 (2009) . [see page 72]

[MSZ18] Jayakrishnan Madathil, Saket Saurabh, and Meirav Zehavi. "Max-Cut Above Spanning Tree Is Fixed-Parameter Tractable". In: *Computer Science - Theory and Applications - 13th International Computer Science Symposium in Russia, CSR 2018, Moscow, Russia, June 6-10, 2018, Proceedings*. 2018, pages 244–256. [see pages 54, 55]

[Mur+10] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. "Introducing the Graph 500". In: *Cray Users Group (CUG)* 19 (2010), pages 45–74 . [see page 104]

[Nic+19] Bogdan Nicolae, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Franck Cappello. "VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale". In: *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*. 2019, pages 911–920. DOI: 10.1109/IPDPS.2019.00099. [see pages 109–111]

[NT75] G.L. Nemhauser and Jr. Trotter L.E. "Vertex Packings: Structural Properties and Algorithms". In: *Mathematical Programming* 8.1 (1975), pages 232–248: Springer-Verlag. DOI: 10.1007/BF01580444. [see pages 11, 13, 17]

[Obe+17] Michael Obersteiner, Alfredo Parra-Hinojosa, Mario Heene, Hans-Joachim Bungartz, and Dirk Pflüger. "A Highly Scalable, Algorithm-Based Fault-Tolerant Solver for Gyrokinetic Plasma Simulations". In: *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA@SC 2017, Denver, CO, USA, November 13, 2017*. 2017, 2:1–2:8. DOI: 10.1145/3148226.3148229. [see page 110]

[Pag+99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The Pagerank Citation Ranking: Bringing Order to the Web.* Technical report. Stanford InfoLab, 1999. [see page 105]

[PD11] Steven J. Plimpton and Karen D. Devine. "MapReduce in MPI for Large-Scale Graph Algorithms". In: *Parallel Computing. Systems & Applications* 37.9 (2011), pages 610–632. DOI: 10.1016/j.parco.2011.02.004. [see pages 92, 104]

[PLP98] James S. Plank, Kai Li, and Michael A. Puening. "Diskless Checkpointing". In: *IEEE Transactions on Parallel and Distributed Systems* 9.10 (1998), pages 972–986. DOI: 10.1109/71.730527. [see page 111]

[Pri05] Elena Prieto. "The Method of Extremal Structure on the k-Maximum Cut Problem". In: *Theory of Computing 2005, Eleventh CATS 2005, Computing: The Australasian Theory Symposium, Newcastle, NSW, Australia, January/February 2005*. 2005, pages 119–126. [see page 54]

[PT19] Patrick Prosser and James Trimble. *Peaty: an Exact Solver for the Vertex Cover Problem*. May 2019. DOI: 10.5281/zenodo.3082356. [see page 21]

[Put+15] Deepak Puthal, Surya Nepal, Cécile Paris, Rajiv Ranjan, and Jinjun Chen. "Efficient Algorithms for Social Network Coverage and Reach". In: *IEEE International Congress on Big Data*. 2015, pages 467–474. DOI: 10.1109/BigDataCongress.2015.75. [see page 9]

[PvdG21] Rick Plachetta and Alexander van der Grinten. "SAT-And-Reduce for Vertex Cover: Accelerating Branch-And-Reduce by SAT Solving". In: *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2021, pages 169–180. [see page 28]

[RA15] Ryan A Rossi and Nesreen K Ahmed. "The Network Data Repository with Interactive Graph Analytics and Visualization". In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. 2015, pages 4292–4293. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9553/9856. [see pages 1, 38, 39, 63, 66]

[RR21] Payas Rajan and Chinya V. Ravishankar. "Tiering in Contraction and Edge Hierarchies for Stochastic Route Planning". In: *29th International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*. 2021, pages 616–625. DOI: 10.1145/3474717.3484267. [see page 83]

[RRW10] Franz Rendl, Giovanni Rinaldi, and Angelika Wiegele. "Solving Max-Cut to Optimality by Intersecting Semidefinite and Polyhedral Relaxations". In: *Mathematical Programming* 121.2 (2010), page 307. DOI: 10.1007/s10107-008-0235-8. [see pages 52, 62]

[RS60] I. S. Reed and G. Solomon. "Polynomial Codes Over Certain Finite Fields". In: *Journal of the Society for Industrial and Applied Mathematics* 8.2 (June 1960), pages 300–304: Society for Industrial & Applied Mathematics (SIAM). DOI: 10.1137/0108018. [see page 115]

[San+08] Pedro V. Sander, Diego Nehab, Eden Chlamtac, and Hugues Hoppe. "Efficient Traversal of Mesh Edges Using Adjacency Primitives". In: *ACM Transactions on Graphics* 27.5 (2008), page 144. DOI: 10.1145/1409060.1409097. [see page 9]

[San20] Peter Sanders. “Connecting MapReduce Computations to Realistic Machine Models”. In: *2020 IEEE International Conference on Big Data (Big Data)*. IEEE. 2020, pages 84–93. [see pages 92, 95, 96, 107]

[Sch13] Sebastian Schlag. “Distributed Duplicate Removal”. Master’s thesis. Karlsruhe Institute of Technology, 2013. [see pages 105, 107]

[Sch20] Christian Schorr. “Improved Branching Strategies for Maximum Independent Sets”. Bachelor’s thesis. Karlsruhe Institute of Technology, 2020. [see page 133]

[SDM10] John Shalf, Sudip S. Dosanjh, and John Morrison. “Exascale Computing Technology Challenges”. In: *High Performance Computing for Computational Science - VECPAR 2010 - 9th International conference, Berkeley, CA, USA, June 22-25, 2010, Revised Selected Papers*. 2010, pages 1–25. DOI: 10.1007/978-3-642-19328-6_1. [see pages 1, 87]

[Seg+13] Pablo San Segundo, Fernando Matía, Diego Rodríguez-Losada, and Miguel Hernando. “An Improved Bit Parallel Exact Maximum Clique Algorithm”. English. In: *Optimization Letters* 7.3 (2013), pages 467–479: Springer-Verlag. DOI: 10.1007/s11590-011-0431-y. [see page 11]

[Sha+19] Faisal Shahzad, Jonas Thies, Moritz Kreutzer, Thomas Zeiser, Georg Hager, and Gerhard Wellein. “CRAFT: A Library for Easier Application-Level Checkpoint/restart and Automatic Fault Tolerance”. In: *IEEE Transactions on Parallel and Distributed Systems* 30.3 (2019), pages 501–514. DOI: 10.1109/TPDS.2018.2866794. [see pages 109–111]

[Shv+10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. “The Hadoop Distributed File System”. In: *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee. 2010, pages 1–10. [see pages 92, 111]

[SLP16] Pablo San Segundo, Alvaro Lopez, and Panos M. Pardalos. “A New Exact Maximum Clique Algorithm for Large and Massive Sparse Graphs”. In: *Computers & Operations Research* 66 (2016), pages 81–94. DOI: 10.1016/j.cor.2015.07.013. [see page 12]

[Sni+14] Marc Snir, Robert W. Wisniewski, Jacob A. Abraham, Sarita V. Adve, Saurabh Bagchi, Pavan Balaji, James F. Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A. Chien, Paul Coteus, Nathan DeBardeleben, Pedro C. Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd S. Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. “Addressing Failures in Exascale Computing”. In: *International Journal of High Performance Computing Applications* 28.2 (2014), pages 129–173. DOI: 10.1177/1094342014522573. [see pages 1, 87]

[SRJ11] Pablo San Segundo, Diego Rodríguez-Losada, and Agustín Jiménez. "An Exact Bit-Parallel Algorithm for the Maximum Clique Problem". In: *Computers & Operations Research and their Application to Problems of World Concern* 38.2 (2011), pages 571–581. DOI: 10.1016/j.cor.2010.07.019. [see page 11]

[SS05] Peter Sanders and Dominik Schultes. "Highway Hierarchies Hasten Exact Shortest Path Queries". In: *13th Annual European Symposium on Algorithms (ESA)*. Springer. 2005, pages 568–579. [see pages 70, 80]

[SS07] Dominik Schultes and Peter Sanders. "Dynamic Highway-Node Routing". In: *6th Workshop on Experimental Algorithms (WEA)*. 2007, pages 66–79. [see page 70]

[SS13] Peter Sanders and Christian Schulz. "Think Locally, Act Globally: Highly Balanced Graph Partitioning". In: *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*. 2013, pages 164–175. [see page 38]

[SS16] Peter Sanders and Christian Schulz. "Scalable Generation of Scale-Free Graphs". In: *Information Processing Letters* 116.7 (2016), pages 489–491. DOI: 10.1016/j.ipl.2016.02.004. [see page 62]

[SS23] Peter Sanders and Matthias Schimek. "Engineering Massively Parallel MST Algorithms". In: *CoRR* abs/2302.12199 (2023). DOI: 10.48550/arXiv.2302.12199. arXiv: 2302.12199. [see pages 105, 107]

[ST02] Peter Sanders and Jesper Larsson Träff. "The Hierarchical Factor Algorithm for All-To-All Communication (Research Note)". In: *Euro-Par 2002, Parallel Processing, 8th International Euro-Par Conference Paderborn, Germany, August 27-30, 2002, Proceedings*. 2002, pages 799–804. DOI: 10.1007/3-540-45706-2_112. [see page 107]

[ST14] Pablo San Segundo and Cristóbal Tapia. "Relaxed Approximate Coloring in Exact Maximum Clique Search". In: *Computers & Operations Research and their Application to Problems of World Concern* 44 (2014), pages 185–192. DOI: 10.1016/j.cor.2013.10.018. [see page 29]

[Ste23] Steinbuch Centre for Computing. *Horeka*. online. [Online; accessed 10-May-2023]. 2023. URL: https://www.scc.kit.edu/dienste/horeka.php. [see page 1]

[Str16] Darren Strash. "On the Power of Simple Reductions for the Maximum Independent Set Problem". In: *Proc. 22nd International Computing and Combinatorics Conference (COCOON 2016)*. Volume 9797. LNCS. Springer, 2016, pages 345–356. DOI: 10.1007/978-3-319-42634-1_28. [see pages 12, 18]

[SU23] Peter Sanders and Tim Niklas Uhl. "Engineering a Distributed-Memory Triangle Counting Algorithm". In: *CoRR* abs/2302.11443 (2023). DOI: 10.48550/arXiv.2302.11443. arXiv: 2302.11443. [see pages 105, 107]

[SWZ02] Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. "Using Multi-Level Graphs for Timetable Information". In: *4th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2002, pages 43–59. [see page 70]

[SZ18] Sándor Szabó and Bogdán Zaválnij. "A Different Approach to Maximum Clique Search". In: *2018 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. Sept. 2018, pages 310–316. DOI: 10.1109/SYNASC.2018.00055. [see page 22]

[TH14] Keita Teranishi and Michael A. Heroux. "Toward Local Failure Local Recovery Resilience Model Using MPI-ULFM". In: *21st European MPI Users' Group Meeting, EuroMPI/ASIA '14, Kyoto, Japan - September 09 - 12, 2014*. 2014, page 51. DOI: 10.1145/2642769.2642774. [see page 110]

[Tom+10] Etsuji Tomita, Yoichi Sutani, Takanori Higashi, Shinya Takahashi, and Mitsuo Wakatsuki. "A Simple and Faster Branch-And-Bound Algorithm for Finding a Maximum Clique". English. In: *Algorithms and Computation (WALCOM'10)*. Edited by Md. Saidur Rahman and Satoshi Fujita. Volume 5942. LNCS. Springer Berlin Heidelberg, 2010, pages 191–203. DOI: 10.1007/978-3-642-11440-3_18. [see page 11]

[Tom+13] Etsuji Tomita, Yoichi Sutani, Takanori Higashi, and Mitsuo Wakatsuki. "A Simple and Faster Branch-And-Bound Algorithm for Finding a Maximum Clique with Computational Experiments". In: *IEICE Transactions on Information and Systems* 96-D.6 (2013), pages 1286–1298. DOI: 10.1587/transinf.E96.D.1286. [see page 29]

[TT77] Robert Endre Tarjan and Anthony E. Trojanowski. "Finding a Maximum Independent Set". In: *SIAM Journal on Computing* 6.3 (1977), pages 537–546. DOI: 10.1137/0206038. [see page 11]

[Uga+11] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. "The Anatomy of the Facebook Social Graph". In: *CoRR* abs/1111.4503 (2011). arXiv: 1111.4503. [see pages 1, 7]

[Val90] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In: *Communications of the ACM* 33.8 (1990), pages 103–111. DOI: 10.1145/79173.79181. [see page 92]

[VBB15] Anurag Verma, Austin Buchanan, and Sergiy Butenko. "Solving the Maximum Clique and Vertex Coloring Problems on Very Large Sparse Networks". In: *INFORMS Journal on Computing* 27.1 (2015), pages 164–177. DOI: 10.1287/ijoc.2014.0618. [see page 12]

[VM97] M. Vijay and R. Mittal. "Algorithm-Based Fault Tolerance: A Review". In: *Microprocess. Microsystems* 21.3 (1997), pages 151–161. DOI: 10.1016/S0141-9331(97)00029-X. [see page 110]

[VT93] Nguyen Van Ngoc and Zsolt Tuza. "Linear-Time Approximation Algorithms for the Max Cut Problem". In: *Combinatorics, Probability Comput.* 2.2 (1993), pages 201–210: Cambridge University Press. DOI: 10.1017/S0963548300000596. [see page 54]

[Wan+13] Yang Wang, Zhipeng Lü, Fred Glover, and Jin-Kao Hao. "Probabilistic GRASP-Tabu Search Algorithms for the UBQP Problem". In: *Computers & Operations Research* 40.12 (2013), pages 3100–3107. DOI: 10.1016/j.cor.2011.12.006. [see page 52]

[WH15] Qinghua Wu and Jin-Kao Hao. "A Review on Algorithms for Maximum Clique Problems". In: *European Journal of Operational Research* 242.3 (2015), pages 693–709. DOI: 10.1016/j.ejor.2014.09.064. [see page 12]

[Wie18] Angelika Wiegele. *BiqMac Library.* [Online; accessed 2-September-2018]. 2018. URL: http://biqmac.aau.at/biqmaclib.html. [see page 63]

[XGA13] Jingen Xiang, Cong Guo, and Ashraf Aboulnaga. "Scalable Maximum Clique Computation Using MapReduce". In: *Proc. IEEE 29th International Conference on Data Engineering (ICDE'13).* Apr. 2013, pages 74–85. DOI: 10.1109/ICDE.2013.6544815. [see page 12]

[Xin+13] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. "Graphx: A Resilient Distributed Graph System on Spark". In: *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-located with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013.* 2013, page 2. DOI: 10.1145/2484425.2484427. [see page 3]

[XN13] Mingyu Xiao and Hiroshi Nagamochi. "Confining Sets and Avoiding Bottleneck Cases: A Simple Maximum Independent Set Algorithm in Degree-3 Graphs". In: *Theoretical Computer Science* 469 (2013), pages 92–104. DOI: 10.1016/j.tcs.2012.09.022. [see pages 11, 14, 17, 33–36]

[XN17] Mingyu Xiao and Hiroshi Nagamochi. "Exact Algorithms for Maximum Independent Set". In: *Information and Computation* 255 (2017), pages 126–146. DOI: 10.1016/j.ic.2017.06.001. [see pages 9, 11, 28]

[Zah+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012.* 2012, pages 15–28.

URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia. [see pages 3, 91, 92, 111]

[Zav19] Bogdan Zavalnij. *Zbogdan/pace-2019 A*. May 2019. DOI: 10.5281/zenodo.3228802. [see page 21]