Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
2015

On Efficiency and Reliability in Computer Science

A dissertation submitted towards the degree Doctor of Natural Sciences (Dr. rer. nat.) of the Faculties of Natural Sciences and Technology of Saarland University

submitted by

Adrian Neumann

Date of the defense: 19. June 2015

Dean of the Faculty

Prof. Dr. Markus Bläser

Examination Board

Prof. Dr. Dr. h.c. mult. Reinhard Wilhelm (Vorsitzender)
Prof. Dr. Dr. h.c. mult. Kurt Mehlhorn
Dr. Andreas Wiese
Dr. Mayank Goswami (Beisitzender)

Contents

Acknowledgments

This thesis would not have been possible without the great help of my advisors and my fellow researchers. Here I want to thank especially Prof. Kurt Mehlhorn for meeting with me week in and week out and always asking the right questions whenever I was stuck on a problem. I also want to thank my co-authors for, true to the nature of modern science, most of these results, and indeed some of the problems, were found during lively discussions. Last but not least I want to thank Andreas Schmidt for diligently reading this thesis and pointing out many mistakes and suggesting many improvements.

Declaration of original authorship

I hereby declare that this dissertation is my own original work except where otherwise indicated. All data or concepts drawn directly or indirectly from other sources have been correctly acknowledged.

This dissertation has not been submitted in its present or similar form to any other academic institution either in Germany or abroad for the award of any other degree.

_____

Adrian Neumann

Zusammenfassung

Effizienz von Algorithmen und Zuverlässigkeit gegen Fehlern in ihrer Implementierung oder Unsicherheiten in der Eingabe ist in der Informatik von großem Interesse. Diese Dissertation präsentiert Ergebnisse für Probleme in diesem Themenfeld.

Zertifizierende Algorithmen ermöglichen zuverlässige Implementierungen durch Berechnung eines Zertifikats für ihre Antworten. Ein einfaches Programm kann die Antworten mit den Zertifikaten überprüfen. Der Nutzer muss nur dem einfachen Programms vertrauen. Wir präsentieren einen neuen zertifizierenden Algorithmus für 3-Kantenzusammenhang und einen vereinfachten zertifizierenden Algorithmus für 3-Knotenzusammenhang.

Den Zustand einer Berechnung gelegentlich zu speichern, sog. *Checkpointing*, verbessert die Zuverlässigkeit. Im Fehlerfall kann ein gespeicherter Zustand wiederhergestellt werden ohne die Berechnung neu zu beginnen. Wir zeigen Strategien für Checkpointing mit begrenztem Speicher, die die Neuberechnungszeit minimieren.

Traditionell sind die Eingaben für Probleme präzise und wohldefiniert. In der Praxis beinhalten die Eingaben allerdings Unsicherheiten und man braucht robuste

Lösungen. Wir betrachten eine robuste Variante des k-median Problem. Hier sind die Kunden in Gruppen eingeteilt und wir möchten die Kosten der teuersten Gruppe minimieren. Dies macht die Lösung robust gegenüber welche der Gruppen letztlich bedient werden soll. Wir zeigen, dass dieses Problem schwer zu approximieren ist und untersuchen Heuristiken.

Abstract

Efficiency of algorithms and robustness against mistakes in their implementation or uncertainties in their input has always been of central interest in computer science. This thesis presents results for a number of problems related to this topic.

Certifying algorithms enable reliable implementations by providing a certificate with their answer. A simple program can check the answers using the certificates. If the the checker accepts, the answer of the complex program is correct. The user only has to trust the simple checker. We present a novel certifying algorithm for 3-edge-connectivity as well as a simplified certifying algorithm for 3-vertex-connectivity.

Occasionally storing the state of computations, so called *checkpointing*, also helps with reliability since we can recover from errors without having to restart the computation. In this thesis we show how to do checkpointing with bounded memory and present several strategies to minimize the worst-case recomputation.

In theory, the input for problems is accurate and well-defined. However, in practice it often contains uncertainties necessitating robust solutions. We consider a robust variant of the well known k-median problem, where the clients are grouped into sets. We want to minimize the connection cost of the expensive group. This solution is robust against which group we actually need to serve. We show that this problem is hard to approximate, even on the line, and evaluate heuristic solutions.

Chapter 1

Introduction

Since the first electronic computers were invented there has been an astonishing amount of progress. The first machines were slow behemoths that filled buildings, required kilowatts of power and a large crew of technicians working relentlessly to keep them running. Nowadays much more powerful machines fit in every pocket, run for days from a small battery and require no maintenance at all. Due to these developments from 1986 to 2007 the total computational power available to humanity has increased by four orders of magnitude [38]. Going even further back, our ability for calculation has increased by fifteen orders of magnitude from a clerk in the nineteenth century to the modern supercomputers we have today [58].

The hardware that we use made tremendous strides in both efficiency and reliability. Similarly, the algorithms we use are much more efficient today than they were at the dawn of the computer age. Algorithmic improvements are harder to quantify, since often not just the speed improvement, but the quality of results as well. Progress in numerical algorithms is relatively easy to quantify. For example, in the field of optimization, the speed-up from algorithmic improvements equals or even exceeds the speed-up from improved hardware [12]. In other areas, like artificial intelligence, progress is harder to put into numbers. For chess software, it took five decades of effort from the early attempts of Shannon [67] and Turing [74] until the special-purpose supercomputer Deep Blue beat the reigning world champion Kasparov in 1997 [24]. In 2009, Pocket Fritz 4 achieved a grand master ranking even though it ran on a handheld computer [16].

The reliability of computers and algorithms also plays a big role in computer science and computer engineering. The hardware we use has become a lot more reliable as technology progressed. The earliest computers that used vacuum tubes instead of transistors suffered from frequent failures as the tubes burnt out. ENIAC, the first general purpose digital computer, had a failure on average every two days [1], whereas the typical user today doesn't experience any hardware failures over the useful lifetime of a machine. Nevertheless, the number of machines in use has increased so much, that even rare failures occur regularly in large datacenters, with multiple machines failing each day [23]. Similarly, the tools we have for writing software have improved tremendously, but writing security critical software is still extremely expensive. Techniques to ameliorate failures in hardware and mistakes in programs have therefore been of interest from the beginnings of the field.

In this thesis we consider a number of problems in computer science related to efficiency and reliability. In the first chapter is connected to the reliability of programmers. We discuss 3-connectivity of graphs and present an efficient certifying algorithm for 3-node- and 3-edge-connectivity. Certifying algorithms are designed such that the user only has to trust a very simple program to be sure that the answers of the program that implements the actual algorithm are correct.

The second chapter introduces a different kind of reliability problem. We examine how to store the state of a long running computation if memory is limited. This technique is called *checkpointing* and has applications in error recovery after hardware failures but is also useful in "regular" computation, for example compression.

The last chapter considers an optimization problem. Here we want to open a set of facilities to cheaply serve clients. This is a classic problem in the field of optimization, called *facility location*. Typically it is assumed that the locations of the clients are known inputs to the algorithm. In our setting, we're not so sure about these locations. Instead we consider the setting where we have a set of possible

client locations and want to find a solution that is not too bad, regardless of which of these positions are realized.

Chapter 2

Certifying Algorithms for 3-Connectivity

2.1 Introduction

Advanced graph algorithms answer complex yes-no questions such as "Is this graph planar?" or "Is this graph k-vertex-connected?". These algorithms are not only nontrivial to implement, it is also difficult to test their implementations extensively, as usually only small test sets are available. It is hence possible that bugs persist unrecognized for a long time. An example is the implementation of the linear time planarity test of Hopcroft and Tarjan [39] in LEDA [53, 52]. A bug in the implementation was discovered only after two years of intensive use.

Certifying algorithms [50] approach this problem by computing an additional *certificate* that proves the correctness of the answer. This may, e.g., be either a 2-coloring or an odd cycle for testing bipartiteness, or either a planar embedding or a Kuratowski subgraph for testing planarity. Certifying algorithms are designed such that checking the correctness of the certificate is substantially simpler than solving the original problem and can be accomplished by a short and easy to understand program.

Since the checker is comparably simple, it is easy to be reasonably sure of its correctness. Then, unlike for normal algorithms, where testcases have to carefully constructed so that the correct solution is known beforehand, for a certifying algorithm *any* input can serve as a testcase, since the checker verifies that the output is correct.

Ideally, checking the correctness is so simple that a formal verification of the checker is feasible. In that case, the solution of *every* instance that passes the checker is *correct by a formal proof*. This level of confidence has already been achieved for a number of checkers in LEDA [3].

The notion of verifying the output after running an algorithm has a long history. The method of casting out nines to (partially) verify the result of arithmetic has been known to the ancient Greeks and has even older roots [36]. Certifying algorithms can be viewed as a special case of *N-version programming* [21]. In N-version programming an instance of a problem is solved by multiple independently implemented algorithms and only if their answers agree are they accepted. This of course requires extra resources. A certifying algorithm can be seen as a pair of independent algorithms. Here the first algorithm helps the other along by producing the certificate. Using the certificate, a simpler and faster program, the checker, can be used instead of an alternative implementation of a full-blown algorithm. This notion was first explored in [69].

Independently, Blum [13, 14] introduced the concept of *program checkers*. Blum's program checkers check correctness probabilistically, for example by using results from the field of interactive proofs, and receives no help in form of additional output from the program. Hence these checkers are independent of the algorithm that solves the original problem. To force the checkers to be simpler to verify than the original program, Blum requires them to run asymptotically faster than the original program.

Certifying algorithms as we understand them today were first used in the context of the LEDA project. Initially the term *Program Checking* or *Result Checking* was used. In [43] the term "Certifying Algorithm" was used for the first time.

The main result of this chapter is a linear time certifying algorithm for 3-edge-connectivity that uses a result of Mader [49] for the certificate. This is joint work

with Kurt Mehlhorn and Jens M. Schmidt [54]. We also show a certifying algorithm for 3-vertex-connectivity. For this problem a linear time algorithm was presented in [65]. Our algorithm does not run in linear time, but it is considerably easier prove its correctness.

Mader showed that every 3-edge-connected graph can be obtained from K_2^3, the graph consisting of two vertices and three parallel edges, by a sequence of three simple operations that each introduce one edge and, trivially, preserve 3-edge-connectivity. We show how to compute such a sequence in linear time for 3-edge-connected graphs. If the input graph is not 3-edge-connected, a 2-edge-cut is computed. The previous algorithms [31, 55, 70, 72, 73] for deciding 3-edge-connectivity are not certifying; they deliver a 2-edge-cut for graphs that are not 3-edge-connected but no certificate in the yes-case.

Our algorithm uses the concept of a *chain decomposition* of a graph introduced in [64]. A chain decomposition is a special ear decomposition [47]. It is used in [63] as a common and simple framework for certifying 1- and 2-vertex, as well as 2-edge-connectivity. Further, [65] uses them for certifying 3-vertex-connectivity. Chain decompositions are an example of *path-based* algorithms (see, e.g., Gabow [30]), which use only the simple structure of certain paths in a DFS-tree to compute connectivity information about the graph.

We use chain decompositions to certify 3-edge-connectivity in linear time. Thus, chain decompositions form a common framework for certifying k-vertex- and k-edge-connectivity for $k \leq 3$ in linear time. We use many techniques from [65], but in a simpler form. Hence this description may also be used as a gentle introduction to the 3-vertex-connectivity algorithm in [65].

We state Mader's result in Section 2.3 and introduce chain decompositions in Section 2.4. In Section 2.5 we show that chain decompositions can be used as a basis for Mader's construction. This immediately leads to an $O((m+n)\log(m+n))$ certifying algorithm (Section 2.6). The linear time algorithm is then presented in Sections 2.7 and 2.8. In Section 2.9 we discuss the verification of Mader construction sequences.

The mincuts in a graph can be represented succinctly by a cactus representation [26, 56, 29]; see Section 2.10. The 3-edge-connected components of a graph are the maximal subsets of the vertex set such that any two vertices in the subset are connected by three edge-disjoint paths. These paths are not necessarily contained in the subset.

Our algorithm can be used to turn any algorithm for computing 3-edge-connected components into a certifying algorithm for computing 3-edge-connected components and the cactus representation of 2-cuts (Section 2.10). An extension of our algorithm computes the 3-edge-connected components and the cactus representation directly (Section 2.11). A similar technique can be used to extend the 3-vertex-connectivity algorithm in [65] to an algorithm for computing 3-vertex-connected components.

In Section 2.13 we show how ideas from Section 2.6 can be used to dramatically simplify the certifying 3-vertex-connectivity algorithm from [65]. The simplified algorithm no longer runs in linear time, but the its proof of correctness is much shorter.

2.2 Related Work

Deciding 3-edge-connectivity is a well studied problem, with applications in diverse fields such as bioinformatics [25] and quantum chemistry [22]. Consequently, there are many different linear time solutions known [31, 55, 70, 72, 73, 56]. None of them is certifying. All but the first algorithm also compute the 3-edge-connected components. The cactus representation of a 2-edge-connected, but not 3-edge-connected graph G, can be obtained from G by repeatedly contracting the 3-edge-connected components to single vertices [56].

The paper [50] is a recent survey on certifying algorithms. For a linear time certifying algorithm for 3-vertex-connectivity, see [65] (implemented in [57]). For general k, there is a randomized certifying algorithm for k-vertex connectivity in [46] with expected running time $O(kn^{2.5} + nk^{3.5})$. There is a non-certifying algorithm [41] for deciding k-edge-connectivity in time $O(m \log^3 n)$ with high probability.

In [31], a linear time algorithm is described that transforms a graph G into a graph G' such that G is 3-edge-connected if and only if G' is 3-vertex-connected. Combined with this transformation, the certifying 3-vertex-connectivity algorithm from [65] certifies 3-edge-connectivity in linear time. However, that algorithm is much more complex than the algorithm given here. Moreover, we were unable to find an elegant method for transforming the certificate obtained for the 3-vertex-connectivity of G' into a certificate for 3-edge-connectivity of G.

2.3 Preliminaries

We consider finite undirected graphs G with $n = |V(G)|$ vertices, $m = |E(G)|$ edges, no self-loops, and minimum degree three, and use standard graph-theoretic terminology from [15], unless stated otherwise. We use $u\,v$ to denote an edge with endpoints u and v.

A set of edges that leaves a disconnected graph upon deletion is called *edge cut*. For $k \geq 1$, let a graph G be *k-edge-connected* if $n \geq 2$ and there is no edge cut $X \subseteq E(G)$ with $|X| < k$. Let $v \to_G w$ denote a path P between two vertices v and w in G and let $s(P) = v$ and $t(P) = w$ be the source and target vertex of P, respectively (as G is undirected, the direction of P is given by $s(P)$ and $t(P)$). Every vertex in $P - \{s(P), t(P)\}$ is called an *inner vertex* of P and every vertex in P is said to *lie on* P.

Let T be an undirected tree rooted at vertex r. For two vertices x and y in T, x is an *ancestor* of y and y is a *descendant* of x if $x \in V(r \to_T y)$, where $V(r \to_T y)$ denotes the vertex set of the path from r to y in T. If additionally $x \neq y$, x is a *proper* ancestor and y is a *proper* descendant. We write $x \leq y$ ($x < y$) if x is an ancestor (proper ancestor) of y. The parent $p(v)$ of a vertex v is its immediate proper ancestor. The parent function is undefined for r. Let K_2^m be the graph on 2 vertices that contains exactly m parallel edges.

Let *subdividing an edge $u\,v$* of a graph G be the operation that replaces $u\,v$ with a path $u\,z\,v$, where z was not previously in G. All 3-edge-connected graphs can be constructed using a small set of operations starting from a K_2^3.

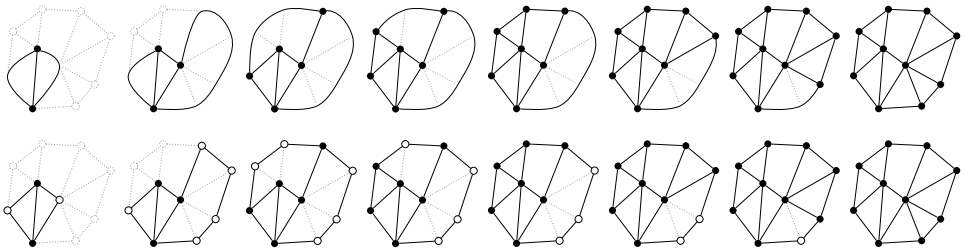


Figure 2.1: Two ways of constructing the 3-edge-connected graph shown in the rightmost column. The upper row shows the construction according to Theorem 2.1. The lower row shows the construction according to Corollary 2.2. Branch (non-branch) vertices are depicted as filled (non-filled) circles. The black edges exist already, while dotted gray vertices and edges do not exist yet.

Theorem 2.1 (Mader [49])**:** Every 3-edge-connected graph (and no other graph) can be constructed from a K_2^3 using the following three operations:

- Adding an edge (possibly parallel or a loop).
- Subdividing an edge $x\,y$ and connecting the new vertex to any existing vertex.
- Subdividing two distinct edges $w\,x$, $y\,z$ and connecting the two new vertices.

A subdivision G' of a graph G is a graph obtained by subdividing edges of G zero or more times. The *branch vertices* of a subdivision are the vertices with degree at least three (we call the other vertices *non-branch*-vertices) and the *links* of a subdivision are the maximal paths whose inner vertices have degree two. If G has no vertex of degree two, the links of G' are in one-to-one correspondence to the edges of G. Theorem 2.1 readily generalizes to subdivisions of 3-edge-connected graphs.

Corollary 2.2: Every subdivision of a 3-edge-connected graph (and no other graph) can be constructed from a subdivision of a K_2^3 using the following three operations:

- Adding a path connecting two branch vertices.
- Adding a path connecting a branch vertex and a non-branch vertex.
- Adding a path connecting two non-branch vertices lying on distinct links.

In all three cases, the inner vertices of the path added are new vertices.

Each path that is added to a graph H in the process of Corollary 2.2 is called a *Mader-path* (*with respect to H*). Note that an ear is always a Mader-path unless both endpoints lie on the same link.

Figure 2.1 shows two constructions of a 3-edge-connected graph, one according to Theorem 2.1 and one according to Corollary 2.2. In this paper, we show how to find the Mader construction sequence according to Corollary 2.2 for a 3-edge-connected graph in linear time. Such a construction can be easily turned into one according to Theorem 2.1.

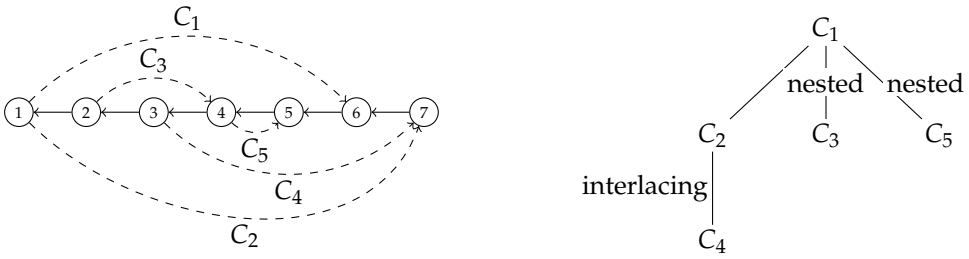


Figure 2.2: The left side of the figure shows a DFS tree with root 1 and a possible chain decomposition; tree-edges are solid and back-edges are dashed. C_1 is (1 6, 6 5, 5 4, 4 3, 3 2, 2 1), C_2 is (1 7, 7 6), C_3 is (2 4), C_4 is (3 7), and C_5 is (4 5). C_3 and C_5 are nested children of C_1 and C_4 is an interlacing child of C_2. Also, $s(C_4)$ s-belongs to C_1.

2.4 Chain Decompositions

We use a very simple decomposition of graphs into cycles and paths. The decomposition was previously used for linear-time tests of 2-vertex- and 2-edge-connectivity [63] and 3-vertex-connectivity [65]. In this paper we show that it can also be used to find Mader's construction for a 3-edge-connected graph. We define the decomposition algorithmically; a similar procedure that serves for the computation of low-points can be found in [61].

Let G be a connected graph without self-loops and let T be a depth-first search tree of G. Let r be the root of T. We orient tree-edges towards the root and back-edges away from the root, i.e., $v < u$ for an oriented tree-edge $u\,v$ and $x < y$ for an oriented back-edge $x\,y$. Note that this is exactly opposite to the usual orientation.

We decompose$G$ into a set $\mathcal{C} = \{C_1, \ldots, C_{|\mathcal{C}|}\}$ of cycles and paths, called *chains*, by applying the following procedure for each vertex v in the order in which they were discovered during the DFS: First, we declare v visited (initially, no vertex is visited), if not already visited before. Then, for every back-edge $v\,w$, we traverse the path $w \to_T r$ until a vertex x is encountered that was visited before; x is a descendant of v. The traversed subgraph $v\,w \cup (w \to_T x)$ forms a new *chain* C with $s(C) = v$ and $t(C) = x$. All inner vertices of C are declared visited. Observe that $s(C)$ and $t(C)$ are already visited when the construction of the chain starts.

Figure 2.2 illustrates these definitions. Since every back-edge defines one chain, there are precisely $m - n + 1$ chains. We number the chains in the order of their construction.

We call $\mathcal{C}$ a *chain decomposition*. It can be computed in time $O(n + m)$. For 2-edge-connected graphs the term decomposition is justified by Lemma 2.3.

Lemma 2.3 ([63])**:** Let $\mathcal{C}$ be a chain decomposition of a graph G. Then G is 2-edge-connected if and only if G is connected and the chains in $\mathcal{C}$ partition $E(G)$.

Since the condition of Lemma 2.3 is easily checked during the chain decomposition, we assume from now on that G is 2-edge-connected. Then $\mathcal{C}$ partitions $E(G)$ and the first chain C_1 is a cycle containing r (since there is a back-edge incident to r). We

say that *r strongly belongs (s-belongs)* to the first chain and any vertex $v \neq r$ *s-belongs* to the chain containing the edge $v\,p(v)$. We use s-belongs instead of belongs since a vertex can belong to many chains when chains are viewed as sets of vertices.

We can now define a parent-tree on chains. The first chain C_1 is the root. For any chain $C \neq C_1$, let the *parent* $p(C)$ of C be the chain to which $t(C)$ s-belongs. We write $C \leq D$ ($C < D$) for chains C and D if C is an ancestor (proper ancestor) of D in the parent-tree on chains.

The following lemma summarizes important properties of chain decompositions.

Lemma 2.4: Let $\{C_1, \ldots, C_{m-n+1}\}$ be a chain decomposition of a 2-edge-connected graph G and let r be the root of the DFS-tree. Then

(1) For every chain C_i, $s(C_i) \leq t(C_i)$.

(2) Every chain C_i, $i \geq 2$, has a parent chain $p(C_i)$. We have $s(p(C_i)) \leq s(C_i)$ and $p(C_i) = C_j$ for some $j < i$.

(3) For $i \geq 2$: If $t(C_i) \neq r$, $t(p(C_i)) < t(C_i)$. If $t(C_i) = r$, $t(p(C_i)) = t(C_i)$.

(4) If $u \leq v$, u s-belongs to C, and v s-belongs to D then $C \leq D$.

(5) If $u \leq t(D)$ and u s-belongs to C, then $C \leq D$.

(6) For $i \geq 2$: $s(C_i)$ s-belongs to a chain C_j with $j < i$.

Proof. (1) to (3) follow from the discussion preceding the Lemma and the construction of the chains. We turn to (4). Consider two vertices u and v with $u \leq v$ and let u s-belong to C and let v s-belong to D. Then $C \leq D$, as the following simple induction on the length of the tree path from u to v shows. If $u = v$, $C = D$ by the definition of s-belongs. So assume u is a proper ancestor of v. Since v s-belongs to D, by definition $v \neq t(D)$ and $v\,p(v)$ is contained in D. Let D' be the chain to which $p(v)$ s-belongs. By induction hypothesis, $C \leq D'$. Also, either $D = D'$ (if $p(v)$ s-belongs to D) or $D' = p(D)$ (if $p(v) = t(D)$) and hence $p(v)$ s-belongs to $p(D)$. In either case $C \leq D$.

Claim (5) is an easy consequence of (4). If $t(D) = r$, $C = C_1$, and the claim follows. If $t(D) \neq r$, $t(D)$ s-belongs to $p(D)$. Thus, $C \leq p(D)$ by (4).

The final claim is certainly true for each C_i with $s(C_i) = r$. So assume $s(C_i) > r$ and let $y = p(s(C_i))$. Since G is 2-edge-connected, there is a back-edge $u\,v$ with $u \leq y$ and $s(C_i) \leq v$. It induces a chain C_k with $k < i$ and hence $s(C_i)y$ is contained in a chain C_j with $j \leq k$. □

2.5 Chains as Mader-paths

We show that, assuming that the input graph is 3-edge-connected, there are two chains that form a subdivision of a K_2^3, and that the other chains of the chain decomposition can be added one by one such that each chain is a Mader-path with respect to the union of the previously added chains. We will also show that chains can be added parent-first, i.e., when a chain is added, its parent was already added. In this way the current graph G_c consisting of the already added chains is always *parent-closed*. We will later show how to compute this ordering efficiently.

We will first give a simple $O((n+m)\log(n+m))$ algorithm and then a linear time algorithm.

Using the chain decomposition, we can identify a K_2^3 subdivision in the graph as follows. We may assume that the first two back-edges explored from r in the DFS have their other endpoint in the same subtree T' rooted at some child of r. The first chain C_1 forms a cycle. The vertices in $C_1 - r$ are then contained in T'. By assumption, the second chain is constructed by another back-edge that connects r with a vertex in T'. If there is no such back-edge, the tree edge connecting r and the root of T' and the back-edge from r into T' form a 2-edge cut. Let $x = t(C_2)$. Then $C_1 \cup C_2$ forms a K_2^3 subdivision with branch vertices r and x. The next lemma derives properties of parent-closed unions of chains.

Lemma 2.5: Let G_c be a parent-closed union of chains that contains C_1 and C_2. Then

(1) For any vertex $v \neq r$ of G_c, the edge $v\,p(v)$ is contained in G_c, i.e., the set of vertices of G_c is a parent-closed subset of the DFS-tree.

(2) $s(C)$ and $t(C)$ are branch vertices of G_c for every chain C contained in G_c.

(3) Let C be a chain that is not in G_c but a child of some chain in G_c. Then C is an ear with respect to G_c and the path $t(C) \to_T s(C)$ is contained in G_c. C is a Mader-path (i.e., the endpoints of C are not inner vertices of the same link of G_c) with respect to G_c if and only if there is a branch vertex on $t(C) \to_T s(C)$.

Proof. (1): Let $v \neq r$ be any vertex of G_c. Let C be a chain in G_c containing the vertex v. If C also contains $v\,p(v)$ we are done. Otherwise, $v = t(C)$ or $v = s(C)$. In the first case, v s-belongs to $p(C)$, in the second case v s-belongs to some $C' \leq C$ by Lemma 2.4.(4). Hence, by parent-closedness, $v\,p(v)$ is an edge of G_c.

(2): Let C be any chain in G_c. Since C_1 and C_2 form a K_2^3, r and $x = t(C_2)$ are branch vertices. If $s(C) \neq r$, the edge $s(C)\,p(s(C))$ is in G_c by (1), the back-edge $s(C)\,v$ inducing C is in G_c, and the path $v \to_T s(C)$ is in G_c by (1). Thus $s(C)$ has degree at least three. If $t(C) \notin \{r, x\}$, let $\widehat{C}$ be the chain to which $t(C)$ s-belongs, i.e. $\widehat{C}$ is the parent of C. As G_c is parent-closed $\widehat{C}$ is contained in G_c. By the definition of s-belongs, $t(C)$ has degree two on the chain $\widehat{C}$. Further, it has degree one on the chain C. Since chains are edge-disjoint, it has degree at least three in G_c.

(3) We first observe that $t(C)$ and $s(C)$ belong to G_c. For $t(C)$, this holds since $t(C)$ s-belongs to $p(C)$ and $p(C)$ is part of G_c by assumption. For $s(C)$, this follows from $s(C) \leq t(C)$ and (1). No inner vertex u of C belongs to G_c, because otherwise the edge $u\,p(u)$ would belong to G_c by (1), which implies that C would belong to G_c, as G_c is a union of chains. Thus C is an ear with respect to G_c, i.e., it is disjoint from G_c except for its endpoints. Moreover, the path $t(C) \to_T s(C)$ belongs to G_c by (1).

If there is no branch vertex on $t(C) \to_T s(C)$, the vertices $t(C)$ and $s(C)$ are inner vertices of the same link of G_c and hence C is not a Mader-path with respect to G_c. If there is a branch vertex on $t(C) \to_T s(C)$, the vertices $t(C)$ and $s(C)$ are inner vertices of two distinct links of G_c and hence C is a Mader-path with respect to G_c. □

We can now prove that chains can always be added in parent-first order.

Theorem 2.6: Let G be a graph and let G_c be a parent-closed union of chains such that no child of a chain $C \in G_c$ is a Mader-path with respect to G_c and there is at least one such chain. Then the extremal edges of every link of length at least two in G_c are a 2-cut in G.

Proof. Assume otherwise. Then there is a parent-closed union G_c of chains such that no child of a chain in G_c is a Mader-path with respect to G_c and there is at least one such chain outside of G_c, but for every link in G_c the extremal edges are not a cut in G.

Consider any link L of G_c. Since the extremal edges of L, that is, the edges in L that are incident to the end vertices of L, do not form a 2-cut, there is a path in $G - G_c$ connecting an inner vertex on L with a vertex that is either a branch vertex of G_c or a vertex on a link of G_c different from L. Let P be such a path of minimum length. By minimality, no inner vertex of P belongs to G_c. Note that P is a Mader-path with respect to G_c. We will show that at least one edge of P belongs to a chain C with $p(C) \in G_c$ and that C can be added, contradicting our choice of G_c.

Let a and b be the endpoints of P, and let z be the lowest common ancestor of all points in P. Since a DFS generates only tree- and back-edges, z lies on P. Since $z \leq a$ and the vertex set of G_c is a parent-closed subset of the DFS-tree, z belongs to G_c. Thus z cannot be an inner vertex of P and hence is equal to a or b. Assume w.l.o.g. that $z = a$. All vertices of P are descendants of a. We view P as oriented from a to b.

Since b is a vertex of G_c, the path $b \to_T a$ is part of G_c by Lemma 2.4 and hence no inner vertex of P lies on this path. Let $a\,v$ be the first edge on P. The vertex v must be a descendant of b as otherwise the path $v \to_P b$ would contain a cross-edge, i.e. an edge between different subtrees. Hence $a\,v$ is a back-edge. Let D be the chain that starts with the edge $a\,v$. D does not belong to G_c, as no edge of P belongs to G_c.

We claim that $t(D)$ is a proper descendant of b or D is a Mader-path with respect to G_c. Since v is a descendant of b and $t(D)$ is an ancestor of v, $t(D)$ is either a proper descendant of b, equal to b, or a proper ancestor of b. We consider each case separately.

If $t(D)$ were a proper ancestor of b the edge $b\,p(b)$ would belong to D and hence D would be part of G_c, contradicting our choice of P. If $t(D)$ is equal to b then D is a Mader-path with respect to G_c. This leaves the case that $t(D)$ is a proper descendant of b.

Let $y\,x$ be the last edge on the path $t(D) \to_T b$ that is not in G_c and let D^* be the chain containing $y\,x$. Then $D^* \leq D$ by Lemma 2.4.(5) (applied with $C = D^*$ and $u = y$) and hence $s(D^*) \leq s(D) \leq a$ by part (4) of the same lemma. Also $t(D^*) = x$. Since $x = t(D^*) \in G_c$, $p(D^*) \in G_c$.

As a and b are not inner vertices of the same link, the path $x = t(D^*) \to_T b \to_T a \to_T s(D^*)$ contains a branch vertex. Thus D^* is a Mader-path by Lemma 2.5. □

Corollary 2.7: If G is 3-edge-connected, chains can be greedily added in parent-first order.

Proof. If we reach a point where not all chains are added, but we can not proceed in a greedy fashion, by Theorem 2.6 we find a cut in G. □

2.6 A First Algorithm

Corollary 2.7 gives rise to an $O((n+m)\log(n+m))$ algorithm, the Greedy-Chain-Addition Algorithm.In addition to G, we maintain the following data structures:

- The current graph G_c. Each link is maintained as a doubly linked list of vertices. Observe that all inner vertices of a link lie on the same tree path and hence are numbered in decreasing order. The vertices in G are labeled *inactive*, *branch*, or *non-branch*. The vertices in $G - G_c$ are called *inactive*. Every non-branch vertex stores a pointer to the link on which it lies and a list of all chains incident to it and having the other endpoint as an inner vertex of the same link.
- A list $\mathcal{L}$ of addable chains. A chain is addable if it is a Mader-path with respect to the current graph.
- For each chain its list of children.

We initialize G_c to $C_1 \cup C_2$. It has three links, $t(C_2) \rightarrow_T r$, $r \rightarrow_{C_1} t(C_2)$, and $r \rightarrow_{C_2} t(C_2)$. We then iterate over the children of C_1 and C_2. For each child, we check in constant time whether its endpoints are inner vertices of the same link. If so, we associate the chain with the link by inserting it into the lists of both endpoints. If not, we add the chain to the list of addable chains. The initialization process takes time $O(n+m)$.

As long as the list of addable chains is non-empty, we add a chain, say C. Let u and v be the endpoints of C. We perform the following actions:

- If u is a non-branch vertex, we make it a branch vertex. This splits the link containing it and entails some processing of the chains having both endpoints on this link.
- If v is a non-branch vertex, we make it a branch vertex. This splits the link containing it, and entails some processing of the chains having both endpoints on this link.
- We add C as a new link to G_c.
- We process the children of C.

We next give the details for each action.

If u is a non-branch vertex, it becomes a branch vertex. Let L be the link of G_c containing u; L is split into links L_1 and L_2 and the set S of chains having both endpoints on L is split into sets S_1, S_2 and S_{add}, where S_i is the set of chains having both endpoints on L_i, $i = 1, 2$, and S_{add} is the set of chains that become addable (because they are incident to u or have one endpoint each in L_1 and L_2). We show that we can perform the split of L in time $O(1 + |S_{\text{add}}| + \min(|L_1| + |S_1|, |L_2| + |S_2|))$. We walk from both ends of L towards u in lockstep fashion. In each step we either move to the next vertex or consider one chain. Once we reach u we stop. Observe that this strategy guarantees the time bound claimed above.

When we consider a chain, we check whether we can move it to the set of addable chains. If so, we do it and delete the chain from the lists of both endpoints. Once, we have reached u, we split the list representing the link into two. The longer part

of the list retains its identity, for the shorter part we create a new list header and redirect all pointers of its elements.

Adding C to G_c is easy. We establish a list for the new link and let all inner vertices of C point to it. The inner vertices become active non-branch vertices.

Processing the children of C is also easy. For each child, we check whether both endpoints are inner vertices of C. If so, we insert the child into the list of its endpoints. If not, we add the child to the list of addable chains.

If $\mathcal{L}$ becomes empty, we stop. If all chains have been added, we have constructed a Mader sequence. If not all chains have been processed, there must be a link having at least one inner vertex. The first and the last edge of this link form a 2-edge-cut.

It remains to argue that the algorithm runs in time $O((n+m)\log(n+m))$. We only need to argue about the splitting process. We distribute the cost $O(1+|S_{\text{add}}|+\min(|L_1|+|S_1|,|L_2|+|S_2|))$ as follows: $O(1)$ is charged to the vertex that becomes a branch vertex. All such charges add up to $O(n)$. $O(|S_{\text{add}}|)$ is charged to the chains that become addable. All such charges add up to $O(m)$. $O(\min(|L_1|+|S_1|,|L_2|+|S_2|))$ is charged to the vertices and chains that define the minimum. We account for these charges with the following token scheme inspired by the analysis of the corresponding recurrence relation in [51].

Consider a link L with k chains having both endpoints on L. We maintain the invariant that each vertex and chain owns at least $\log(|L|+k)$ tokens. When a link is newly created we give $\log(n+m)$ tokens to each vertex of the link and to each chain having both endpoints on the link. In total we create $O((n+m)\log(n+m))$ tokens. Assume now that we split a link L with k chains into links L_1 and L_2 with k_1 and k_2 chains respectively. Then $\min(|L_1|+k_1,|L_2|+k_2) \leq (|L|+k)/2$ and hence we may take one token away from each vertex and chain of the sublink that is charged without violating the token invariant.

Theorem 2.8: The Greedy-Chain-Addition algorithm runs in time

$$O((n+m)\log(n+m)).$$

2.7 A Classification of Chains

When we add a chain in the Greedy-Chain-Addition algorithm, we also process its children. Children that do not have both endpoints as inner vertices of the chain can be added to the list of addable chains immediately. However, children that have both endpoints as inner vertices of the chain cannot be added immediately and need to be observed further until they become addable. We now make this distinction explicit by classifying chains into two types, interlacing and nested.

We classify the chains $\{C_3,\dots C_{m-n+1}\}$ into two types. Let C be a chain with parent $\widehat{C}=p(C)$. We distinguish two cases[1] for C.

- If $s(C)$ is an ancestor of $t(\widehat{C})$ and a descendant of $s(\widehat{C})$, C is *interlacing*. We have $s(\widehat{C}) \leq s(C) \leq t(\widehat{C}) \leq t(C)$.

[1] In [65], three types of chains are distinguished. What we call nested is called Type 1 there and what we call interlacing is split into Types 2 and 3 there. We do not need this finer distinction.

- If $s(C)$ is a proper descendant of $t(\widehat{C})$, C is *nested*. We have $s(\widehat{C}) \leq t(\widehat{C}) < s(C) \leq t(C)$ and $t(C) \rightarrow_T s(C)$ is contained in $\widehat{C}$.

These cases are exhaustive as the following argument shows. Let $s(\widehat{C})\, v$ be the first edge on $\widehat{C}$. By Lemma 2.4, $s(\widehat{C}) \leq s(C) \leq v$. We split the path $v \rightarrow_T s(\widehat{C})$ into two parts corresponding to the two cases above, namely $t(\widehat{C}) \rightarrow_T s(\widehat{C})$, and $(v \rightarrow_T t(\widehat{C}))\backslash t(\widehat{C})$. Depending on which of these paths $s(C)$ lies, it is classified as interlacing or nested.

The following simple observations are useful. For any chain $C \neq C_1$, $t(C)$ s-belongs to $\widehat{C}$. If C is nested, $s(C)$ and $t(C)$ s-belong to $\widehat{C}$. If C is interlacing, $s(C)$ s-belongs to a chain which is a proper ancestor of $\widehat{C}$ or $\widehat{C} = C_1$. The next lemma confirms that interlacing chains can be added once their parent belongs to G_c.

Lemma 2.9: Let G_c be a parent-closed union of chains that contains C_1 and C_2, let C be any chain contained in G_c, and let D be an interlacing child of C not contained in G_c. Then D is a Mader-path with respect to G_c.

Proof. We have already shown in Lemma 2.5 that D is an ear with respect to G_c, that the path $t(D) \rightarrow_T s(D)$ is part of G_c, and that $s(C)$ and $t(C)$ are branching vertices of G_c. Since D is interlacing, we have $s(C) \leq s(D) \leq t(C) \leq t(D)$. Thus $t(D) \rightarrow_T s(D)$ contains a branching vertex and hence D is a Mader-path by Lemma 2.5.(3). □

2.8 A Linear Time Algorithm

According to Lemma 2.9, interlacing chains whose parent belongs to the current graph are always Mader-paths and can be added. Nested chains have both endpoints on their parent chain and can only be added once the tree-path connecting its endpoints contains a branching point. Consider a chain nested in chain C_i. Which chains can help its addition by creating branching points on C_i? First, interlacing chains having their source on some C_j with $j \leq i$, and second, chains nested in C_i and their interlacing offspring having their source on C_i. Chains having their source on some C_j with $j > i$ cannot help because they have no endpoint on C_i. This observation shows that chains can be added in phases. In the i-th phase, we try to add all chains having their source vertex on C_i.

The overall structure of the linear-time algorithm is given in Algorithm 1. An implementation in Python is available at `https://github.com/adrianN/edge-connectivity`. The algorithm operates in phases and maintains a current graph G_c. Let C_1, C_2, $\ldots$, C_{m-n+1} be the chains of the chain decomposition in the order of creation. We initialize G_c to $C_1 \cup C_2$. In phase i, $i \in [1, m-n+1]$, we consider the i-th chain C_i and either add all chains C to G_c for which the source vertex $s(C)$ s-belongs to C_i to G_c or exhibit a 2-edge-cut. As already mentioned, chains are added parent-first and hence G_c is always parent-closed. We maintain the following invariant:

Invariant: After phase i, G_c consists of all chains for which the source vertex s-belongs to one of the chains C_1 to C_i.

Lemma 2.10: For all i, the current chain C_i is part of the current graph G_c at the beginning of phase i or the algorithm has exhibited a 2-edge-cut before phase i.

Algorithm 1: Certifying linear-time algorithm for 3-edge connectivity.

Input: $G = (V, E)$
Let $\{C_1, C_2, \ldots, C_{m-n+1}\}$ be a chain decomposition of G as described in Sect. 2.4;
Initialize G_c to $C_1 \cup C_2$;
for *i from 1 to* $m - n + 1$ **do**
    /* Phase i: add all chains whose source s-belongs to C_i */
    Group the chains C for which $s(C)$ s-belongs to C_i into segments;
    /* Part I of Phase i: add segments with interlacing root */
    Add all segments whose minimal chain is interlacing to G_c;
    /* Part II of Phase i: add segments with nested root */
    Either find an insertion order $S_1, \ldots, S_k$ on the segments having a nested minimal chain or exhibit a 2-edge-cut and stop;
    for *j from 1 to* k **do**
        Add the chains contained in S_j parent-first;

Proof. The initial current graph consists of chains C_1 and C_2 and hence the claim is true for the first and the second phase. Consider $i > 2$. The source vertex $s(C_i)$ s-belongs to a chain C_j with $j < i$ (Lemma 2.4.(6)) and hence C_i is added in phase j. □

The next lemma gives information about the chains for which the source vertex s-belongs to C_i. None of them belongs to G_c at the beginning of phase i (except for chain C_2 that belongs to G_c at the beginning of phase 1) and they form subtrees of the chain tree. Only the roots of these subtrees can be nested. All other chains are interlacing.

Lemma 2.11: Assume that the algorithm reaches phase i without exhibiting a 2-edge-cut. Let $C \neq C_2$ be a chain for which $s(C)$ s-belongs to C_i. Then C is not part of G_c at the beginning of phase i. Let D be any ancestor of C that is not in G_c. Then:

(1) $s(D)$ s-belongs to C_i.

(2) If D is nested, it is a child of C_i.

(3) If $p(D)$ is not part of the current graph, D is interlacing.

Proof. We use induction on i. Consider the i-th phase and let $C \neq C_2$ be chains whose source vertex $s(C)$ s-belongs to C_i. We first prove that C is not in G_c. This is obvious, since in the j-th phase we add exactly the chains whose source vertex s-belongs to C_j.

(1): Let D be any ancestor of C which is not part of G_c. By Lemma 2.4, we have $s(D) \leq s(C)$ and hence $s(D)$ belongs to C_j for some $j \leq i$. If $j < i$, D would have

been added in phase j, a contradiction to the assumption that D does not belong to G_c at the beginning of phase i.

(2): $s(D)$ s-belongs to C_i by (1). If D is nested, $s(D)$ and $t(D)$ s-belong to the same chain. Thus D is a child of C_i.

(3): If $p(D)$ is not part of the current graph, $p(D) \neq C_i$ by Lemma 2.10 and hence D is not a child of C_i. Hence by (2), D is interlacing. □

We can now define the segments with respect to C_i by means of an equivalence relation. Consider the set $\mathcal{S}$ of chains whose source vertex s-belongs to C_i. For a chain $C \in \mathcal{S}$, let C^* be the minimal ancestor of C that does not belong to G_c. Two chains C and D in $\mathcal{S}$ belong to the same segment if and only if $C^* = D^*$. In Figure 2.2 on page 12, if we start with $G_c = C_1 \cup C_2$, we form three segments in the first phase, namely $\{C_4\}$, $\{C_3\}$, and $\{C_5\}$. The first segment can be added according to Lemma 2.9. Then C_3 can be added and then C_5.

Consider any $C \in \mathcal{S}$. By part (1) of the preceding lemma either $p(C) \in \mathcal{S}$ or $p(C)$ is part of G_c. Moreover, C and $p(C)$ belong to the same segment in the first case. Thus segments correspond to subtrees in the chain tree. In any segment only the minimal chain can be nested by Lemma 2.11. If it is nested, it is a child of C_i (parts (2) and (3) of the preceding lemma). Since only the root of a segment may be a nested chain, once it is added to the current graph all other chains in the segment can be added in parent-first order by Lemma 2.9. All that remains is to find the proper ordering of the segments faster than in the previous section. We do so in Lemma 2.15. If no proper ordering exists, we exhibit a 2-edge-cut.

Lemma 2.12: All chains in a segment S can be added in parent-first order if its minimal chain can be added.

Proof. By Lemma 2.11 all but the minimal chain in a segment are interlacing. Thus the claim follows from Lemma 2.9. □

We come to part I of phase i, the addition of all segments whose minimal chain is interlacing. As a byproduct, we will also determine all segments with nested minimal chain. We iterate over all chains C whose source $s(C)$ s-belongs to C_i. For each such chain, we traverse the path C, $p(C)$, $p(p(C))$, ... until we reach a chain that belongs to G_c or is already marked (initially, all chains are unmarked). We now distinguish cases. If the last chain on the path is nested we mark all chains on the path with the nested chain. If we hit a marked chain we copy the marker to all chains in the path. Otherwise, i.e., all chains are interlacing and unmarked, we add all chains in the path to G_c in parent-first order, as these segments can be added according to Corollary 2.12.

It remains to compute a proper ordering of the segments in which the minimal chain is nested or to exhibit a 2-edge-cut. We do so in part II of phase i. For simplicity, we will say 'segment' instead of 'segment with nested minimal chain' from now on.

For a segment S let the *attachment points* of S be all vertices in S that are in G_c. Note that the attachment points must necessarily be endpoints of chains in S and hence adding the chains of S makes the attachment points branch vertices. Nested children C of C_i can be added if there are branch vertices on $t(C) \rightarrow_T s(C)$, therefore adding a segment can make it possible to add further segments.

Lemma 2.13: Let C be a nested child of C_i and let S be the segment containing C. The attachment points of S consist of $s(C)$, $t(C)$, and the vertices $s(D)$ of the other chains in the segment. All such points lie on the path $t(C) \to_T s(C)$ and hence on C_i.

Proof. Let D be any chain in S different from C. By Lemma 2.11, C is the minimal chain in S. Since S is a subtree of the chain tree, we have $C < D$ and hence by Lemma 2.4 $t(C) \leq t(D)$. Since none of the chains in S is part of G_c, parent-closedness implies that no vertex on the path $(t(D) \to_T t(C)) - t(C)$ belongs to G_c. In particular, either $t(D) = t(C)$ or $t(D)$ is not a vertex of G_c and hence not an attachment point of S. It remains to show $s(C) \leq s(D) \leq t(C)$. Since $C \leq D$, we have $s(C) \leq s(D)$ by Lemma 2.4. Since $s(D) \leq t(D)$ and $t(C) \leq t(D)$ we have either $s(D) \leq t(C) \leq t(D)$ or $t(C) < s(D) \leq t(D)$. In the former case, we are done. In the latter case, $s(D)$ is not a vertex of G_c by the preceding paragraph, a contradiction, since $s(D)$ s-belongs to C_i by Lemma 2.11. □

For a set of segments $S_1, \ldots, S_k$, let the *overlap graph* be the graph on the segments and a special vertex R for the branch vertices on C_i. In the overlap graph, there is an edge between R and a vertex S_i, if there are attachment points $a_1 \leq a_2$ of S_i such that there is a branch vertex on the tree path $a_2 \to_T a_1$. Further, between two vertices S_i and S_j there is an edge if there are attachment points a_1, a_2 in S_i and b_1, b_2 in S_j, such that $a_1 \leq b_1 \leq a_2 \leq b_2$ or $b_1 \leq a_1 \leq b_2 \leq a_2$. We say that S_i and S_j *overlap*.

Lemma 2.14: Let $\mathcal{C}$ be a connected component of the overlap graph H and let S be any segment with respect to C_i whose minimal chain C is nested. Then $S \in \mathcal{C}$ if and only if

(i) $R \in \mathcal{C}$ and there is a branch vertex on $t(C) \to_T s(C)$ or

(ii) there are attachments a_1 and a_2 of S and attachments b_1 and b_2 of segments in $\mathcal{C}$ with $a_1 \leq b_1 \leq a_2 \leq b_2$ or $b_1 \leq a_1 \leq b_2 \leq a_2$.

Proof. We first show $S \in \mathcal{C}$ if (i) or (ii) holds. For (i) the claim follows directly from the definition of the overlap graph. For (ii), assume $S \notin \mathcal{C}$ for the sake of a contradiction. Then either $R \notin \mathcal{C}$ or there is no branch vertex in $t(C) \to_T s(C)$ by (i). Further, no segment in $\mathcal{C}$ overlaps with S and hence any segment in $\mathcal{C}$ has its attachments points either strictly between a_1 and a_2 or outside the path $a_2 \to_T a_1$. Moreover, both classes of segments are non-empty. However, segments in the two classes do not overlap and R cannot be connected to the segments in the former class. Thus $\mathcal{C}$ is not connected, a contradiction.

If neither (i) nor (ii) hold, there can be no segment in $\mathcal{C}$ overlapping S and either S is not connected to R or no segment in $\mathcal{C}$ is connected to R. □

Lemma 2.15: Assume the algorithm reaches phase i. If the overlap graph H induced by the segments with respect to C_i is connected, we can add all segments of C_i. If H is not connected, we can exhibit a 2-edge-cut for any component of H that does not contain R.

Proof. Assume first that H is connected. Let $R, S_1, \ldots, S_k$ be the vertices of H in a preorder, e.g. the order they are explored by a DFS, starting at R, the vertex corresponding to the branch vertices on C_i. An easy inductive argument shows that we can add all segments in this order. Namely, let $k \geq 1$ and let C be the minimal chain of S_k. All attachment points of S_k lie on the path $t(C) \to_T s(C)$ by Lemma 2.13, and there is either an edge between R and S_k or an edge between S_j and S_k for some $j < k$. In the former case, there is a branch vertex on $t(C) \to_T s(C)$ at the beginning of the phase, in the latter case there is one after adding S_j. Thus the minimal chain of S_k can be added and then all other chains by Lemma 2.12.

On the other hand, suppose H is not connected. Let $\mathcal{C}$ be any connected component of H that does not contain R, and let $\mathcal{C}_R$ be the connected component that contains R. Let x and y be the minimal and maximal attachment points of the segments in $\mathcal{C}$, and let G_c be the current graph after adding all chains in $\mathcal{C}_R$. We first show that there is no branch vertex of G_c on the path $y \to_T x$. Assume otherwise and let w be any such branch vertex. Observe first that there must be a chain $C \in \mathcal{C}$ with $s(C) \leq w \leq t(C)$. Otherwise, every chain in $\mathcal{C}$ has all its attachment points at proper ancestors of w or at proper descendants of w and hence $\mathcal{C}$ is not connected. Let S be the segment containing C. By Lemma 2.13, we may assume that C is the minimal chain of S. Since $S \notin \mathcal{C}_R$, $R\,S$ is not an edge of H and hence no branch vertex exists on the path $t(C) \to_T s(C)$ at the beginning of part II of the phase. Hence w is an attachment point of a segment in $\mathcal{C}_R$. In particular $\mathcal{C}_R$ contains at least one segment. We claim that $\mathcal{C}_R$ must also have an attachment point outside $t(C) \to_T s(C)$. This holds since all initial branch vertices are outside the path and since $\mathcal{C}_R$ is connected. Thus $S \in \mathcal{C}_R$ by Lemma 2.14, a contradiction.

We show next that the tree-edge $x\,p(x)$ and the edge $z\,y$ from y's predecessor z on C_i to y form a 2-edge-cut; $z\,y$ may be a tree-edge or a back-edge. The following argument is similar to the argument in Theorem 2.6, but more refined.

Assume otherwise. Then, as in the proof of Theorem 2.6, there is a path $P = a \to b$ such that $a \leq u$ for all $u \in P$, and either a lies on $y \to_T x$ and b does not, or vice versa, and no inner vertex of P is in G_c. Moreover, the first edge $a\,v$ of P is a back-edge and v is a descendant of b. Note that unlike in the proof of Theorem 2.6, a and b need not lie on different links, as we want to show that $x\,p(x)$ and $z\,y$ form a cut and these might be different from the last edges on the link containing x and y.

Let D be the chain that starts with the edge $a\,v$. D does not belong to G_c, as no edge of P belongs to G_c. In particular, a does not s-belong to C_j for $j < i$ (as otherwise, D would already be added). Since $a \leq b$ and one of a and b lies on $y \to_T x$ (which is a subpath of C_i), a s-belongs to C_i. By the argument from the proof of Theorem 2.6, $t(D)$ is a descendant of b.

Let D^* be the chain that contains the last edge of P. If $t(D) = b$, $D = D^*$. Otherwise, $t(D)$ is a proper descendant of b. Let $u\,b$ be the last edge on the path $t(D) \to_T b$. We claim that $u\,b$ is also the last edge of P. This holds since the last edge of P must come from a descendant of b (as ancestors of b belong to G_c) and since it cannot come from a child different from y as otherwise P would have to contains a cross-edge. Thus $D^* \leq D$ by Lemma 2.4.(5) and hence $s(D^*) \leq s(D) \leq a$ by part (4) of the same lemma.

D and D^* belong to the same segment with respect to C_i, say S, and a and b are vertices in $S \cap G_c$. This can be seen easily. Since a s-belongs to C_i, D belongs to some

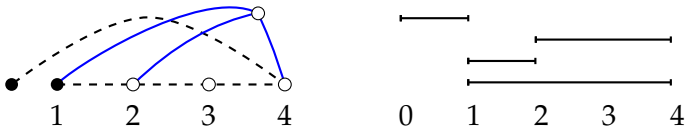


Figure 2.3: Intervals for the solid segment with respect to the dashed chain. It has the attachment points 1,2,4. Filled vertices are branching points.

segment with respect to C_i and since $D^* \leq D$, D^* belongs to the same segment. Since $t(D^*) = b$ and b is a vertex of G_c, D^* is the minimal chain in S. Thus D^* is nested and hence b s-belongs to C_i. Hence a and b are attachment points of S.

Thus S overlaps with $\mathcal{C}$ and hence $S \in \mathcal{C}$ by Lemma 2.14. Therefore x and y are not the extremal attachment points, that is the minimal (or maximal) vertices in $S \cap G_c$, of $\mathcal{C}$, a contradiction. □ □

It remains to show that we can find an order as required in Lemma 2.15, or a 2-edge-cut, in linear time. We reduce the problem of finding an order on the segments to a problem on intervals. W.l.o.g. assume that the vertices of C_i are numbered consecutively from 1 to $|C_i|$. Consider any segment S, and let $a_0 \leq a_1 \leq \ldots \leq a_k$ be the set of attachment points of S, i.e., the set of vertices that S has in common with C_i. By Lemma 2.13, a_0 and a_k are the endpoints of the minimal chain in S and each a_i, $0 < i < k$, is equal to $s(D)$ for some other chain in S. We associate the intervals

$$\{[a_0, a_\ell] | 1 \leq \ell \leq k\} \cup \{[a_\ell, a_k] | 1 \leq \ell < k\},$$

with S and for every branch vertex v on C_i we define an interval $[0, v]$. See Figure 2.3 for an example.

We say two intervals $[a, a']$, $[b, b']$ *overlap* if $a \leq b \leq a' \leq b'$. Note that overlapping is different from intersecting; an interval does not overlap intervals in which it is properly contained or which it properly contains. This relation naturally induces a graph H' on the intervals. Contracting all intervals that belong to the same segment into one vertex makes H' isomorphic to the overlap graph as required for Lemma 2.15. Hence we can use H' to find the order on the segments.

A naive approach that constructs H', contracts intervals, and runs a DFS will fail, since the overlap graph can have a quadratic number of edges. However, using a method developed by Olariu and Zomaya [59], we can compute a spanning forest of H' in time linear in the number of intervals. The presentation in [59] is for the PRAM and thus needlessly complicated for our purposes. A simpler explanation can be found in the appendix.

The number of intervals created for a chain C_i is bounded by

$$|\text{NestedChildren}(C_i)| + 2|\text{Interlacing}(C_i)| + |V_{\text{branch}}(C_i)|,$$

where NestedChildren(C_i) are the nested children of C_i, Interlacing(C_i) are the interlacing chains that start on C_i, and $V_{\text{branch}}(C_i)$ is the set of branch vertices on C_i. Note that we generate the interval $[s(C), t(C)]$ for each nested child C, and the intervals $[s(C), s(D)]$ and $[s(D), t(C)]$ for each interlacing chain D belonging to a segment with nested minimal chain C. Thus the total time spend the ordering procedure is $O(m)$. From the above discussion follows:

Theorem 2.16: For a 3-edge-connected graph, a Mader construction sequence can be found in time $O(n + m)$.

2.9 Verifying the Mader Sequence

Our algorithm computes a certificate alongside its answer. The certificate is either a 2-edge-cut, or a sequence of Mader-paths. In this section we discuss how to verify that the certificate is correct. We will see that this is a very simple procedure compared to connectivity testing.

If the graph is not 3-connected, that is, the certificate is a 2-edge-cut, we simply remove the two edges and verify that G is no longer connected.

For checking the Mader sequence, we doubly-link each edge in a Mader-path to the corresponding edge in G. Let G' be a copy of G. We remove the Mader-paths, in reverse order of the sequence, suppressing vertices of degree two as they occur. This can create multiple edges and loops. Let G'_i be the multi-graph before we remove the i-th path P_i. We need to verify the following:

- G must have minimum degree three.
- The union of Mader-paths must be isomorphic to G and the Mader-paths must partition the edges of G. This is easy to check using the links between the edges of the paths and the edges of G.
- The paths we remove must be ears. More precisely, at step i, P_i must have been reduced to a single edge in G'_i, as inner vertices of P_i must have been suppressed if P_i is an ear for G'_i.
- The P_i must not subdivide the same link twice. That is, after deleting the edge corresponding to P_i, it must not be the case that both endpoints are still adjacent (or equal, i.e. P_i is a loop) but have degree two.
- When only two paths are left, the graph must be a K_2^3.

2.10 The Cactus Representation of 2-Cuts

We review the cactus representation of 2-cuts in a 2-connected but not 3-connected graph and show how to certify it.

A *cactus* is a graph in which every edge is contained in exactly one cycle. Dinits, Karzanov, and Lomonsov [26] showed that the set of mincuts of any graph has a cactus representation, i.e., for any graph G there is a cactus C and a mapping $\phi : V(G) \to V(C)$ such that the mincuts of G are exactly the preimages of the mincuts of C, i.e., for every mincut[2] $A \subseteq V(C)$, $\phi^{-1}(A)$ is a mincut of G, and all mincuts of G can be obtained in this way. The pair (C, ϕ) is called a *cactus representation* of G. Fleiner and Frank [29] provide a simplified proof for the existence of a cactus

[2]For this theorem, a cut is specified by a set of vertices, and the edges in the cut are the edges with exactly one endpoint in the vertex set.

representation. We will call the elements of $V(G)$ *vertices*, the elements of $V(C)$ *nodes*, and the preimages of nodes of C *blobs*.

In general, a cactus representation needs to include nodes with empty preimages. This happens for example for the K_4; its cactus is a star with double edges where the central node has an empty preimage and the remaining nodes correspond to the vertices of the K_4. For graphs whose mincuts have size two, nodes with empty preimages are not needed, and a cactus representation can be obtained by contracting the 3-edge-connected components into a single node.

Lemma 2.17 ([56, Section 2.3.5])**:** Let G be a 2-edge-connected graph that is not 3-edge-connected. Contracting each 3-edge-connected components of G into a node yields a cactus representation (C, ϕ) of G with the following properties:

i) The edges of C are in one-to-one correspondence to the edges of G that are contained in a 2-cut.

ii) For every node $c \in V(c)$, $\phi^{-1}(c)$ is a 3-edge-connected component of G.

Proof. Consider the graph C' whose nodes correspond to the 3-edge-connected components of G and there is an edge between two nodes u, v in C' if there is an edge between the subgraphs in G induced by u and v. Alternatively, C' is obtained from G by contracting each 3-edge-connected component of G into a node. Clearly, the edges of C' correspond to the edges of G that are contained in 2-edge-cuts.

We show that C' is a cactus. Assume first that there is an edge $u\,v$ in C' contained in two distinct cycles X and Y. Let P be a maximal path in $X - Y$; note that P is non-empty. Then the endpoints a and b of P are connected by at least three edge-disjoint paths in G, namely P and the two a-b-paths in Y. This contradicts that a and b are in different 3-edge-connected components. Assume next that there is an edge $u\,v$ in C' that is not contained in any cycle. Then this edge is a 1-edge-cut in G, a contradiction. □

2.10.1 Verifying a Cactus Representation

Let G be a graph and let (C, ϕ) be an alleged cactus-representation of its 2-cuts in the sense of Lemma 2.17. We show how to certify an cactus representation in linear time. To verify the cactus representation, we need to check two things. First, we need to ensure that C is indeed a cactus graph, that is, every edge of C is contained in exactly one cycle, that ϕ is a surjective mapping and hence there are no empty blobs, and that every edge of G either connects two vertices in the same blob or is also present in C. Second, we need to verify that the blobs of C are 3-edge-connected components of G. For this purpose, the cactus representation is augmented by a Mader construction sequence for each blob B. The verification procedure from Sect. 2.9 can then be applied.

We first verify that C is a cactus. We compute a chain decomposition of C and verify that every chain is a cycle. We label all edges in the i-th cycle by i. We have now verified that C is a cactus.

Surjectivity of ϕ is easy to check. We then iterate over the edges $u\,v$ of G. If its endpoints belong to the same blob, we associate the edge with the blob. If its endpoints do not belong to the same blob, we add the pair $\phi(u)\,\phi(v)$ to a list.

Having processed all edges, we check whether the constructed list and the edge list of C are identical by first sorting both lists using radix sort and then comparing them for identity.

We finally have to check that the blobs of C correspond to 3-edge-connected components of G. Our goal is to use the certifying algorithm for 3-edge-connectivity on the substructures of G that represent 3-edge-connected components. Let B be any blob. We already collected the edges having both endpoints in B. We also have to account for the paths using edges outside B. We do so by creating an edge $u\,v$ for a every path in G leaving B at vertex u and returning to B at vertex v. It is straightforward to compute these edges; we look at all edges having exactly one endpoint in the blob. Each such edge corresponds to an edge in C. For each such edge, we know to which cycle it belongs. The outgoing edges pair up so that the two edges of each pair belong to the same cycle.

The maximality of each blob B is given by the fact that every edge of C is contained in a 2-edge-cut of C and hence contained in a 2-edge-cut of G.

Every algorithm for computing the 3-edge-connected components of a graph, e.g. [55, 70, 72, 73, 56], can be turned into a certifying algorithm for computing the cactus representation of 2-cuts. We obtain the cactus C and the mapping ϕ by contraction of the 3-edge-connected components (Lemma 2.17). Then one applies our certifying algorithm for 3-edge-connectivity to each 3-edge-connected component. The drawback of this approach is that it requires *two* algorithms that check 3-connectivity. In the next section we will show how to extend our algorithm so that it computes the 3-edge-connected components and the cactus representation of 2-cuts of a graph directly.

2.11 Computing a Cactus Representation

We discuss how to extend the algorithm to construct a cactus representation. We begin by examining the structure of the 2-cuts of G more closely to extend our algorithm such that it finds all 2-cuts of the graph and encodes them efficiently.

We will first show that the two edges of every 2-edge-cut of G are contained in a common chain. This restriction allows us to focus on the 2-edge-cuts that are contained in the currently processed chain C_i only. In the subsequent section, we show how to maintain a cactus for every phase i of the algorithm that represents all 2-edge-cuts of the graph of the branch vertices and links of $C_1 \cup \ldots \cup C_i$ in linear space. The final cactus will therefore represent all 2-edge-cuts in G.

There is one technical detail regarding the computation of overlap graphs: For the computation of a Mader-sequence in Section 2.8, we stopped the algorithm when the first 2-edge-cut occurred, as then a Mader-sequence does not exist anymore. Here, we simply continue the algorithm with processing the next chain C_{i+1}. This does not harm the search for cuts in subsequent chains, as the fact that 2-edge-cuts are only contained in common chains guarantees that every 2-edge-cut that contains an edge e in C_i has its second edge also in C_i.

For simplicity, we assume that G is 2-edge-connected and has minimum degree three from now on. Then all 3-edge-connected components contain at least two vertices.

2.11.1 Properties and Representation of 2-cuts on Chains

In phase i of the algorithm, using Lemma 2.15, we can find a 2-edge-cut for each connected component of the overlap graph H that does not contain R (R is the special vertex in H that represents the branch vertices on C_i). Lemma 2.19 shows that the set of edges contained in these cuts is equal to the set of edges contained in any cut on C_i. Lemma 2.18 states easy facts about 2-edge-cuts, in particular, that the edges of any 2-edge-cut are contained in a common chain. The proofs can be found in many 3-connectivity papers, e.g. [55, 70, 72, 73]. As in the previous sections, all DFS-tree-edges are oriented towards the root, while back-edges are oriented away from the root.

Lemma 2.18: Let T be a DFS-tree of a 2-edge-connected graph G. Every 2-edge-cut $(u\,v, x\,y)$ of G satisfies the following:

(1) At least one of $u\,v$ and $x\,y$ is a tree-edge, say $x\,y$.

(2) $G - u\,v - x\,y$ has exactly two components. Moreover, the edges $u\,v$ and $x\,y$ have exactly one endpoint in each component.

(3) The vertices u, v, x, and y are contained in the same leaf-to-root path of T.

(4) If $u\,v$ and $x\,y$ are tree-edges and w.l.o.g. $u \leq y$, the vertices in $y \rightarrow_T u$ and $\{x, v\}$ are in different components of $G - u\,v - x\,y$.

(5) If $u\,v$ is a back-edge, then $x\,y \in (v \rightarrow_T u)$ and, additionally, the vertices in $v \rightarrow_T x$ and $y \rightarrow_T u$ are in different components of $G - u\,v - x\,y$.

Let $\mathcal{C}$ be a chain decomposition of G. For every 2-edge-cut $\{u\,v, x\,y\}$ of G, $u\,v$ and $x\,y$ are contained in a common chain $C \in \mathcal{C}$.

In phase i of the algorithm, using Lemma 2.15, we can find a 2-edge-cut for any connected component of the overlap graph H that does not contain R (R is the special vertex in H that represents the branch vertices on C_i). The next lemma shows that the set of edges contained in these cuts is equal to the set of edges contained in any cut on C_i.

Lemma 2.19: Let $\mathcal{E}$ be the set of edges that are contained in the 2-edge cuts induced by the connected components of the overlap graph H at the beginning of part II of phase i. Then any 2-edge-cut $\{x\,y, u\,v\}$ on C_i is a subset of $\mathcal{E}$.

Proof. Assume for the sake of contradiction that there is an edge $u\,v$ in the 2-edge-cut that is not in $\mathcal{E}$. We distinguish the following cases.

First assume that both $u\,v$ and $x\,y$ are tree-edges and w.l.o.g. $v < u \leq y < x$. Since G has minimal degree three, every node on C_i has an incident edge that is not on C_i. Hence it is either a branch vertex, or belongs to some segment with respect to C_i (incident back-edges start chains in segments w.r.t. C_i, incident tree edges are the last edges of chains in segments w.r.t. C_i). As $s(C_i) \leq v$ is a branch vertex, by Lemma 2.18.(4) the path $y \rightarrow_T u$ can not contain a branch vertex. In particular, u is not a branch vertex.

Let S_u be any segment having u as attachment vertex. All segments in the connected component of S_u in H must have their attachment vertices on $y \rightarrow_T u$ and the

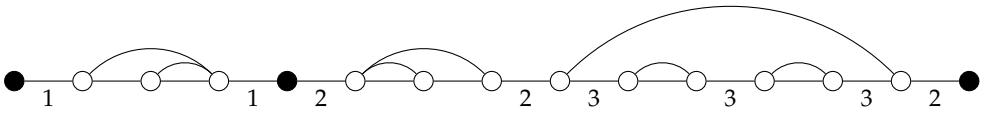


Figure 2.4: The intervals induced by the connected components of the overlap graph H form a laminar family. The levels of this family encode which edges form pairwise 2-cuts. Two edges in the figure are labeled with the same number if they form a cut. Filled vertices are branch vertices.

connected component does not contain R. Hence this connected component induces a cut containing $u\,v$.

Now assume that one of $u\,v$ and $x\,y$ is a back-edge. If $u\,v$ is the back-edge, then $u = s(C_i)$ and we have $u < y < x < v$ by Lemma 2.18. The path $v \to_T x$ cannot contain a branch vertex. Let S_v be any segment that has v as attachment vertex. All segments in the connected component of S_v must have their attachment vertices on $v \to_T x$ and the connected component does not contain R. Hence $u\,v$ is contained in a cut induced by this connected component.

If on the other hand $u\,v$ is the tree-edge we have $y < v < u < x$ basically the same argument applies when we replace S_v by a segment S_u containing u. □

In this section we show how to compute a space efficient representation of all 2-cuts on the chain C_i. Using this technique we can store all 2-cuts in G in linear space. In the next section we will then use this to construct the cactus-representation of all 2-cuts in G.

Number the edges in C_i as $e_1, e_2, \ldots, e_k$. Here e_1 is a back-edge and e_2 to e_k are tree edges. We start with a simple observation. Let $h < i < j$. If (e_h, e_i) and (e_i, e_j) are 2-edge-cuts, then (e_i, e_j) is a 2-edge-cut.

Using this observation, we want to group the edges of 2-edge-cuts of C_i such that (i) every two edges in a group form a 2-edge-cut and (ii) no two edges of different groups form a 2-edge-cut. The existence of such a grouping has already been observed in [55, 70, 73]. We show how to find it using the data structures we have on hand during our algorithm.

Consider the overlap graph H in phase i of our algorithm. We need some notation. Let I be the set of intervals on C_i that contains for every component of H (except the component representing the branch vertices on C_i) with extremal attachment vertices a and b the interval $[a, b]$. Since the connected components of H are maximal sets of overlapping intervals, I is a *laminar family*, i.e. every two intervals in I are either disjoint or properly contained in each other. In particular, no two intervals in I share an endpoint. The layers of this laminar family encode which edges form pairwise 2-cuts in G, see Figure 2.4. We define an equivalence relation to capture this intuition.

For an interval $[a, b]$, $a < b$, let $\ell([a, b])$ and $r([a, b])$ be the edges of C_i directly before and after a and b, respectively. We call $\{\ell([a, b]), r([a, b])\}$ the *interval-cut* of $[a, b]$. For a subset $S \subseteq I$ of intervals, let E_S be the union of edges that are contained in interval-cuts of intervals in S. According to Lemma 2.19, every 2-edge-cut in C_i consists of edges in E_I.

We now group the edges of E_I using the observation above. Let two intervals $I_1 \in I$ and $I_2 \in I$ *contact* if $r(I_1) = \ell(I_2)$ or $\ell(I_1) = r(I_2)$. Clearly, the transitive closure $\equiv$ of the contact relation is an equivalence relation. Every block B of $\equiv$ is a set of pairwise disjoint intervals which are contacting consecutively. This allows us to compute the blocks of $\equiv$ efficiently. We can compute them in time $|I|$ and store them in space $|I|$ by using a greedy algorithm that iteratively extracts the inclusion-wise maximal intervals in I that are contacting consecutively.

Lemma 2.20 ([55, 70, 73])**:** Two edges e and e' in C_i form a 2-edge-cut if and only if e and e' are both contained in E_B for some block B of $\equiv$.

2.11.2 An Incremental Cactus Construction

In this section we show how to construct a cactus representation incrementally along our algorithm for constructing a Mader sequence. At the beginning of each phase i, we will have a cactus for the graph G^i whose vertices are the branch vertices that exist at this time and whose edges are the links between these branch vertices.

We assume that G is 2-edge-connected but not 3-edge connected and that G has minimum degree three. This ensures that in phase i every vertex on the current chain C_i belongs to some segment or is a branch vertex.

We will maintain a cactus representation (C, ϕ), i.e., for every node v of C, the blob $B = \phi^{-1}(v)$ is the vertex-set of a 3-edge-connected component in G^i. We begin with a single blob that consists of the two branch vertices of the initial K_2^3, which clearly are connected by three edge-disjoint paths.

Consider phase i, in which we add all chains whose source s-belongs to C_i. At the beginning of the phase, the endpoints of C_i and some branch vertices on C_i already exist in G^i. We have a cactus representation of the current graph. The endpoints of C_i are branch vertices and belong to the same blob B, since 2-edge-cuts are contained in chains.

We add all segments that do not induce cut edges and tentatively assign all vertices of C_i to B. If the algorithm determines that C_i does not contain any 2-edge-cut, the assignment becomes permanent, the phase is over and we proceed to phase $i + 1$. Otherwise we calculate the efficient representation of 2-edge-cuts on C_i from Sect. 2.11.1.

Let e_1 be the first edge on C_i in a 2-edge-cut, let A be a block of the contact equivalence relation described in the last section containing e_1 and let $E_A = \{e_1, e_2, \ldots, e_\ell\}$ such that e_j comes before e_{j+1} in C_i for all j. Then every two edges in E_A form a 2-edge-cut. We add a cycle with $\ell - 1$ empty blobs $B_2, \ldots, B_\ell$ to B in C. The ℓ new edges correspond to the ℓ edges in E_A.

For every pair $e_j = (a, b)$, $e_{j+1} = (c, d)$ in E_A we remove the vertices between these edges from B. Since the edges in C_i are linearly ordered, removing the vertices in a subpath takes constant time. We place the end vertices b and c of the path between e_j and e_{j+1} in the blob B_j, add the segments that induced this cut and recurse on the path between b and c. That is, we add all vertices on the path from b to c to B_j, check for cut edges on this path, and, should some exist, add more blobs to the cactus. The construction takes constant time per blob. Figure 2.5 shows an example.

Graphs that contain nodes of degree two can be handled in the same way, if we add a cycle to each degree two node u. This cycle creates a segment w.r.t. the chain

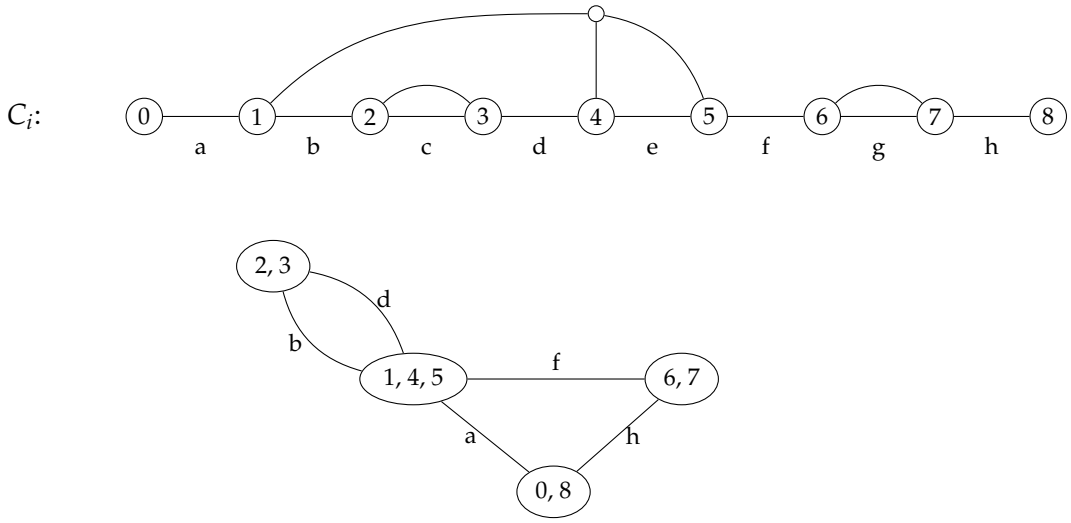


Figure 2.5: The segments attached to chain C_i and the corresponding part of the cactus. We first tentatively assign vertices 1–7 to the blob containing the endpoints $\{0, 8\}$ of C_i. The top level cuts are the pairs in the block $\{a, f, h\}$. So we create a cycle with three edges and attach it to the blob containing 0 and 8. We move vertices 1–5 to the blob between a and f, vertices 6–7 to the blob between f and h, and keep vertices 0 and 8 in the parent blob. We then recurse into the first blob. The second level cuts are the pairs in the block $\{b, d\}$. So we create a cycle with two edges and move vertices 2 and 3 to the new blob.

to which u s-belongs and hence the algorithm correctly identifies the two incident edges as cut edges.

Lemma 2.21: The above incremental procedure constructs a cactus representation of the 2-edge-cuts in G in linear time.

Proof. Each vertex in G s-belongs to some chain. In the phase in which that chain is treated, all its vertices are added to a blob. Whenever we move a vertex to different blob, we remove it from its previous blob. Therefore each vertex of G is contained in exactly one blob.

Whenever we add edges to the cactus, we do so by adding a cycle that shares exactly one node with the existing cactus. Hence every edge in the cactus lies on exactly one cycle.

Let $\{e_1, e_2\}$ be a 2-edge-cut in G. The two edges must lie on a common cycle in the cactus, since the edges in the cactus are in one-to-one correspondence with edges of G and cutting a cycle in only one place cannot disconnect a graph. As the cycles of the cactus touch in at most one vertex, e_1 and e_2 are a cut in the cactus as well.

Conversely let e'_1, e'_2 be a cut in the cactus and let e_1, e_2 be the corresponding edges in G. Then e'_1, e'_2 must lie on some common cycle which, upon their removal, is split into two non-empty parts H_1, and H_2. Assume that $G - e_1 - e_2$ is still connected,

then there must be a path from a vertex in the preimage of H_1 to a vertex in the preimage of H_2 in $G - e_1 - e_2$. This path must contain at least one edge $u\,v$ that does not participate in any 2-edge-cut, as otherwise it would be a path in the cactus as well. Moreover, u and v must lie in different blobs B_u and B_v of the cactus.

The one that was created last, say B_u, must be different from the initial blob. Consider the time when B_u was created in the incremental construction of the cactus. We introduced a cycle to some preexisting blob B^* on which all edges where cut edges, in particular the two cut edges incident to B_u. However, the edge $u\,v$ still connects B_u to the rest of graph, since B_v also exists at this time, a contradiction. □

2.12 Computing all 3-Vertex-Connected Components

By applying the idea of the last section, the certifying algorithm for 3-vertex-connectivity [65] (which is also based on chain decompositions) can be used to compute the 3-vertex-connected components of a graph. This has been conjectured in [64, p. 18] and yields a linear-time certifying algorithm to construct a SPQR-tree of a graph; we refer to [40, 34] for details about 3-vertex-connected components and SPQR-trees.

A pair of vertices $\{x, y\}$ is a separation pair of G if $G - x - y$ is disconnected. Similarly to the edge-connectivity case, it suffices to compute all vertices that are contained in separation pairs of G in order to compute all 3-vertex-connected components of G. We assume that G is 2-vertex-connected and has minimum degree 3.

For a rooted tree T of G and a vertex $x \in G$, let $T(x)$ be the subtree of T rooted at x. The following lemmas show that separation pairs can only occur in chains. Weaker variants of Lemma 2.22 can be found in [40, 75, 76].

Lemma 2.22: Let T be a DFS-tree of a 2-connected graph G and $\mathcal{C}$ be a chain decomposition of G. For every separation pair $\{x, y\}$ of G, x and y are contained in a common chain $C \in \mathcal{C}$.

Proof. The following simple observation will be useful. Let r be the root of T and let $x \neq r$ be any vertex. Then for every $t \in T(x) - x$, there is a path P from t to a vertex $s \in G - T(x)$ such that P consists only of vertices in $T(x) - x \cup s$.

We first prove that x and y are *comparable* in T, i.e., contained in a leaf-to-root path of T. Assume they are not. Then $G - x - y$ consists of at most three connected components: one containing the least common ancestor of x and y in T and the at most two connected components that contain the proper descendants of x and y, respectively. According to the observation above, these components coincide, contradicting that $\{x, y\}$ is a separation pair.

Let x' be the child of x in T that lies on the path $y \to_T x$. Clearly, if $x' = y$, the chain containing the edge xy is a common chain containing x and y. Otherwise, $x' \neq y$. If $x = r$, then there is a back-edge rt such that $t \in T(y)$, according to the fact that $G - r$ is connected by $T - r$ and due to the observation above. This back-edge rt implies that the first chain C that traverses a vertex of $T(y)$ starts at r and, hence, contains x and y.

In the remaining case, $x' \neq y$ and $x \neq r$. Let st be a back-edge that connects an ancestor s of x with a descendant t of x' (possibly x' itself) such that s is minimal; this edge st exists, since G is 2-vertex-connected. According to [63], C_1 is the only cycle in $\mathcal{C}$ and it follows that $s < x$. If $t \in T(y)$, the first chain C in $\mathcal{C}$ that contains such a back-edge contains x and y and, hence, satisfies the claim. Otherwise, t is a vertex in $T(x') - T(y)$. Due to the back-edge st, $G - x - T(y)$ is contained in one connected component of $G - x - y$. According to the observation above (applied on y), $\{x, y\}$ can form a separation pair only if y has a child y' such that all back-edges that end in $T(y')$ start either in $T(y)$ or at x. Since G is 2-connected, there must be a back-edge from x to $T(y')$. The first chain C in $\mathcal{C}$ containing such a back-edge gives the claim, as it contains x and y. □

Similarly to edge-connectivity, the connected components of the overlap graph for C_i represent all vertices in separation pairs that are contained in C_i. The connected components of the overlap graph can be computed efficiently [65, Lemma 51]. After finding all these vertices for C_i, a simple modification allows the algorithm in [65, p. 508] to continue, ignoring all previously found separation pairs: For every separation pair $\{x, y\}$, $x < y$, that has been found when processing C_i, there is a vertex v strictly between x and y in C_i. Furthermore, by doing a preprocessing [65, Property B, p. 508] one can assume that also $t(C_i) \rightarrow_T s(C_i)$ has an inner vertex w. We eliminate every separation pair $\{x, y\}$ after processing C_i by simply adding the new back-edge vw to G. As the new chain containing vw is just an edge, this does not harm future processing steps.

According to Lemma 2.22, this gives all vertices in the graph that are contained in separation pairs. The 3-vertex-connected components can then be computed in linear time by iteratively splitting separation pairs and gluing together certain remaining structures, as shown in [40, 34].

2.13 A Simplified Certifying Algorithm for 3-Vertex Connectivity

In section 2.6 we gave a simple algorithm for checking 3-edge-connectivity. A very similar algorithm can be used to check 3-vertex-connectivity. In [65] a linear time certifying algorithm is presented. In this section we describe a much simpler greedy algorithm that runs in time $O((n + m) \log(n + m))$.

The certificate for 3-vertex-connectivity is almost identical to the one for 3-edge-connectivity. It is a construction sequence that uses three operations and starts from a K_2^3.

Theorem 2.23 ([9])**:** Every 3-vertex-connected graph (and no other graphs) can be constructed from a K_2^3 using the following three operations

1. Adding an edge (possibly parallel).
2. Subdividing an edge $x\,y$ and connecting the new vertex to any existing vertex, *except x or y.*
3. Subdividing two distinct edges $w\,x$, $y\,z$ and connecting the two new vertices. *Except for the first step that turns the K_2^3 into a K_4, the edges must not be parallel.*

(Emphasis on the differences to Theorem 2.1)

Just like Theorem 2.1, this theorem readily generalizes to adding paths to a subdivision.

For edge-connectivity, an ear is always a Mader-path, unless the endpoints lie on the same link. Vertex-connectivity is slightly more restrictive. The endpoints of the ear must not lie on the same link or on parallel links. Furthermore, if exactly one endpoint of the ear is a branch vertex, it must not lie at the end of the link that contains the non-branch vertex. We must therefore be more careful when adding chains greedily in the algorithm.

We will in fact impose additional restrictions on the construction sequence we find:

1. We only add a path $a \to b$ if the tree path $a \to_T b$ contains a branch vertex as an inner node.
2. We never construct two parallel links L_1, L_2 such that one of them consists only of tree edges.

The following lemmas shows under which circumstances a path is a Mader-path in the case of vertex connectivity that does not violate the above restrictions.

Lemma 2.24: [66, Lemma 72] A path $P = a \to b$ is a Mader-path w.r.t. a parent-first subdivision G_c that respects restriction 2 if

1. P is an ear w.r.t. G_c: $P \cap G_c = \{a, b\}$
2. The vertices a and b are either both branch vertices, one is a branch vertex and the other does not lie on an incident link, or both are non-branch vertices and lie on different links.

Proof. We begin by showing that a path that fulfills these requirements is a Mader-path. Assume for the sake of contradiction that a path $P = a \to_T b$ like in the lemma is not a Mader-path. It is easy to see that the only thing that can prevent P from being a Mader-path is that a and b lie on parallel links Q and Z, so assume this to be the case.

Both Q and Z must contain at least one back-edge because of restriction 2. Let C_q and C_z be the chains that contain Q and Z. As these chains contain only one back-edge, this implies that $s = s(C_q) = s(C_z)$ and $s \in Q \cap Z$. Let $t = Q \cap Z - s$ be the other common vertex of the two links. By construction the vertices of Q and Z lie in different subtrees of the DFS tree. Since P links these two subtrees and the tree does not contain a cross-edge, P must contain an ancestor of t. As G_c is parent-first, this ancestor is part of G_c, contradicting $P \cap G_c = \{a, b\}$. □

As we can see from Lemma 2.24, restriction 2 makes it easier to add new paths, as we do not have to check whether the endpoints lie on parallel links. Note that G_c may violate restriction 2 after we add a Mader-path with this lemma. The next lemma shows when we can add a Mader-path without violating the restriction.

Lemma 2.25: A Mader-path $P = a \to b$ can be added to a parent-first subdivision G_c that respects restriction 2 if P contains a back-edge and the tree path $a \to_T b$ contains a branch vertex as inner vertex.

Proof. We show that G_c still respects restriction 2 after adding P. The path P becomes a new link in G_c. Moreover, a and b become branching vertices and split the links on which they lie. As $a \to_T b$ contains a branch vertex, P is not parallel to a link that contains only tree edges. Similarly, splitting the links on which a and b lie can not create a new link that consists only of tree edges and is parallel to another link in G_c, unless this was already the case before the splitting. □

Lemma 2.25 implies that we can add a chain C if $s(C) \to_T t(C)$ contains a branch vertex as an inner vertex and $s(C)$ and $t(C)$ lie on different links.

The general idea for the algorithm stays the same as in the edge-connectivity case. We compute a chain decomposition and want to find a construction sequence by adding chains. Chains are addable if the conditions of Lemma 2.25 are satisfied. We keep track of the chains that are currently addable and add them greedily. Adding chains creates new branch vertices and makes the adding of other chains possible. So after every adding of a chain we check whether new chains can be added.

However, sometimes this procedure gets stuck, that is, no chain can be added, even though there is no cut in the graph. See for example Figure 2.6. In the figure, $C_1 \cup C_2$ form the initial K_2^3. The chain C_3 is the only ear with respect to this subgraph. However, we are not allowed to add this chain, since this would constitute subdividing the $a\,c$ edge in the K_2^3 and connecting the new vertex d to a, which is forbidden by operation 2 in Theorem 2.23. Note however that the path $b\,f\,e\,d$ is a Mader-path and after adding this path, the remaining edges $a\,f$ and $a\,e$ can be added. The structures that cause this situation are called *caterpillars* in [65].

2.14 Caterpillars

Caterpillars are collections of chains that can be decomposed into a set of Mader-paths under certain conditions. A chain D starts a caterpillar if its parent $\widehat{D}$ is not added and $s(\widehat{D}) = s(p(\widehat{D}))$. Let $D, \widehat{D}, p(\widehat{D}), \ldots, D^*$ be all ancestors of D such that none is added and all except D itself start at the same node as their parent. This collection of chains forms a caterpillar. To find caterpillars after adding a chain C, we examine the chains D with $s(D)$ on C and check whether they are interlacing and have a parent $\widehat{D}$ that has the same source vertex as its parent, i.e. $s(\widehat{D}) = p(\widehat{D})$. If so, we collect all the chains belonging to the caterpillar by traversing the chain tree upwards until we reach a chain that is already added to the graph, or the chain does no longer begin at the same node as its parent.

Next we discuss under which circumstances caterpillars can be added and how to detect the necessary conditions during the execution of the algorithm.

Let C_k be the least common ancestor of all chains in the caterpillar. In Figure 2.7 we see the two conditions under which a caterpillar can be decomposed into a set of Mader-paths. If $s(D)$ is an inner vertex of $t(C_k) \to_T s(C_k)$ (case 1), the caterpillar can be added as soon as C_k is added (left side of the figure). If on the other hand $s(D)$ lies on C_k (case 2), then the caterpillar can be added if the path $s(C_k) \to_{C_k} t(D^*)$ contains a branch node. We call a caterpillar that fulfills one of these conditions *good*. See Lemma 76 in [66] why these are the only cases were caterpillars can be decomposed to Mader-paths (this Lemma is however unnecessary for the correctness of our algorithm).

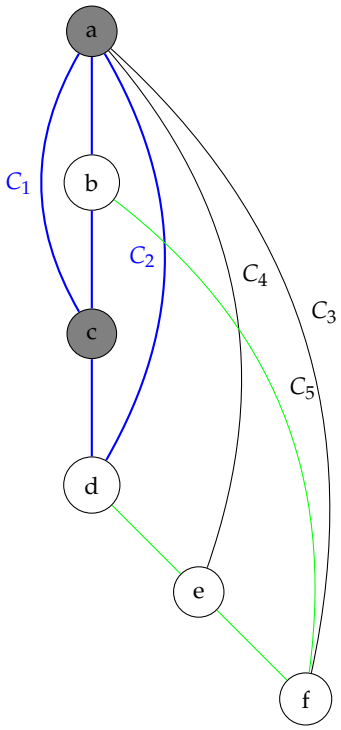


Figure 2.6: A graph where no chain can be added to the K_2^3 (blue edges). The green path $b\ f\ e\ d$ is a Mader-path. After adding it, the remaining paths can be added.

Which of the two cases apply can be decided as the caterpillar is first detected. A type 1 caterpillar can be added as soon as C_k is added. A type two caterpillar can be added as soon as $s(C_k) \rightarrow_{C_k} t(D^*)$ contains a branch vertex. Both cases are easy to detect during the algorithm.

Lemma 2.26: If a caterpillar is good, it can be added.

Proof. We distinguish between the two possible configurations for good caterpillars as shown in Figure 2.7. First consider the case where $s(D) \in t(C_k) \rightarrow_T s(C_k)$.

The path $P = s(D) \rightarrow_D t(D) \rightarrow_T t(D^*)$ (green in Figure 2.7) can be added with Lemma 2.25, as it is an ear and its endpoints lie on different links. It is easy to see that the paths that remain of the caterpillar after adding P can be added as well.

In the other case $s(D) \in C_k$ and there is a branch vertex a on the path $s(C_k) \rightarrow_{C_k} s(D)$. The green path $P = s(D) \rightarrow_D t(D) \rightarrow_{D^*} s(D^*)$ in Figure 2.7 can be added by Lemma 2.25. Next we add the path $t(D) \rightarrow_T t(D^*)$. As this path does not contain a back-edge, we need an additional argument why it can be added. Clearly, it is a Mader-path, as it is an ear and the branch vertex a prevent the link on which $t(D^*)$ lies to be parallel to the link created by P (even in the case where $s(D) = t(C_k)$). It also does not violate restriction 2, as there is no parallel link to it. The remaining paths of this caterpillar can then by added according to Lemma 2.25. □

2.14.1 The Greedy Vertex-Connectivity Algorithm

The algorithm is basically identical to the greedy chain addition algorithm for edge-connectivity from Section 2.6. Let us re-examine the algorithm to see where changes are necessary.

We begin by computing a chain decomposition and a $K_2^3 = C_1 \cup C_2$. During the chain decomposition we check that the graph is 2-vertex-connected.[3] As before we maintain the links of the current subdivision as doubly linked lists and label all vertices as *inactive*, *branch*, or *non-branch*. Also as before, we manage a list of addable structures and for each chain store the list of its children.

To detect caterpillars we also store for each vertex v which chains start at v. For every caterpillar starting at some chain D that we detect while updating this information, we add a reference to the caterpillar to the end of the chain C_k that contains $s(D)$. This way, we can easily detect when a type 2 caterpillar (Figure 2.7, right side) becomes addable.

For initialization, we iterate over the children of C_1 and C_2. We add the first chain, say C, whose endpoints are not inner vertices of the same link and thus create a K_4.[4] It might happen that there is no such chain, for example in the case of Figure 2.6. In that case we detect the presence of a good caterpillar and add it completely. For each chain that is a child of the chains in the subdivision at this point, we check whether it is addable using Lemma 2.25 and if so add it to the list of addable structures $\mathcal{L}$. We also initialize for every vertex v the list of chains starting at v and detect caterpillars.

[3] The graph is 2-vertex-connected if the minimum degree is at least 2 and C_1 is the only chain that forms a cycle. See Lemma 11 in [65].

[4] We do this because of the exception in operation 3 in Theorem 2.23.

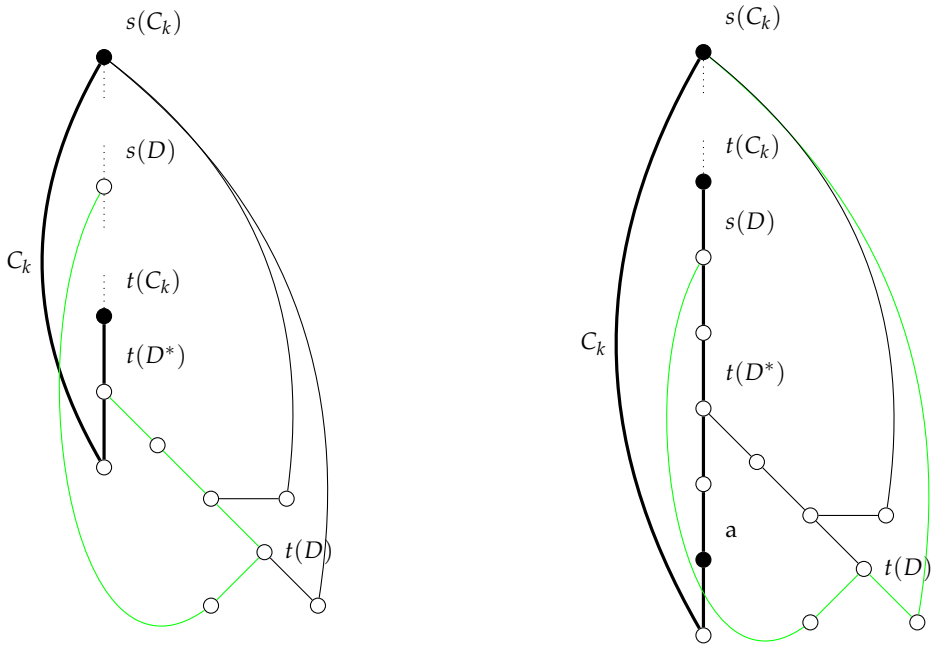


Figure 2.7: The two configurations when a caterpillar can be added. Left: $s(D)$ lies on the tree path $t(C_k) \to_T s(C_k)$. Right: $s(D)$ s-belongs to C_k and there is a branch vertex a on the path $s(C_k) \to_{C_k} s(D)$. In green are shown the first Mader-paths into which the caterpillars are decomposed.

As long as there are elements in $\mathcal{L}$, we take one and add it. This makes the attachment points[5] of the structure branch vertices (if they have been non-branch before) and splits the links on which they lie. As in the edge-connectivity algorithm, process the children of D and check the applicability of Lemma 2.25 to update $\mathcal{L}$ and compute the necessary extra information stored at the vertices of D. Furthermore, we traverse the links that are split by the adding of D and add the structures that have become addable to $\mathcal{L}$.

Lemma 2.27: If the Greedy-Chain-Addition Algorithm stops before all chains are added, the graph contains a cut.

Proof. This proof is very similar to the proof of Theorem 2.6.

Let G_c be the current subdivision. For the sake of contradiction, suppose the following

a) The graph is 3-connected.

b) No chain or caterpillar can be added to the subdivision.

c) $G_c \neq G$.

Because of c) and the fact that we add only (collections of) chains, there is at least one chain that is not added. W.l.o.g. it is a child of a chain that is already added.

There must be a non-branch node in the subdivision. Otherwise all vertices are branch vertices and every child of a chain that is already added can be added.

Let L be a link of length at least two (i.e. it contains at least one non-branch vertex). Because of a), there must be a path P in $G - G_c$ (vertex disjoint from G_c) that connects a vertex on L, to a vertex of the subdivision that is not on L (including its endpoints). If no such path exists, the endpoints of L form a cut.

Note that P is a Mader-path for G_c. Let a and b be the endpoints of P. Let z be the LCA of all vertices on P. Since DFS trees do not contain cross edges, z is equal to a or b. Say a. We view P as oriented from a to b. Let v be the first node of P.

Note that by construction, every path in G_c between a and b contains a branch vertex as an inner vertex, since we chose at least one of them to be a non-branch vertex.

The tree path $b \to_T a$ is part of G_c and hence disjoint from P. The vertex v must be a descendant of b, otherwise P contains a cross edge. Since $a < b$, $a\,v$ is a back-edge and starts a chain, D. D is not yet added, since P is disjoint from G_c.

Let $y\,x$ be the last edge on the path $t(D) \to_T b$ that is not in G_c and let D^* be the chain that contains $y\,x$. Note that $y\,x$ need not be an edge of P, if the last edge of P is a back-edge. Since $y\,x$ is a tree-edge, D^* does not consist of a single back-edge. We claim that if G is 3-connected either D^* can be added alone, or D^* is part of a caterpillar that can be added.

By Lemma 2.4, $D^* \leq D$ and hence $s(D^*) \leq s(D) \leq a$.

[5]The attachment points of a structure S to a subdivision G_c are the vertices in $S \cap G_c$.

Claim: D^* is either addable or part of an addable caterpillar.

There is a branch vertex on $b \to_T a$ and a and b lie on different links by our choice of P. Since $s(D^*) \leq s(D) = a$ and $t(D^*) = x \leq b$, there is a branch vertex on $t(D^*) \to_T s(D^*)$. If D^* is a Mader-path, we can thus add with with Lemma 2.25.

Assume then that D^* is not a Mader-path. The vertices $s(D^*)$ and $t(D^*)$ do not lie on the same link because of the branch vertex on $t(D^*) \to_T s(D^*)$ and neither do they lie on parallel links, since P is a Mader-path.

The only possibility left is that $s(D^*)$ and $t(D^*)$ lie on links that share one endpoint. Then $s(D^*) = s(\widehat{D^*})$ and D^* is part of a caterpillar. W.l.o.g. we can assume that D^* is the minimal chain in the caterpillar.

Now we show that we can add this caterpillar.

The path P is a Mader-path that uses an edge of the caterpillar. C_k is already added, hence $s(C_k) \leq \text{lca}(a, b)$. Moreover $a, b \in C_k \cup t(C_k) \to_t s(C_k)$. Therefore $a = s(C_k)$.

If $s(D)$ is on $s(C_k) \to_T t(C_k)$, then the caterpillar is good and can be added. Otherwise $s(D)$ is on C_k. Since D is the minimal interlacing chain that enters the caterpillar, $s(D) \leq b$. As P is a Mader-path, the caterpillar is good. □

It easily follows from this theorem that if the algorithm does not find a construction sequence and instead stops at some subdivision G_c, the endpoints of any link of size at least three in this subdivision form a cut in the graph.

By the argument from Section 2.6, the runtime of this algorithm is also $O((n + m) \log(n + m))$.

2.15 Conclusion

We presented a certifying linear time algorithm for 3-edge-connectivity based on chain decompositions of graphs. It is simple enough for use in a classroom setting and can serve as a gentle introduction to the certifying 3-vertex-connectivity algorithm of [65]. We also provide an implementation in Python, available at `https://github.com/adrianN/edge-connectivity`. We also show how the algorithm from [65] can be simplified using a greedy approach.

We also show how to extend the algorithm to construct and certify a cactus representation of all 2-edge-cuts in the graph. From this representation the 3-edge-connected components can be readily read off. The same techniques are used to find the 3-vertex-connected components using the algorithm from [65], and thus present a certifying construction of SPQR-trees.

Mader's construction sequence is general enough to construct k-edge-connected graphs for any $k \geq 3$, and can thus be used in certifying algorithms for larger k. Even though Mader proves the existence of the sequences, it remains unclear how to compute them. We hope that the chain decomposition framework can be adapted to work in these cases too.

Chapter 3

Online Checkpointing with Improved Worst-Case Guarantees

3.1 Introduction

In the previous chapter, we discussed a certifying algorithm which, by design, provides protection against implementation errors and intermittent hardware faults like flipped memory bits. In many scenarios it is necessary to prepare for more faults that make it impossible to continue a computation, for example an interruption in power supply.

Checkpointing is a technique that allows reverting a long computation to an arbitrary previous state quickly by storing selected intermediate states and restarting the computation from a stored state (instead of redoing everything from the beginning). Checkpointing is one of the fundamental techniques in computer science. Classic results date back to the seventies [18], more recent topics are checkpointing in distributed [27], sensor network [60], or cloud [77] architectures.

Checkpointing usually involves a careful trade-off between the speed-up of reversions to previous states and the costs incurred by setting checkpoints (time, memory). Much of the classic literature (see [32] and the references therein) studies checkpointing with the focus of gaining fault tolerance against immediately detectable faults. Consequently, only reversions to the most recent checkpoint are needed. However, setting a checkpoint can be expensive, because the whole system state has to be copied to secondary memory. In such scenarios, the central question is how often to set a checkpoint such that the expected time spent on setting checkpoints and redoing computations from the last checkpoint is minimized (under a stochastic failure model and further, possibly time-dependent [71], assumptions on the cost of setting a checkpoint).

In this chapter, we will regard a checkpointing problem of a different nature. If not fault-tolerance of the system is the aim of checkpointing, then often the checkpoints can be kept in main memory. Applications of this type arise in data compression [10] and numerics [37, 68]. In such scenarios, the cost of setting a checkpoint is small compared to the cost of the regular computation. Consequently, the memory used by the stored checkpoints is the bottleneck. Memory can of course also become a bottleneck for fault-tolerance applications. A prominent example are backup solutions like Apple's TimeMachine. The memory for backups is limited, but one wants to have a record of the history that is as complete as possible.

The first to provide an abstract framework for memory-limited checkpointing independent of a particular application were Ahlroth, Pottonen and Schumacher [2]. They do not make assumptions on which reversion to previous states will be requested, but simply investigate how checkpoints can be set in an online fashion such that at all times their distribution is balanced over the total computation history.

They assume that the system is able to store up to k checkpoints (plus a free checkpoint at time 0). At any point in time, a previous checkpoint may be discarded and replaced by the current system state as new checkpoint. Costs incurred by such a change are ignored. However, as it turns out, good checkpointing algorithms do not set checkpoints very often. For all algorithms discussed in the remainder of this section, each checkpoint is changed only $O(\log T)$ times up to time T.

The max-ratio discrepancy measure. Each set of checkpoints, together with the current state and the state at time 0, partitions the time from the process start to

the current time T into $k+1$ disjoint intervals. Clearly, without further problem-specific information, an ideal set of checkpoints would lead to all these intervals having identical length. Of course, this is not possible at all points in time due to the restriction that new checkpoints can only be set on the current time. As discrepancy measure for a checkpointing algorithm, Ahlroth et al. mainly regard the *maximum gap ratio*, that is, the maximum ratio of the longest interval vs. the shortest interval (ignoring the last interval, which can be arbitrarily small), over all current times T. They show that there is a simple algorithm achieving a discrepancy of two: Start with all checkpoints placed evenly, e.g., at times $1, \ldots, k$. At an even time T, remove one of the checkpoints at an odd time and place it at T. This will lead to all checkpoints being at the even times $2, 4, \ldots, 2k$ when $T = 2k$ is reached. Since these checkpoints form a scaled copy of the initial ones, we can continue in this fashion forever. It is easy to see that at all times, the intervals formed by neighboring checkpoints have at most two different lengths, the larger being twice the smaller in case that not all lengths are equal. This shows the discrepancy of two.

It seems tempting to believe that one can do better, but, in fact, not much improvement is possible for general k as shown by the lower bound of $2^{1-1/\lceil (k+1)/2 \rceil} = 2(1 - o(1))$. For small values of k, namely $k = 2, 3, 4$, and 5, better upper bounds of approximately $1.414, 1.618, 1.755$, and 1.755, respectively, were shown.

The maximum distance discrepancy measure. In this work, we shall regard a different, and, as we find, more natural discrepancy measure. Recall that the actual cost of reverting to a particular state is basically the cost of redoing the computation from the preceding checkpoint to the desired point in time. Adopting a worst-case view on the time to revert to, our aim is to keep the length of the longest interval small (at all times). Note that with time progressing, the interval lengths necessarily grow. Hence a fair point of comparison is the length $T/(k+1)$ of a longest interval in the (at time T) optimal partition of the time frame into equal length intervals. For this reason, we say that a checkpointing algorithm (using k checkpoints) has *maximum distance discrepancy* (or simply discrepancy) q if it places the checkpoints in such a way that at all times T, the longest interval has length at most $qT/(k+1)$. We denote by $q^*(k)$ the infimum discrepancy among all checkpointing algorithms using k checkpoints.

This maximum distance discrepancy measure was suggested in [2]. There it was remarked that an upper bound of β for the gap-ratio discrepancy implies an upper bound of $\beta(1 + \frac{1}{k})$ for the maximum distance discrepancy. Furthermore, for all k an upper bound of 2 and a lower bound of $1 + \frac{1}{k}$ is shown for $q^*(k)$. For $k = 2, 3, 4$, and 5, stronger upper bounds of $1.785, 1.789, 1.624$, and 1.565, respectively, were shown.

Our results. In this section, we show that the optimal discrepancy $q^*(k)$ is asymptotically bounded away from both one and two by a constant. We present algorithms that achieve a discrepancy of $1.59 + O(1/k)$ for all k (Theorem 3.5), and a discrepancy of $\ln(4) + o(1) \leq 1.39 + o(1)$ for k being any power of two (Theorem 3.6). For small values of k, and this might be an interesting case in applications with memory-consuming states, we show superior bounds by suggesting a class of checkpointing algorithms and optimizing their parameters via a combination

of exhaustive search and linear programming (Table 3.1). Experiments suggest $q^*(k) \leq 1.7$ for all k (Sect. 3.6). We complement these constructive results by a lower bound for $q^*(k)$ of $2 - \ln(2) - O(1/k) \geq 1.3 - O(1/k)$ (Theorem 3.17). We round off this work with a natural, but seemingly nontrivial result: We show that for each k there is indeed a checkpointing algorithm having discrepancy $q^*(k)$ (Theorem 3.10). In other words, the infimum in the definition of $q^*(k)$ can be replaced by a minimum.

The results of this chapter are joint work with Karl Bringmann, Benjamin Doerr, and Jacub Sliacan [17].

3.2 Notation and Preliminaries

In the checkpointing problem with $k \geq 2$ checkpoints, we consider a long running computation during which we can choose to replace an old checkpoint with the current state. We assume that our storage can hold at most k checkpoints simultaneously, and that there are implicit checkpoints at time $t = 0$ and the current time. We ignore any costs for placing or maintaining checkpoints. Consequently, we may assume that we only delete a previous checkpoint when a new one is placed.

A checkpointing algorithm for checkpoint placement can be described by two infinite sequences. First, the time points where new checkpoints are placed, i.e., a non-decreasing infinite sequence of reals $t_1 \leq t_2 \leq \ldots$ such that $\lim_{i\to\infty} t_i = \infty$, and second, a rule that describes which old checkpoints to delete when a new one is installed, that is, an injective function $d : [k+1..\infty) \to \mathbb{N}$ satisfying $d_i < i$ for all $i \geq k+1$.

The algorithm A described by (t, d) will start with $t_1, \ldots, t_k$ as initial checkpoints and then for each $i \geq k+1$, at time t_i remove the checkpoint at t_{d_i} and set a new checkpoint at the current time t_i. We call the act of removing a checkpoint and placing a new one a *step* of A. Note that there is little point in setting the first k checkpoints to zero, so to make the following discrepancy measure meaningful, we shall always require that $t_k > 0$.

We call the checkpoints that exist at time T *active*. The active checkpoints, together with the two implicit checkpoints at times 0 and T, define a sequence of $k+1$ interval lengths $\mathcal{L}_T = (\ell_0, \ldots, \ell_k)$. The *discrepancy* $q(A, T)$ of an algorithm A at time $T \geq t_k$ is a measure of how long the maximal interval is, normalized to be one if all intervals have the same length. It is calculated as

$$q(A, T) := (k+1)\bar{\ell}_T / T,$$

where $\bar{\ell}_T = ||\mathcal{L}_T||_\infty$ denotes the length of the longest interval. We also use the term *discrepancy* when we refer to the scaled length of a single interval.

The discrepancy $\mathrm{Discr}(A)$ of an algorithm A then is the supremum over the discrepancy over all times T, i.e.,

$$\mathrm{Discr}(A) := \sup_{T \geq t_k} q(A, T).$$

Hence the discrepancy of an algorithm would be 1, if it kept its checkpoints evenly distributed at all times. We denote the infimum discrepancy of a checkpointing

algorithm using k checkpoints by

$$q^*(k) := \inf_A \mathrm{Discr}(A),$$

where A runs over all algorithms using k checkpoints. We will see in Sect. 3.7 that algorithms achieving this discrepancy actually exist.

Note that we allow checkpointing algorithms to work in a continuous time scale. One can convert any such algorithm to an algorithm with integral checkpoints by rounding down all checkpointing times t_i. Our bounds for the discrepancy accurately bound the number of recomputation steps in this setting, since $\lfloor t_i \rfloor - \lfloor t_{i-1} \rfloor \leq t_i - t_{i-1} + 1$, but with discrete time there are at most $\lfloor t_i \rfloor - \lfloor t_{i-1} \rfloor - 1$ steps to recompute in this interval.

In the definition of the discrepancy, the supremum is never attained at some T with $t_i < T < t_{i+1}$ for any i, as shown in the following lemma.

Lemma 3.1: In the definition of the discrepancy it suffices to consider times $T = t_i$ for all $i \geq k$, i.e., we have

$$\mathrm{Discr}(A) = \sup_{i \geq k} q(A, t_i).$$

Proof. Let T be a time with $t_i < T < t_{i+1}$ for some $i \geq k$. We show that

$$q(A, T) \leq \max\{q(A, t_i), q(A, t_{i+1})\}.$$

Denote the active checkpoints at time T by $x_1, \ldots, x_k$. Note that $x_k = t_i$, since t_i was the last time we set a checkpoint. Consider the last interval $I_k = [x_k, T]$. Its discrepancy is

$$(k+1)\frac{T - x_k}{T} \leq (k+1)\frac{t_{i+1} - x_k}{t_{i+1}} \leq q(A, t_{i+1}).$$

Any other interval at time T is of the form $I_{j-1} = [x_{j-1}, x_j]$ for some $1 \leq j \leq k$ (where we set $x_0 := 0$). Its discrepancy is

$$(k+1)\frac{x_j - x_{j-1}}{T} \leq (k+1)\frac{x_j - x_{j-1}}{t_i} \leq q(A, t_i).$$

This proves the claim. □

To bound the discrepancy of an algorithm we need to bound the largest of the $q(A, t_i)$ over all $i \geq k$. For this purpose, it suffices to look at the two newly created intervals at time t_i for each i, as made explicit by the following lemma.

Lemma 3.2: Let $i > k$ and let ℓ_1, ℓ_2 be the lengths of the two newly created intervals at time t_i due to the removal and the insertion of a checkpoint. Then

$$\max\{q(A, t_{i-1}), q(A, t_i)\} = \max\left\{q(A, t_{i-1}), \frac{(k+1)\ell_1}{t_i}, \frac{(k+1)\ell_2}{t_i}\right\}.$$

Proof. If ℓ_1 or ℓ_2 is the longest interval at time t_i the claim holds. Any other interval existed already at time t_{i-1} and had a larger discrepancy at this time, as we divide by the current time to compute the discrepancy. Thus, if any other interval is the longest at time t_i, then we have $q(A, t_{i-1}) \geq q(A, t_i)$ and the claim holds again. □

Lemma 3.3: We can assume without loss of generality that $t_k = 1$ for any algorithm.

Proof. Let $A = (t, d)$ be an algorithm such that $t_k \neq 1$. Then there is an algorithm $A' = (t', d)$ with $t'_i = t_i / t_k$ such that $\mathrm{Discr}(A) = \mathrm{Discr}(A')$ and $t'_k = 1$. This can be easily seen as our discrepancy measure is invariant under scaling. □

Often, it will be useful to use a different notation for the checkpoint that is removed in step i. Instead of the global index d, one can also use the index $p : [k+1..\infty) \to [1..k]$ among the active checkpoints, i.e.,

$$p_i = d_i - |\{j \in [1..i-1] \mid d_j < d_i\}|.$$

We call an algorithm $A = (t, p)$ *cyclic*, if the p_i are periodic with some period n, i.e., $p_i = p_{i+n}$ for all i, and after n steps A has transformed the intervals to a scaled version of themselves, that is, $\mathcal{L}_{t_{k+jn}} = \gamma^j \mathcal{L}_{t_k}$ for some $\gamma > 1$ and all $j \in \mathbb{N}$. We call γ the *scaling factor*. For a cyclic algorithm A, it suffices to fix the *pattern* of removals $P = (p_{k+1}, \ldots, p_{k+n})$ and the checkpoint positions $t_1, \ldots, t_k, t_{k+1}, \ldots, t_{k+n}$, where $t_k = 1$ and $t_{k+n} = \gamma$.

Since cyclic algorithms transform the starting position to a scaled copy of itself, it is easy to see that their discrepancy is given by the maximum over the discrepancies during one period, i.e., for cyclic algorithms A with period n we have

$$\mathrm{Discr}(A) = \max_{i \in [k+1..k+n]} q(A, t_i).$$

This makes this class of algorithms easy to analyze.

3.3 Introductory Example—A Simple Bound for $k = 3$

For the case of $k = 3$ there is a very simple algorithm, SIMPLE, with a discrepancy of $4/\phi^2 \approx 1.53$, where $\phi = (\sqrt{5}+1)/2$ is the golden ratio. We use it to familiarize ourselves with the notation we introduced in Sect. 3.2. The algorithm is cyclic with a pattern of length one. We prove the following theorem.

Theorem 3.4: For $k = 3$ there is a cyclic algorithm SIMPLE with period length one and

$$\mathrm{Discr}(\textsc{Simple}) = \frac{4}{\phi^2}.$$

Proof. We fix the pattern to be $P = (1)$, that is, the algorithm SIMPLE always removes the oldest checkpoint. For this simple pattern it is easy to calculate the discrepancy depending on the scaling factor γ. Since the intervals need to be a scaled copy of themselves after just one step and we can fix $t_3 = 1$ by Lemma 3.3, we know immediately that

$$t_1 = \frac{1}{\gamma^2},\ t_2 = \frac{1}{\gamma},\ t_3 = 1,\ t_4 = \gamma.$$

Hence, the discrepancy is determined by considering the situation at time t_4, where we just deleted the checkpoint at t_1,

$$4 \cdot \max\left\{\frac{t_2-0}{t_4}, \frac{t_3-t_2}{t_4}, \frac{t_4-t_3}{t_4}\right\} = 4 \cdot \max\left\{\frac{1}{\gamma^2}, \frac{\gamma-1}{\gamma^2}, \frac{\gamma-1}{\gamma}\right\}.$$

Since $\gamma > 1$, the second term is always smaller than the third and can be ignored. As $1/\gamma^2$ is decreasing and $(\gamma-1)/\gamma$ is increasing, the maximum is minimal when they are equal. A simple calculation shows this to be the case at $\gamma = \phi$.

Hence for $k = 3$ the algorithm with pattern (1) and checkpoint positions $t_1 = 1/\phi^2$, $t_2 = 1/\phi$, $t_3 = 1$, and $t_4 = \phi$ has discrepancy $4/\phi^2 \approx 1.53$. □

The experiments in Sect. 3.6 indicate that for $k = 3$ this is optimal among all cyclic algorithms with a period of length at most 6.

3.4 An Upper Bound for Large k

In this section we present an algorithm, LINEAR, with a discrepancy of roughly 1.59 for large k. This improves upon the asymptotic bound of 2 from [2].

Like the algorithm SIMPLE of the previous section, the algorithm LINEAR is cyclic. It has a simple pattern of length k. The pattern is just $(1, \ldots, k)$, that is, at the i-th step of a period LINEAR deletes the i-th active checkpoint. Overall, during one period LINEAR removes all checkpoints at times t_i with odd index i, as shown in Figure 3.1.

This removal pattern is identical to the one of the POWERS-OF-TWO algorithm from [2]. However, that algorithm starts with a uniform checkpoint distribution where removing any checkpoint doubles the maximum interval. This leads to an asymptotic discrepancy of two. In contrast, LINEAR places checkpoints using a polynomial function. For $i \in [1..2k]$ we set $t_i = (i/k)^\alpha$, where α is a constant. In the analysis we optimize the choice of α and set $\alpha := 1.302$. For this algorithm we show the following theorem.

Theorem 3.5: The algorithm LINEAR has a discrepancy of at most

$$\mathrm{Discr}(\text{LINEAR}) \leq 1.586 + O(k^{-1}).$$

Experiments show that the discrepancy of the algorithm LINEAR is close to the bound of 1.586 even for moderate sizes of k. Comparisons using the optimization method from Sect. 3.6 indicate that for the pattern $(1, \ldots, k)$ of algorithm LINEAR, different checkpoint placements can yield only improvements of about 4.5% for large k. Experimental results are summarized in Figure 3.4 on page 57.

Proof. As the algorithm LINEAR is cyclic, we can again compute the discrepancy from the $2k$ checkpoint positions and the pattern,

$$\mathrm{Discr}(\text{LINEAR}) = \max_{k<i\leq 2k} (k+1)\bar{\ell}_{t_i}/t_i,$$

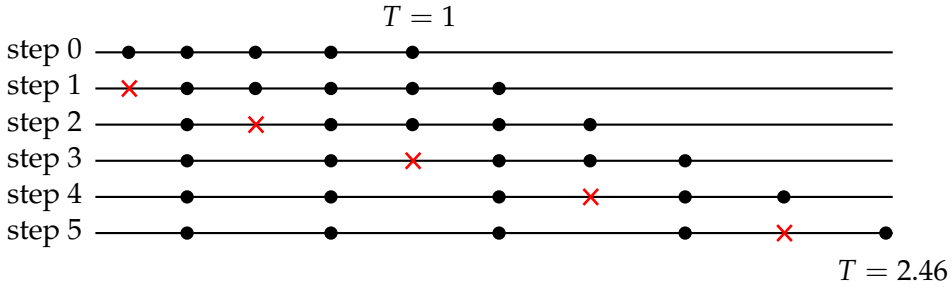


Figure 3.1: One period of the algorithm LINEAR from Sect. 3.4 for $k = 5$. After one period all intervals are scaled by the same factor.

where $\bar{\ell}_{t_i}$ is the length of the longest interval at time t_i. By Lemma 3.2 it suffices to consider newly created intervals at times $t_{k+1}, \ldots, t_{2k}$. Note that at time t_i we create the intervals $[t_{i-1}, t_i]$ (from insertion of a checkpoint at t_i) and $[t_{2(i-k)-2}, t_{2(i-k)}]$ (from deletion of the checkpoint at $t_{2(i-k)-1}$). The discrepancy of the new interval by insertion is, for $k < i \leq 2k$,

$$(k+1)\frac{t_i - t_{i-1}}{t_i} = (k+1)\frac{i^\alpha - (i-1)^\alpha}{i^\alpha} \leq (k+1)\frac{(k+1)^\alpha - k^\alpha}{(k+1)^\alpha}.$$

Using $(x+1)^c - x^c \leq c(x+1)^{c-1}$ for any $x \geq 0$ and $c \geq 1$, this simplifies to

$$(k+1)\frac{t_i - t_{i-1}}{t_i} \leq (k+1)\frac{\alpha(k+1)^{\alpha-1}}{(k+1)^\alpha} = \alpha,$$

for any constant $\alpha \geq 1$.

For the new interval from deleting the checkpoint at $t_{2(i-k)-1}$ we get a discrepancy of

$$\begin{aligned}(k+1)\frac{t_{2(i-k)} - t_{2(i-k)-2}}{t_i} &= (k+1)\frac{(2(i-k))^\alpha - (2(i-k)-2)^\alpha}{i^\alpha} \\ &\leq (k+1)2^\alpha \frac{\alpha(i-k)^{\alpha-1}}{i^\alpha},\end{aligned}$$

where we used again $(x+1)^c - x^c \leq c(x+1)^c$. An easy computation shows that $(i-k)^{\alpha-1}/i^\alpha$ is maximized at $i = \alpha k$ over $k < i \leq 2k$. Hence, we can bound this discrepancy by

$$(k+1)\frac{t_{2(i-k)} - t_{2(i-k)-2}}{t_i} \leq \left(1 + \frac{1}{k}\right)2^\alpha \frac{\alpha(\alpha-1)^{\alpha-1}}{\alpha^\alpha} = 2^\alpha \left(1 - \frac{1}{\alpha}\right)^{\alpha-1} + O(\tfrac{1}{k}).$$

We optimize the latter term numerically and obtain for $\alpha = 1.302$ an upper bound of

$$1.586 + O(k^{-1}).$$

Note that this bound is larger than the bound $\alpha = 1.302$ from the new intervals from insertion. Hence, overall we get the desired upper bound. □

3.5 An Improved Upper Bound for Large k

In this section we present the algorithm BINARY that yields a discrepancy of roughly $\ln(4) \approx 1.39$ for large k. Compared to the algorithm LINEAR from the last section, BINARY has a considerably better discrepancy at the price of a more involved analysis, and it only works for k being a power of two. Algorithm BINARY is cyclic with period length $k/2$.

Theorem 3.6: For $k \geq 8$ being any power of 2, the algorithm BINARY has discrepancy

$$\mathrm{Discr}(\textsc{Binary}) \leq \ln(4) + \frac{0.05}{\lg(k/4)} + O\Big(\frac{1}{k}\Big).$$

Here and in the remainder of this chapter, let 'lg' denote the binary and 'ln' the natural logarithm. Note that the term $O(1/k)$ quickly tends to 0, whereas the $\Theta(1/\lg(k/4))$ term is small due to the constant 0.05. Hence, this discrepancy is close to $\ln(4)$ already for moderate k. Also note that $\ln(4)$ is less than 0.1 larger than our lower bound from Sect. 3.8, leaving room for less than a 6% improvement over the upper bound for algorithm BINARY for large k. We verified experimentally that the algorithm BINARY yields very good bounds already for relatively small k. The results are summarized in Figure 3.5 on page 57.

3.5.1 The Algorithm BINARY

The initial checkpoints $t_1, \ldots, t_k$ satisfy the equation

$$t_i = \alpha t_{i/2} \tag{3.5.1}$$

for each even $1 \leq i \leq k$ and some $\alpha = \alpha(k) \geq 2$. Precisely, we set

$$\alpha := 2^{1+\frac{\lg(\sqrt{2}/\ln 4)}{\lg(k/4)}} \approx 2^{1+\frac{0.029}{\lg(k/4)}}.$$

However, the usefulness of this expression becomes clear only in the analysis of the algorithm.

During one period we delete all odd checkpoints $t_1, t_3, \ldots, t_{k-1}$ and insert the new checkpoints

$$t_{k+i} := \alpha t_{k/2+i}, \tag{3.5.2}$$

for $1 \leq i \leq k/2$. Then after one period we end up with the checkpoints

$$= \alpha \cdot \begin{array}{l} (t_2, t_4, \ldots, t_{k-2}\ , t_k\ ,\ t_{k+1}\ , t_{k+2}\ , \ldots, t_{k+k/2}) \\ (t_1, t_2, \ldots, t_{k/2-1}, t_{k/2},\ t_{k/2+1}, t_{k/2+2}, \ldots, t_{k/2+k/2}) \end{array} = \alpha(t_1, t_2, \ldots, t_k),$$

which proves cyclicity. Note that (3.5.1) and (3.5.2) allow us to compute all t_i from the values $t_{k/2+1}, \ldots, t_k$, however, we still have some freedom to choose the latter values. By Lemma 3.3 we can set $t_k := 1$, then $t_{k/2} = \alpha^{-1}$. In between these two values, we interpolate $\lg t_i$ linearly, i.e., we set for $i \in (k/2..k]$

$$t_i := \alpha^{2i/k-2}, \tag{3.5.3}$$

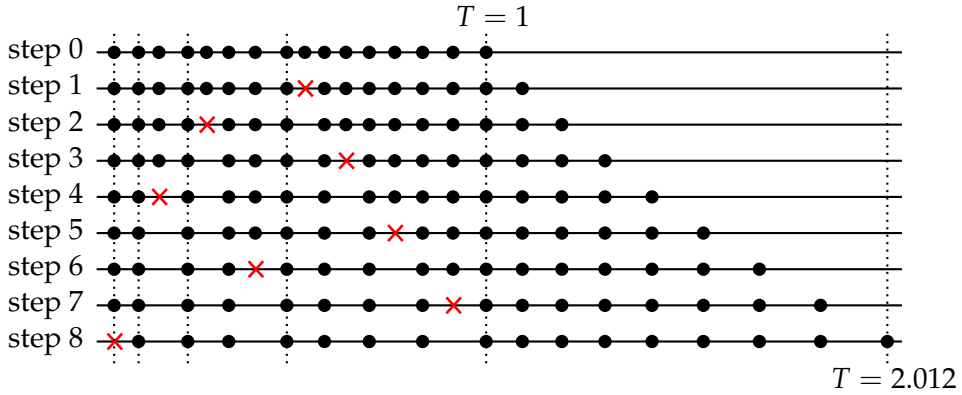


Figure 3.2: One period of the algorithm BINARY for $k = 16$. Note that, recursively, checkpoints are removed twice as often from the right half of the initial setting (at steps i where $i \equiv 1 \pmod 2$) as from the second quarter.

completing the definition of the t_i. Note that this equation also works for $i = k$ and $i = k/2$.

There is one more freedom we have with this algorithm, namely in which order we delete all odd checkpoints during one period, i.e., we need to fix the pattern of removals. In iteration $1 \leq i \leq k/2$ we insert the checkpoint t_{k+i} and remove the checkpoint $t_{d(i+k)}$, defined as follows. For $m \in \mathbb{N} = \mathbb{N}_{\geq 1}$ let $2^{\sigma(m)}$ be the largest power of 2 that divides m. We define $S\colon \mathbb{N} \to \mathbb{N}, S(m) := m/2^{\sigma(m)}$. Note that $S(m)$ is an odd integer. Using this definition, we set

$$d(k+i) := S\Big(\frac{k}{2} + i\Big), \tag{3.5.4}$$

finishing the definition of the algorithm BINARY. If we write this down as a pattern, then we have $p_i = 1 + k/(2^{1+\sigma(i)})$ for $1 \leq i < k/2$ and $p_{k/2} = 1$. The example in Figure 3.2 provides intuition for this pattern.

The following lemma implies that during one period we delete all odd checkpoints $t_1, t_3, \ldots, t_{k-1}$ (and no point is deleted twice).

Lemma 3.7: The function S induces a bijection between $\{k/2 < i \leq k\}$ and $\{1 \leq i \leq k \mid i \text{ is odd}\}$.

Proof. Let $A := \{k/2 < i \leq k\}$ and $B := \{1 \leq i \leq k \mid i \text{ is odd}\}$. Since $S(m) \leq m$ and $S(m)$ is odd for all $m \in \mathbb{N}$, we have $S(A) \subseteq B$. Moreover, A and B are of the same size. We present an inverse function to finish the proof. Let $x \in B$. Note that there is a unique number $y \in \mathbb{N}$ such that $x2^y \in A$, since A is a range between two consecutive powers of 2 and $x \leq k$. Setting $S^{-1}(x) = x2^y$ we have found the inverse. □

3.5.2 Discrepancy Analysis

We now bound the largest discrepancy encountered during one period, i.e.,

$$\text{Discr}(\text{BINARY}) = \max_{1 \le i \le k/2} q(\text{BINARY}, t_{k+i}) = (k+1) \max_{1 \le i \le k/2} \overline{\ell}_{t_{k+i}} / t_{k+i}.$$

We first compute the maximum and later multiply with the factor $k+1$. By Lemma 3.2, we only have to consider intervals newly created by insertion and deletion at any step.

Intervals from Insertion: We first compute the discrepancy of the interval newly added at time t_{k+i}, $1 \le i \le k/2$. Its length is $t_{k+i} - t_{k+i-1}$, so its discrepancy (without the factor $k+1$) is

$$\begin{aligned} \frac{t_{k+i} - t_{k+i-1}}{t_{k+i}} &= 1 - \frac{t_{k+i-1}}{t_{k+i}} \\ &= 1 - \frac{t_{k/2+i-1}}{t_{k/2+i}} \\ &\overset{(3.5.3)}{=} 1 - \alpha^{-2/k}, \end{aligned}$$

where the second equality holds because of (3.5.2) if $i > 1$ or (3.5.1) if $i = 1$. Using $e^x \ge 1 + x$ for $x \in \mathbb{R}$ yields a bound on the discrepancy of

$$\frac{t_{k+i} - t_{k+i-1}}{t_{k+i}} \le \ln(\alpha^{2/k}) = \frac{1}{k} \ln(\alpha^2)).$$

Deleting t_1: We show similar bounds for the intervals we get from deleting an old checkpoint. We first analyze the deletion of t_1. This case needs separate treatment, since t_1 has no predecessor. Note that t_1 is deleted in step $k/2$ at time $t_{3k/2}$. The deletion of t_1 creates the interval $[0, t_2]$. This interval has discrepancy

$$\frac{t_2}{t_{3k/2}} \overset{(3.5.2),(3.5.1)}{=} \frac{\alpha t_1}{\alpha t_k} \overset{(3.5.1)}{=} \alpha^{-\lg k} \le 1/k,$$

since we choose $\alpha \ge 2$. Hence, this discrepancy is dominated by the one we get from newly inserted intervals.

Other Intervals from Deletion: It remains to analyze the discrepancy of the intervals we get from deletion in the general case, i.e., at some time t_{k+i}, $1 \le i < k/2$. At this time we delete the checkpoint $d(k+i)$, so we create the interval $[t_{d(k+i)-1}, t_{d(k+i)+1}]$ of discrepancy

$$q_i := \frac{t_{d(k+i)+1} - t_{d(k+i)-1}}{t_{k+i}} \overset{(3.5.2),(3.5.4)}{=} \frac{t_{S(k/2+i)+1} - t_{S(k/2+i)-1}}{\alpha t_{k/2+i}}.$$

Let $h := \sigma(k/2+i)$, so that 2^h is the largest power of 2 dividing $k/2+i$, and $2^h S(k/2+i) = k/2+i$. Then $t_{S(k/2+i)+1} = \alpha^{-h} t_{k/2+i+2^h}$ by (3.5.1), and a similar statement holds for $t_{S(k/2+i)-1}$, yielding

$$q_i = \alpha^{-1-h} \frac{t_{k/2+i+2^h} - t_{k/2+i-2^h}}{t_{k/2+i}}.$$

Using (3.5.3) we get $t_{k/2+i} = \alpha^{2i/k-1}$. Comparing this with the respective terms for $t_{k/2+i+2^h}$ and $t_{k/2+i-2^h}$ yields

$$\begin{aligned} q_i &= \alpha^{-1-h}\left(\alpha^{2^{h+1}/k} - \alpha^{-2^{h+1}/k}\right) \\ &= \alpha^{-1-h} \cdot 2\sinh\left(\ln\left(\alpha^2\right) 2^h/k\right). \end{aligned}$$

By elementary means one can show that the function $f(x) = x^{-A}\sinh(Bx)$, $A \geq 1, B > 0$, is convex on $\mathbb{R}_{\geq 0}$. Since convex functions have their maxima at the boundaries of their domain, and since by the above equation q_i can be expressed using $f(2^h)$ (for $A = \lg \alpha$ and $B = \ln(\alpha^2)/k$), we see that q_i is maximal at (one of) the boundaries of h. Recall that we treated $i = k/2$ separately, and observe that the largest power of 2 dividing $k/2 + i$, $1 \leq i < k/2$ is at most $k/4$. Hence, we have $0 \leq 2^h \leq k/4$ and

$$q_i \leq \max\left\{2\alpha^{-1}\sinh(\ln(\alpha^2)/k), 2\alpha^{-1}(k/4)^{-\lg\alpha}\sinh(\ln(\alpha)/2)\right\}.$$

We simplify using $\alpha \geq 2$ and $\sinh(x) = x + O(x^2)$ to get

$$q_i \leq \max\left\{\ln(\alpha^2)/k + O(1/k^2), (k/4)^{-\lg\alpha}\sinh(\ln(\alpha)/2)\right\}. \tag{3.5.5}$$

The first term is already of the desired form. For the second one, note that by setting $\alpha = 2$ we would get a discrepancy of $4\sinh(\ln(2)/2)/k = \sqrt{2}/k$. We get a better bound by choosing

$$\alpha := 2^{1+\frac{c}{\lg(k/4)}},$$

with $c := \lg(\sqrt{2}/\ln(4)) \approx 0.029$. Then the second bound on q_i from above becomes

$$(k/4)^{-\lg\alpha}\sinh(\ln(\alpha)/2) = \frac{4}{k}2^{-c}\sinh\left(\frac{\ln(2)}{2}\left(1 + \frac{c}{\lg(k/4)}\right)\right).$$

The particular choice of c allows to bound the derivative of $\sinh((1+x)\ln(2)/2)$ for $x \in [0, c]$ from above by

$$\frac{\ln(2)}{2}\cosh((1+c)\ln(2)/2) < 0.39.$$

This yields

$$\sinh\left(\frac{\ln(2)}{2}\left(1 + \frac{c}{\lg(k/4)}\right)\right) \leq \sinh(\ln(2)/2) + \frac{0.39c}{\lg(k/4)}.$$

Thus, in total the second bound on q_i from inequality (3.5.5) becomes

$$(k/4)^{-\lg\alpha}\sinh(\ln(\alpha)/2) \leq \frac{4}{k}2^{-c}\sinh(\ln(2)/2) + \frac{4 \cdot 2^{-c} \cdot 0.39c}{k\lg(k/4)}.$$

Since $c = \lg(\sqrt{2}/\ln(4)) = \lg(4\sinh(\ln(2)/2)/\ln(4))$, this becomes

$$\leq \ln(4)/k + 0.044/(k\lg(k/4)).$$

Overall discrepancy: In total, we can bound the discrepancy $q := \text{Discr}(\text{BINARY})$ of our algorithm (now including the factor of $k+1$) by

$$q \leq (k+1) \max \left\{ \ln(\alpha^2)/k + O(1/k^2), \ln(4)/k + 0.044/(k \lg(k/4)) \right\}.$$

Using $(k+1)/k = 1 + O(1/k)$ and

$$\ln(\alpha^2) = \ln(4) \left(1 + \frac{c}{\lg(k/4)}\right) \leq \ln(4) + \frac{0.040}{\lg(k/4)},$$

this bound can be simplified to

$$q \leq \max\{\ln(4) + 0.040/\lg(k/4) + O(1/k), \ln(4) + 0.044/\lg(k/4) + O(1/k)\},$$

which proves Theorem 3.6.

3.6 Upper Bounds via Combinatorial Optimization

In this section, we show how to find upper bounds on the optimal discrepancy $q^*(k)$ for fixed k. We do so by constructing cyclic algorithms using an exhaustive enumeration of all short patterns in the case of very small k or randomized local search on the patterns for larger k, combined with linear programming to optimize the checkpoint positions. This yields good algorithms as summarized in Table 3.1. In the following we describe our algorithmic approach.

Finding Checkpoint Positions: First we describe how to find a nearly optimal cyclic algorithm for a given pattern P and a scaling factor γ, i.e., how to optimize the checkpoint positions. To do so, we construct a linear program that is feasible if a cyclic algorithm with discrepancy λ and scaling factor γ exists. We use three kinds of constraints: We fix the ordering of the checkpoints, enforce that the i-th active checkpoint after one period is a factor γ larger than the i-th initial checkpoint, and bound the discrepancy of each interval during the period from above by λ. We then use binary search to optimize λ.

Lemma 3.8: For a fixed pattern P of length n and scaling factor γ, let $q^* = \inf_A \text{Discr}(A)$ be the optimal discrepancy among algorithms A using P and γ. Then finding an algorithm with discrepancy at most $q^* + \epsilon$ reduces to solving $O(\log \epsilon^{-1})$ linear feasibility problems with $O(nk)$ inequalities and $k+n$ variables.

Proof. For a fixed pattern and scaling factor, we can tune the discrepancy of the algorithm by cleverly choosing the time points when to remove an old checkpoint and place a new one. By solving a linear feasibility problem we can check whether a cyclic algorithm with scaling factor γ and pattern P exists that guarantees a discrepancy of at most λ. We can then optimize over λ to find an approximately optimal algorithm.

We construct a linear program with the $k+n$ time points $(t_1, \ldots, t_{k+n})$ as variables (where we can set $t_k = 1$ without loss of generality). It uses three kinds of constraints. The first kind is of the form

$$t_i \leq t_{i+1},$$

for all $i \in [1..k+n)$. There are thus $n+k$ constraints of this form. These constraints are satisfied if the checkpoint positions have the correct ordering, i.e. checkpoints with larger index are placed at later times.

The second kind of constraints enforces the scaling factor. Since the pattern is fixed, we can compute at all steps which checkpoints are active. For $i \in [1..k]$ and $j \in [0..n]$, let τ_i^j be the variable of the i-th active checkpoint in step j and let τ_0^j be 0 for all j. It is easy to see that the algorithm has a scaling factor of γ if the i-th active checkpoint in the n-th step is by a factor of γ larger than in the first step. We encode this as k constraints of the form

$$\tau_i^n = \gamma \tau_i^0,$$

for all $i \in [1..k]$. Lastly we encode an upper bound of λ for the discrepancy. Since the discrepancy of a cyclic algorithm is given by

$$\max_{k<i\leq k+n} (k+1)\bar{\ell}_{t_i}/t_i,$$

and each $\bar{\ell}_{t_i}$ can be expressed by a maximum over k terms, we can encode a discrepancy guarantee of λ with nk constraints of the form

$$\tau_{i+1}^j - \tau_i^j \leq \lambda \tau_k^j/(k+1),$$

for all $i \in [0..k)$ and $j \in [0..n]$.

A feasible solution of this system of inequalities corresponds to a sequence of checkpoint positions and, together with the pattern P, provides an algorithm with discrepancy at most λ. Since algorithms with discrepancy 2 are known [2], we can restrict our attention to $\lambda \leq 2$. Using a simple binary search over $\lambda \in [1,2]$ we can find an approximately optimal algorithm for this value of γ and the pattern P. □

Finding Scaling Factors: Next we show how to find scaling factors γ for which algorithms with good discrepancy exist. We first show an upper bound for γ.

Lemma 3.9: A cyclic algorithm with k checkpoints, discrepancy $\lambda < k$, and a period length of n can have a scaling factor of at most

$$\gamma \leq \left(\frac{1}{1-\lambda/(k+1)}\right)^n.$$

Proof. Consider any checkpointing algorithm $A = (t, d)$ with k checkpoints and discrepancy λ. We bound the time the algorithm can delay setting a new checkpoint before the last interval violates the performance guarantee. At any time t_i, $i \geq k$, the largest interval has length $\bar{\ell}_{t_i} \geq t_i - t_{i-1}$, as there is no checkpoint in the time interval $[t_{i-1}, t_i]$. Hence, we have

$$(k+1)\frac{t_i - t_{i-1}}{t_i} \leq \lambda.$$

Rearranging, this yields

$$t_i \leq \frac{1}{1-\lambda/(k+1)} t_{i-1}.$$

k	3	4	5	6	7	8	9	10
Discr.	1.529	1.541	1.472	1.498	1.499	1.499	1.488	1.492
k	15	20	30	50	100			
Discr.	1.466	1.457	1.466	1.481	1.484			

Table 3.1: Upper bounds for different k. For $k < 8$ all patterns up to length k were tried. For $k = 8$ all patterns up to length 7 were tried. For larger k, patterns were found via randomized local search.

Iterating this n times, we get

$$t_{k+n} \leq \left(\frac{1}{1-\lambda/(k+1)}\right)^n t_k.$$

Hence, for any cyclic algorithm (with discrepancy λ, k checkpoints, and a period length of n) we get the desired bound on the scaling factor $\gamma = t_{k+n}/t_k$. □

For any given pattern length n, Lemma 3.9 yields an upper bound on γ, while a trivial lower bound is given by $\gamma > 1$. Now, for any given pattern P we optimize over γ using a linear search with a small step size over the possible values for γ. For each tested γ, we optimize over the checkpoint positions using the linear programming approach described above.

Finding Patterns: For small k and n, we can exhaustively enumerate all k^n removal patterns of period length n. Some patterns can be discarded as they obviously cannot lead to a good algorithm or are equivalent to some other pattern: No pattern that never removes the first checkpoint can be cyclic. Furthermore, patterns are equivalent under cyclic shifts, so we can assume without loss of generality that all patterns end with removing the first checkpoint. Lastly, it never makes sense to remove the currently last checkpoint. Hence, for k checkpoints there are at most $(k-1)^{n-1}$ interesting patterns of length n. This finishes the description of our combinatorial optimization approach.

Results: We ran experiments that exhaustively try all patterns of length n with $n \leq k$ for $k \in [3..7]$. For $k = 8$ we stopped the search after examining patterns of length 7. For larger k we used a randomized local search to find good patterns. The upper bounds we found are summarized in Table 3.1 , and for $k \leq 8$ the removal patterns and time points when to place new checkpoints can be found in Figure 3.3. Note that for $k = 3$ this procedure re-discovers the golden ratio algorithm of Sect. 3.3.

When we combine the results presented in Table 3.1 with the algorithm LINEAR (Theorem 3.5 and Figure 3.4), we can read off a global upper bound of $q^*(k) \leq 1.6$ for the optimal discrepancy for *any* k.

For a fixed pattern, the method is efficient enough to find good checkpoint positions for much larger k. For $k \leq 1000$ we experimentally compared the algorithm LINEAR of Sect. 3.4 with algorithms found for its pattern $(1, \ldots, k)$. The experiments show

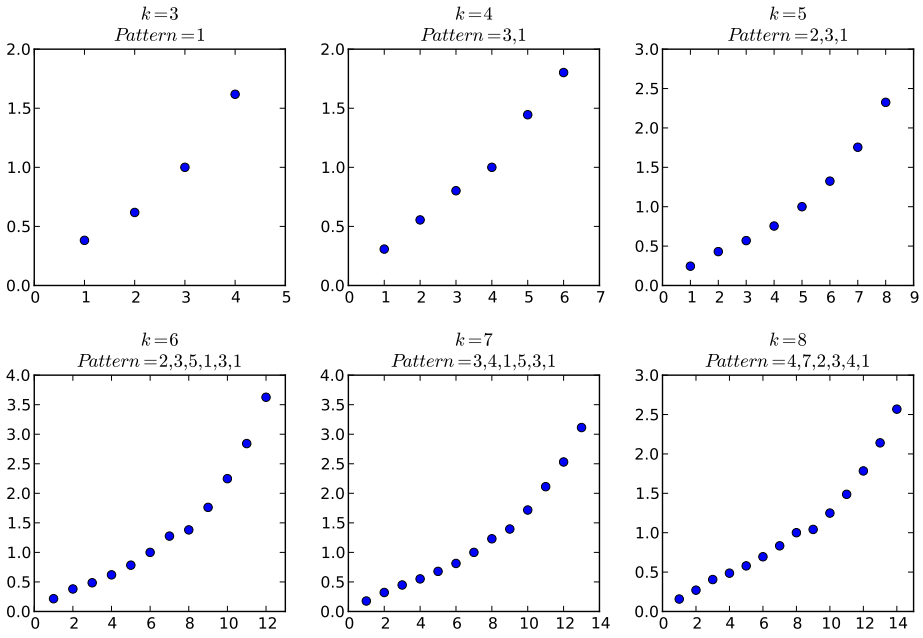


Figure 3.3: Time points where the i-th checkpoint is placed to achieve the bounds of Table 3.1. Time is on the y-axis, iteration is on the x-axis.

that for $k = 1000$ LINEAR is within 4.5% of the optimized bounds. For the algorithm BINARY of Sect. 3.5, this comparison is even more favorable. For $k = 1024$ the algorithm places its checkpoints so well that the optimization procedure improves discrepancy only by 1.9%. The results are summarized in Figure 3.4 and Figure 3.5.

Quality of the constructed algorithms. There are two steps in the above optimization algorithm that make it difficult to estimate how close the discrepancies in Table 3.1 are to the optimal ones. First, we are only optimizing over short patterns, and it might be that much larger pattern lengths are necessary for optimal checkpointing algorithms. Second, we do not know how smoothly the optimal discrepancy for fixed pattern P and scaling factor γ behaves with varying γ, i.e., we do not know whether our linear search for γ yields any approximation on the discrepancy λ. In experiments we tried all patterns of length n with $n \leq 2k$ for $k \in \{3, 4, 5\}$ and found no better algorithm than for the shorter patterns of length up to k. Moreover, smaller step sizes in the linear search for γ lead only to small improvements, indicating that the discrepancy is continuous in γ. This suggests that the reported algorithms might be near optimal.

3.7 Existence of Optimal Algorithms

In this section, we prove that optimal algorithms for the checkpointing problem exist, i.e., that there is an algorithm having discrepancy equal to the infimum

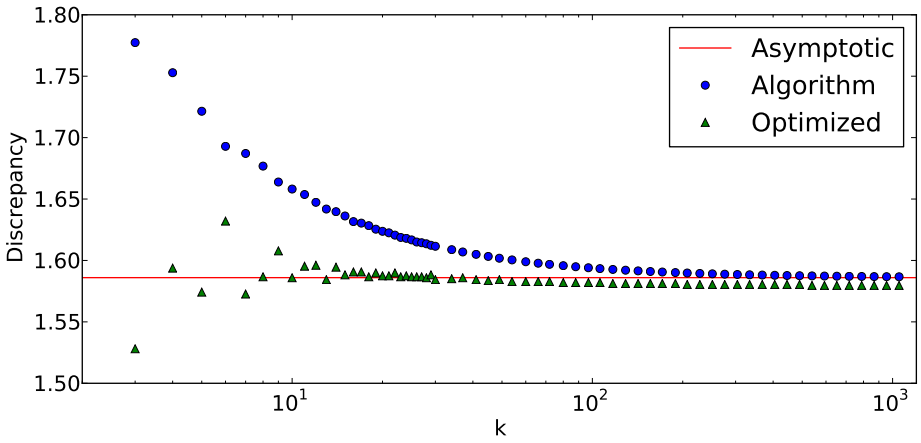


Figure 3.4: The discrepancy of algorithm LINEAR from Sect. 3.4 for different values of k compared to the upper bounds for its pattern found via the combinatorial method from Sect. 3.6. For large k LINEAR is about 4.5% worse.

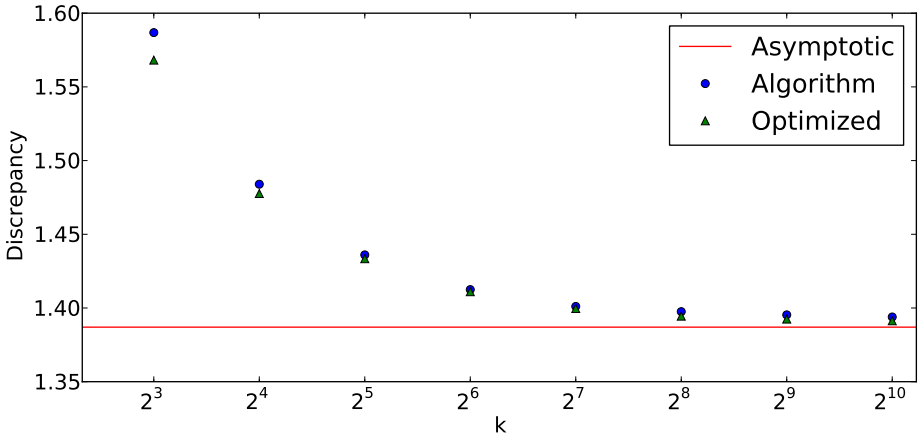


Figure 3.5: The discrepancy of algorithm BINARY from Sect. 3.5 for some values of k, compared to the upper bounds for its pattern found via the combinatorial method from Sect. 3.6. For $k = 1024$, the optimization procedure finds a checkpoint placement with only 1.9% better discrepancy.

discrepancy $q^*(k) := \inf_A \mathrm{Discr}(A)$ among all algorithms for k checkpoints.

Theorem 3.10: For each k there exists a checkpointing algorithm A for k checkpoints with $\mathrm{Discr}(A) = q^*(k)$, i.e., there is an optimal checkpointing algorithm.

As we will see throughout this section, this a non-trivial statement. From the proof of Theorem 3.10, we gain additional insight in the behavior of good algorithms. In particular, we show that we can assume without increasing discrepancy that for all i the i-th checkpoint is set by a factor of at least $(1 + 1/k)^{\Theta(i)}$ later than the first checkpoint.

An initial set of checkpoints can be described by a vector $x = (x_1, \ldots, x_k)$, $0 \leq x_1 \leq \ldots \leq x_k$. By Lemma 3.3, we can assume that $x_k = 1$. We denote by X the set of all initial sets of checkpoints (described by vectors $x \neq 0$ as above).

We say that $A = (t, d)$ is an algorithm for an initial set $x \in X$ of checkpoints if $t_i = x_i$ for all $i \in [1..k]$. We denote by $q(x) := \inf_A \mathrm{Discr}(A)$, where A runs over all algorithms for x, the *discrepancy* of x. An initial set of checkpoints $x \in X$ is called *optimal* if $q(x) = \inf_{x \in X} q(x) = q^*(k)$.

Lemma 3.11: Optimal initial sets of checkpoints exist.

Proof. It is not hard to see that $q(\cdot)$ is continuous on X: Let $x, x' \in X$ with $|x - x'|_\infty \leq \varepsilon$ and consider an algorithm $A = (t, d)$ for x. We construct an algorithm $A' = (t', d)$ for x' by setting $t'_i = t_i$ for $i > k$. Then $|\mathrm{Discr}(A) - \mathrm{Discr}(A')| \leq 2\varepsilon$, since any interval's length is changed by at most 2ε. This implies $|q(x) - q(x')| \leq 2\varepsilon$ and, thus, shows continuity of $q(\cdot)$.

Now, since $q(\cdot)$ is continuous on X and X is compact, there exists an $x \in X$ such that $q(x) = \inf_{x \in X} q(x) = q^*(k)$. □

An easy observation is that if some checkpointing algorithm leads to a vector x of checkpoints at some time, then we may continue from there using any other algorithm for x. The discrepancy of this combined algorithm is at most the maximum of the two discrepancies. We next formalize this notion.

Definition 3.12: Let $A = (t, d)$ be a checkpointing algorithm. Let $i > k$. We call $q_{A,i} = \max_{j \in [k..i]} \overline{\ell}_{t_j}(k+1)/t_j$ the *partial discrepancy* of A observed in the time up to t_i.

Observation 3.13: Assume that when running A, at time t_i the checkpoints $x = (x_1, \ldots, x_k = t_i)$ are active. Let $A' = (t', d')$ be an algorithm for x. Then the checkpointing algorithm obtained from running A until time t_i and then continuing with algorithm A' is a checkpointing algorithm that has discrepancy at most $\max\{q_{A,i}, \mathrm{Discr}(A')\}$. If we run this combined algorithm only until some time t'_j, then the partial discrepancy observed till then is $\max\{q_{A,i}, q_{A',j}\}$.

The above observation implies that in the following we may instead of looking at an arbitrary time simply assume that the algorithm just started, that is, that the current set of checkpoints is the initial one.

The following lemma shows that we can, without loss of generality (i.e., without losing discrepancy), assume that an algorithm for the checkpointing problem does not set checkpoints too close together. While also of independent interest, among others because it shows how to keep additional costs for setting and removing checkpoints low, we will need this statement in our proof of Theorem 3.10.

Lemma 3.14: Let $A = (t, d)$ be an algorithm for the checkpointing problem with $\mathrm{Discr}(A) < k + 1$. Then there is an algorithm $A' = (t', d')$ with the same starting position such that

(i) $\mathrm{Discr}(A') \leq \mathrm{Discr}(A)$ and

(ii) $t'_{k+3} \geq t'_k \left(1 + \dfrac{\mathrm{Discr}(A)}{k + 1 - \mathrm{Discr}(A)}\right) \geq t'_k \left(1 + \dfrac{1}{k}\right)$.

Proof. Let $r = \mathrm{Discr}(A)/(k + 1 - \mathrm{Discr}(A))$ for convenience. By way of contradiction, assume that the lemma is false. Let A be a counter-example such that $i := \min\{i \in \mathbb{N} \mid t_{k+i} \geq 1 + r\}$ is minimal (the minimum is well-defined, since for any algorithm the sequence $(t_i)_i$ tends to infinity). Note that $i \geq 4$, since A is a counter-example.

Assume that there is a $j \in [1..i - 1]$ such that t_{k+j} in the further run of A is removed (and replaced by the then current time t_x) earlier than both t_{k+j-1} and t_{k+j+1}. Consider the Algorithm A' that arises from A by the following modifications. Let t_y be the checkpoint that was removed to install the checkpoint t_{k+j}. Let A' be the checkpointing algorithm that proceeds as A except that t_y is not replaced by t_{k+j}, but by t_x, and t_{k+j} is never created. The only interval which could cause this algorithm to have a worse discrepancy than A is $[t_{k+j-1}, t_{k+j+1}]$. However, this interval contributes $(k + 1)(t_{k+j+1} - t_{k+j-1})/t_{k+j+1} \leq (k + 1)r/(1 + r) = \mathrm{Discr}(A)$ to the discrepancy of A'. Hence, $\mathrm{Discr}(A') \leq \mathrm{Discr}(A)$ and A' has fewer checkpoints in the interval $[1, 1 + r]$ contradicting the minimality of A. Thus, there is no $j \in [1..i - 1]$ such that t_{k+j} is removed earlier than both $t_{k+j-1}$and t_{k+j+1} (*).

We consider now separately the two cases that t_{k+1} is removed earlier than t_{k+i-2} and vice versa. Note first that $k + 1 < k + i - 2$ by assumption that $i \geq 4$.

Assume first that t_{k+1} is removed (at some time t_x) earlier than t_{t+i-2}. Then t_k must have been removed even earlier (at some time t_y), otherwise we found a contradiction to (*). Let A' be an algorithm working identically as A, except that at time t_y the checkpoint t_{k+1} is removed (instead of t_k) and at time t_x the checkpoint t_k is removed (instead of t_{k+1}). Since the checkpoint at t_{t+i-2} is still present, the only interval affected by this exchange, namely the one with t_k as left endpoint, has length at most r. Hence as above, this contributes at most $\mathrm{Discr}(A)$ to the discrepancy of A'. The algorithm A' has the property that there is a checkpoint between t_k and t_{k+i-2} which is removed before these two points. The earliest such checkpoint, call it t_{k+j}, has the property that t_{k+j} is removed earlier than both t_{k+j-1} and t_{k+j+1}, contradicting earlier arguments.

A symmetric argument shows that also t_{k+i-2} being removed before t_{k+1} leads to a contradiction. Consequently, our initial assumption that $i \geq 4$ cannot hold, proving the claim. □

The following is a global variant of Lemma 3.14. It shows that any reasonable checkpointing algorithm does not store new checkpoints too often.

Theorem 3.15: Let $A = (t, d)$ be a checkpointing algorithm with $\mathrm{Discr}(A) < k + 1$. Then there is an algorithm $A' = (t', d')$ with the same starting position such that

(i) $\mathrm{Discr}(A') \leq \mathrm{Discr}(A)$ and

(ii) $t'_{i+3} \geq (1+1/k) \cdot t'_i$ for all $i \geq k$.

Proof. Let $j \geq k$ be the smallest index with a small jump, $t_{j+3} < (1+1/k)t_j$. Using Lemma 3.14 (on the remainder of algorithm A starting at time t_j) we can remove this small jump and get an algorithm $A' = (t', d')$ with $\mathrm{Discr}(A') \leq \mathrm{Discr}(A)$ and $t'_{i+3} \geq (1+1/k) \cdot t'_i$ for all $k \leq i \leq j$, i.e., we patched the earliest small jump. Iterating this patching procedure infinitely often yields the desired algorithm. □

Lemma 3.16: For any optimal initial set $x = (x_1, \ldots, x_k)$, there is an algorithm $A = (t, d)$ such that

(i) $q_{A,k+3} = \max_{j \in [k..k+3]} \ell_{t_j}(k+1)/t_j \leq q^*(k)$,

(ii) $t_{k+3} \geq t_k(1+1/k)$, and the set of checkpoints active at time t_{k+3} is again optimal.

Proof. By the definition of optimality, for each $n \in \mathbb{N}$ there is an algorithm $A^{(n)}$ for x that has discrepancy at most $q^*(k) + 1/n$. Let $(t^{(n)}_{k+1}, t^{(n)}_{k+2}, t^{(n)}_{k+3})$ denote the corresponding next three checkpoints. By Lemma 3.14, we may assume that $t^{(n)}_{k+3} \geq t_k(1+1/k)$ for all $n \in N$.

Note that (using the same arguments as in Lemma 3.9) any algorithm having discrepancy at most 2.5 satisfies $t_{k+i} \leq 6^i t_k$ for any $k \geq 2$. Hence, $(t^{(n)}_{k+1}, t^{(n)}_{k+2}, t^{(n)}_{k+3})_n$ is a sequence in the compact space $[t_k, 6^3 t_k]^3$. This sequence has a convergent subsequence with limit $(t_{k+1}, t_{k+2}, t_{k+3})$. Also, since there are only finitely many values possible for $(d^{(n)}_{k+1}, d^{(n)}_{k+2}, d^{(n)}_{k+3})$, this subsequence can be chosen such that this d-tuple is constant, say $(d_{k+1}, d_{k+2}, d_{k+3})$. For this subsequence, also all $k+1$ intervals existing at the three times of interest converge. Consequently, the discrepancy caused by each of them also converges to a value of at most $q^*(k)$. This defines the three steps of algorithm A, satisfying $q_{A,k+3} \leq q^*(k)$.

Similarly, we observe that the set of checkpoints $x^{(n)}$ active at time $t^{(n)}_{k+3}$ when running algorithm $A^{(n)}$ has discrepancy at most $q^*(k) + 1/n$. Consequently, the active checkpoints we get from the limit checkpoints $(t_{k+1}, t_{k+2}, t_{k+3})$ and deletions $(d_{k+1}, d_{k+2}, d_{k+3})$ are again optimal.

Finally, since all $t^{(n)}_{k+3} \geq t_k(1+1/k)$, this also holds for t_{k+3}. □

We are now in position to prove the main result of this section, Theorem 3.10. For this, we repeatedly apply Lemma 3.16: We start with an optimal set of checkpoints x. Then we run the algorithm delivered by Lemma 3.16 for three steps. This creates no partial discrepancy larger than $q^*(k)$ and we end up with another optimal set of checkpoints. From this, we continue to apply Lemma 3.16 and execute three steps of the algorithm obtained. By Observation 3.13, the partial discrepancy of the combined algorithm is again at most $q^*(k)$. Iterating infinitely, this yields an optimal algorithm, which proves Theorem 3.10.

3.8 Lower Bound

In this section, we prove a lower bound on the discrepancy of all checkpointing algorithms. For large k we get a lower bound of roughly 1.3, so we have a lower bound that is asymptotically larger than the trivial bound of 1. Moreover, it shows that algorithm BINARY from Sect. 3.5 is nearly optimal, as for large k the presented lower bound is within 6% of the discrepancy of BINARY.

Theorem 3.17: All checkpointing algorithms with k checkpoints have a discrepancy of at least

$$2 - \ln 2 - O(k^{-1}) \geq 1.306 - O(k^{-1}).$$

The remainder of this section is devoted to the proof of the above theorem.

Let $A = (t, d)$ be an arbitrary checkpointing algorithm and let $q' := \text{Discr}(A)$ be its discrepancy. For convenience, we define $q = kq'/(k+1)$ and bound q. Since $q < q'$ this suffices to show a lower bound for the discrepancy of A. For technical reasons we add a *gratis checkpoint* at time t_k that must not be removed by A. That is, even after the removal of the original checkpoint at t_k, there still is the gratis checkpoint active at t_k. Clearly, this can only improve the discrepancy. We analyze the discrepancy of A from time t_k until it deleted $k/(2q)$ of the initial checkpoints[1]. More formally, we let t' be the minimal time at which the number of active checkpoints of A contained in $[0, t_k]$ is $k - k/(2q)$. We show that it cannot happen that the checkpointing algorithm A never deletes $k/(2q)$ points from $[0, t_k]$.

Lemma 3.18: We have $t' < \infty$.

Proof. Consider a large $i > k$ and the algorithm's discrepancy at time t_i. By assumption, there are at most $k/(2q)$ active checkpoints in $(t_k, t_i]$. Hence, by comparing with an equidistant spread we can bound the discrepancy (at time t_i) by

$$\text{Discr}(A) \geq \frac{k+1}{t_i} \cdot \frac{t_i - t_k}{k/(2q)} = 2q\frac{k+1}{k}\Big(1 - \frac{t_k}{t_i}\Big) = 2\,\text{Discr}(A)\Big(1 - \frac{t_k}{t_i}\Big).$$

Letting $i \to \infty$, so that $t_i \to \infty$, we obtain the contradiction $\text{Discr}(A) \geq 2\,\text{Discr}(A)$. □

Hence, in the following we can assume that $t' < \infty$. We partition the intervals that exist at time t' into three types:

1. Intervals existing both at time t_k and t'. These intervals are contained in $[0, t_k]$.
2. Intervals that are contained in $[0, t_k]$, but did not exist at time t_k. These intervals were created by the removal of some checkpoint in $[0, t_k]$ after time t_k.
3. Intervals contained in $[t_k, t']$.

[1] To be precise we should round $\frac{k}{2q}$ to one of its nearest integers. When doing so, all calculations in the remainder of this section go through as they are; this only slightly increases the hidden constant in the error term $O(k^{-1})$.

Note that we need the gratis checkpoint at t_k in order for these definitions to make sense, as otherwise there could be an interval overlapping t_k.

Let $\mathcal{L}_i$ denote the set of intervals of type i for $i \in \{1,2,3\}$, and set $k_i := |\mathcal{L}_i|$. Let $\mathcal{L}_2 = \{I_1, \ldots, I_{k_2}\}$, where the intervals are ordered by their creation times $\tau_1 \leq \ldots \leq \tau_{k_2}$. Since each interval in $\mathcal{L}_2$ contains at least one deleted point we have

$$k_2 \leq \frac{k}{2q},$$

and we set $m := \frac{k}{2q} - k_2$. Then m counts the number of deleted checkpoints in $[0, t_k]$ that did not create an interval in $\mathcal{L}_2$, but some strict sub-interval of an interval in $\mathcal{L}_2$. We call these m removed checkpoints *free*.

We first bound the length of the intervals in $\mathcal{L}_1$ and $\mathcal{L}_2$.

Lemma 3.19: The length of any interval in $\mathcal{L}_1$ is at most qt_k/k.

Proof. As all intervals in $\mathcal{L}_1$ already are present at time t_k and the algorithm has discrepancy q', we have for any $I \in \mathcal{L}_1$

$$(k+1)|I|/t_k \leq q' = (k+1)q/k.$$

The bound follows. □

Lemma 3.20: The length of any interval $I_i \in \mathcal{L}_2$ is at most

$$|I_i| \leq \frac{t_k}{k/q - m - i}.$$

Proof. As the algorithm has discrepancy q', we know

$$|I_i| \leq q\tau_i/k. \tag{3.8.1}$$

In the following we bound τ_i, the time of creation of I_i. At time τ_i there are at most $m + i$ intervals in $\mathcal{L}_3$, since at most m free checkpoints and i checkpoints from the creation of $I_1, \ldots, I_i$ are available. Comparing with an equidistant spread of $m + i$ checkpoints in $[t_k, \tau_i]$ and the algorithm's discrepancy, the longest interval L in $[t_k, \tau_i]$ (at time τ_i) has length

$$\frac{\tau_i - t_k}{m+i} \leq |L| \leq \frac{q\tau_i}{k}.$$

Rearranging the outer inequality yields a bound on τ_i of

$$\tau_i \leq \frac{kt_k}{k - (m+i)q}.$$

Substituting this into (3.8.1) yields the desired result. □

Furthermore, we need a relation between k_1, k, m, and q.

Lemma 3.21: We have

$$k_1 = k + m - k/q + 1.$$

Proof. As the intervals in $\mathcal{L}_1$ and $\mathcal{L}_2$ partition $[0, t_k]$, there are $k_1 + k_2$ intervals left in $[0, t_k]$ at time t'. Note that each but one such interval has its left endpoint among the k active checkpoints from time t_k (the one exception having as left endpoint 0). Hence, there are $k_1 + k_2 - 1$ checkpoints left in $[0, t_k]$. Comparing with the number $k_2 + m$ of deleted checkpoints in $[0, t_k]$ until time t' and their overall number k yields

$$(k_2 + m) + (k_1 + k_2 - 1) = k.$$

Rearranging this and plugging in $k_2 = \frac{k}{2q} - m$ (which holds by definition of m) yields the desired result. □

Now we use our bounds on the length of intervals from $\mathcal{L}_1$ and $\mathcal{L}_2$ to find a bound on q. Note that the intervals in $\mathcal{L}_1$ and $\mathcal{L}_2$ partition $[0, t_k]$, so that

$$t_k = \sum_{I \in \mathcal{L}_1} |I| + \sum_{I' \in \mathcal{L}_2} |I'|.$$

Using Lemmas 3.19 and 3.20, we obtain

$$t_k \leq k_1 \frac{q t_k}{k} + \sum_{i=1}^{k_2} \frac{t_k}{k/q - m - i}.$$

Substituting k_1 using Lemma 3.21 yields

$$\begin{aligned} t_k &\leq (k + m - k/q + 1)\, q t_k / k + \sum_{i=1}^{k/(2q)-m} \frac{t_k}{k/q - m - i} \\ &= t_k \left(q - 1 + m\frac{q}{k} + O(k^{-1}) + \sum_{i=1}^{k/(2q)-m} \frac{1}{k/q - m - i} \right). \end{aligned} \tag{3.8.2}$$

Recall that $H_n = \sum_{1 \leq i \leq n} i^{-1}$ is the n-th harmonic number. Rearranging (3.8.2) yields

$$q \geq 2 - m\frac{q}{k} - O(k^{-1}) - H_{k/q-m-1} + H_{k/(2q)-1}.$$

Observe that we have $m\frac{q}{k} + H_{k/q-m-1} \leq H_{k/q-1}$, implying

$$\begin{aligned} q &\geq 2 + H_{k/(2q)-1} - H_{k/q-1} - O(k^{-1}) \\ &\geq 2 + H_{k/(2q)} - H_{k/q} - O(k^{-1}), \end{aligned}$$

since we can hide the last summands of $H_{k/(2q)}$ and $H_{k/q}$ by $O(k^{-1})$. In combination with the asymptotic behavior of $H_n = \ln n + \gamma + O(n^{-1})$, where γ is the Euler-Mascheroni constant, we obtain

$$\begin{aligned} q &\geq 2 + \ln(k/(2q)) - \ln(k/q) - O(k^{-1}) \\ &= 2 - \ln(2) - O(k^{-1}). \end{aligned}$$

This finishes the proof of Theorem 3.17.

3.9 Conclusion

In this chapter we considered the problem of maintaining a set of k checkpoints (each one storing a state of a long computation) such that the longest time interval between two checkpoints is kept small. This allows to rewind an ongoing computation to an arbitrary previous state with small recomputation time. Our performance measure was scarcely studied before; the most relevant work [2] mainly regarded a less natural measure. We improved the best guarantee from $2 - o(1)$ [2] to a constant smaller than 2 and the best lower bound from $1 + o(1)$ [2] to a constant greater than 1. In particular, for k being a power of 2 our lower bound is just 6% less than our upper bound. Moreover, we proved the existence of optimal algorithms.

There remain some open problems to investigate. For the checkpointing problem some natural greedy algorithms exist. For example, an algorithm could always remove the old checkpoint that creates the smallest new interval. It would be interesting to bound the performance of such algorithms, however, it seems difficult to keep track of the interval lengths during the execution. Empirically, greedy heuristics seem to become periodic after a sufficient number of steps, but this property also seems difficult to prove.

Additionally, other discrepancy measures might be of interest. In our setting we optimize the worst-case recomputation time. One could also try to optimize the expected recomputation time, by keeping the sum of the squares of interval lengths small. In some applications, one might even know some probability distribution on the past states to which one wants to rewind. Here, better checkpoint placements could be possible.

Chapter 4

Inapproximability of the Robust k-Median Problem and Heuristic Solutions

4.1 Introduction

In the previous two chapters we studied problems related to the reliability of the implementation of algorithms as well as the hardware they run on. In this chapter we study a variation of the k-median problem where the *input* contains uncertainty.

In the classical k-median problem, we are given a set of clients located on a metric space with distance function $d : V \times V \to \mathbb{R}$. The goal is to open a set of facilities $F \subseteq V$, $|F| = k$, so as to minimize the sum of the connection costs of the clients in V, i.e., their distances from their nearest facilities in F. This is a central problem in approximation algorithms, and has received a large amount of attention in the past two decades [19, 7, 20, 45, 44].

At SODA 2008 Anthony et al. [4] introduced a generalization of the k-median problem. In their setting, the set of clients that are to be connected to some facility is not known in advance, and the goal is to perform well in spite of this uncertainty about the future. They formulated the problem as follows.

Definition 4.1 (Robust k-Median)**:** An instance of this problem is a triple $(V, \mathcal{S}, d)$. This defines a set of *locations* V, a collection of m sets of *clients* $\mathcal{S} = \{S_1, \ldots, S_m\}$, where $S_i \subseteq V$ for all $i \in \{1, \ldots, m\}$, and a metric distance function $d : V \times V \to \mathbb{R}$. We have to open a set of k facilities $F \subseteq V$, $|F| = k$, and the goal is to minimize the cost of the most expensive set of clients, i.e. minimize $\max_{i=1}^{m} \sum_{v \in S_i} d(v, F)$. Here, $d(v, F)$ denotes the minimum distance of the client v from any location in F, i.e. $d(v, F) = \min_{u \in F} d(u, v)$.

Robust k-Median is a natural generalization of the classical k-median problem (for $m = 1$). The different sets S_i can be interpreted as guesses as to which clients will want to connect to the facilities in the future. As the set that is actually realized is unknown, the min-max optimization ensures that no set of clients leads to especially high costs. Additionally, we can think of it as capturing a notion of *fairness*. To see this, interpret each set S_i as a *group* of clients who pay $\sum_{v \in S_i} d(v, F)$ for connecting to a facility. The objective ensures that no single group pays too much, while minimizing the cost. Anthony et al. [4] gave an $O(\log m)$-approximation algorithm for this problem, and a lower bound of $(2 - \epsilon)$ by a reduction from Vertex Cover. The lower bound was improved to $\log^{\alpha} n$ for small constant $\alpha > 0$ in [8]. Note that their lower bound does not hold in the line metric.

Our Results. We prove nearly tight hardness of approximation for Robust k-Median. We show that, unless $\mathsf{NP} \subseteq \cap_{\delta > 0} \mathsf{DTIME}(2^{n^{\delta}})$, it admits no poly-time $o(\log m / \log \log m)$-approximation, *even on uniform and line metrics*. Moreover, we examine several natural heuristics for this problem.

Our first hardness result is tight up to a constant factor, as a simple rounding scheme gives a matching upper bound on uniform metrics (Sect. 4.3.1). Our second result shows that Robust k-Median is a rare problem with super-constant hardness of approximation even on line metrics. This surprising result puts Robust k-Median in sharp contrast to most other geometric optimization problems which admit polynomial time approximation schemes, e.g. [5, 42].

Despite our hardness results, the heuristics we tried perform well experimentally. This is especially true if the clients are distributed uniformly in the plane and

assigned to groups uniformly at random. This suggests that some restrictions on the client distribution might make the problem tractable.

The results of this chapter are joint work with Sayan Bhattacharya, Parinya Chalermsook and Kurt Mehlhorn. The publication is available at [11].

Our Techniques. First, we note that Robust k-Median on uniform metrics is equivalent to the following variant of the set cover problem: Given a set U of ground elements, a collection of sets $\mathcal{X} = \{X \subseteq U\}$, and an integer $t \leq |\mathcal{X}|$, our goal is to select t sets from $\mathcal{X}$ in order to minimize the number of times an element from U is hit (Lemma 4.7). We call this problem Minimum Congestion Set Packing (MCSP). This characterization allows us to focus on proving the hardness of MCSP, and to employ the tools developed for the set cover problem.

We now revisit the reduction used by Feige [28], building on results of Lund and Yannakakis [48], to prove the hardness of the set cover problem and discuss how our approach differs. Intuitively, they compose a Label Cover instance with a set system that has some desirable properties. Informally speaking, in the Label Cover problem, we are given a graph where each vertex v can be assigned a label from a set L, and each edge e is equipped with a constraint $\Pi_e \subseteq L \times L$ specifying the accepting pairs of labels for e. Our goal is to find a labeling of vertices that maximizes the number of accepting edges. This problem is known to be hard to approximate to within a factor of $2^{\log^{1-\epsilon}|E|}$ [6, 62], where $|E|$ is the number of edges. Thus, if we manage to reduce Label Cover to MCSP, we would hopefully obtain a large hardness of approximation factor for MCSP as well.

From a given Label Cover instance, [48] creates an instance of Set Cover by having sets of the form $S(v, \ell)$ for each vertex v and each label $\ell \in L$. Intuitively the set $S(v, \ell)$ means choosing label ℓ for vertex v in the Label Cover instance. Now, if we assume that the solution is well behaved, in the sense that for each vertex v, only one set of the form $S(v, \ell)$ is chosen in the solution, we would be immediately done (because each set indeed corresponds to a label). However, solutions need not have this form, e.g. choosing sets $S(v, \ell)$ and $S(v, \ell')$ translates to having two labels ℓ, ℓ' for the Label Cover instance. To prevent an ill-behaved solution, *partition systems* were introduced and used in both [48] and [28]. Feige considers the hypergraph version of Label Cover to obtain a sharper hardness result of $\ln n - O(\ln \ln n)$ instead of $\frac{1}{4} \ln n$ in [48]; here n denotes the size of the universe.

Now we highlight how our reduction is different. The high level idea stays the same, i.e. we have sets of the form $S(v, \ell)$ that represent assigning label ℓ to vertex v. However, we need a different partition system and a totally different analysis. Moreover, while a reduction from standard Label Cover gives nearly tight $O(\log n)$ hardness for Set Cover, it can (at best) only give a $2 - \epsilon$ hardness for MCSP. For our results, we do need a reduction from Hypergraph Label Cover. This suggests another natural distinction between MCSP and Set Cover.

Finally, to obtain the hardness result for the line metric, we embed the instance created from the MCSP reduction onto the line while preserving values of optimal solutions. This way we get the same hardness gap for line metrics.

In the experimental part of this work we study how different heuristics perform. We wanted to see how the structure of the input instance affects the performance of

these heuristics. This might give insights into which classes of instances are hard, and which might be tractable.

We compare instances in which the groups of clients are distributed uniformly in the plane with instances where the groups form actual groups, that is, the members of a group are spatially clustered together.

As exact solutions for the problem are difficult to find, we use the natural Linear Programming relaxation of the problem as a performance baseline.

4.2 Preliminaries

We will show that Robust k-Median is $\Omega(\log m / \log\log m)$ hard to approximate, even for the special cases of *uniform metrics* (Sect. 4.3) and *line metrics* (Sect. 4.4). Recall that d is a uniform metric iff we have $d(u,v) \in \{0,1\}$ for all locations $u, v \in V$. Further, d is a line metric iff the locations in V can be embedded into a line in such a way that $d(u,v)$ equals the euclidean distance between u and v, for all $u, v \in V$. Throughout this chapter, we will denote any set of the form $\{1, 2, \ldots, i\}$ by $[i]$. Our hardness results will rely on a reduction from the *r-Hypergraph Label Cover* (HGLC) problem, which is defined as follows.

Definition 4.2 (r-Hypergraph Label Cover (HGLC))**:** An instance of this problem is a triple (G, π, r), where $G = (\mathcal{V}, \mathcal{E})$ is a r-partite hypergraph with vertex set $\mathcal{V} = \bigcup_{j=1}^{r} \mathcal{V}_j$ and edge set $\mathcal{E}$. Each edge $h \in \mathcal{E}$ contains one vertex from each part of $\mathcal{V}$, i.e. $|h \cap \mathcal{V}_j| = 1$ for all $j \in [r]$. Every set $\mathcal{V}_j$ has an associated set of *labels* L_j. Further, for all $h \in \mathcal{E}$ and $j \in [r]$, there is a mapping $\pi_h^j : L_j \to C$ that projects the labels from L_j to a common set of *colors* C.

The problem is to assign to every vertex $v \in \mathcal{V}_j$ some label $\sigma(v) \in L_j$. We say that an edge $h = (v_1, \ldots, v_r)$, where $v_j \in \mathcal{V}_j$ for all $j \in [r]$, is *strongly satisfied* under σ iff the labels of all its vertices are mapped to the same element in C, i.e. $\pi_h^j(\sigma(v_j)) = \pi_h^{j'}(\sigma(v_{j'}))$ for all $j, j' \in [r]$. In contrast, we say that the edge is *weakly satisfied* iff there exists some pair of vertices in h whose labels are mapped to the same element in C, i.e. $\pi_h^j(\sigma(v_j)) = \pi_h^{j'}(\sigma(v_{j'}))$ for some $j, j' \in [r]$, $j \neq j'$.

For ease of exposition, we will often abuse the notation and denote by $j(v)$ the part of $\mathcal{V}$ to which a vertex v belongs, i.e. if $v \in \mathcal{V}_j$ for some $j \in [r]$, then we set $j(v) \leftarrow j$. The next theorem will be crucial in deriving our hardness result. The proof of this theorem follows from Feige's r-Prover system [28] (Appendix B).

Theorem 4.3: Let $r \in \mathbb{N}$ be a parameter. There is a polynomial time reduction from n-variable 3-SAT to r-HGLC with the following properties:

- (Yes-Instance) If the formula is satisfiable, there is a labeling that strongly satisfies every edge in G.
- (No-Instance) If the formula is not satisfiable, every labeling weakly satisfies at most a $2^{-\gamma r}$ fraction of the edges in G, for some universal constant γ.
- The number of vertices in the graph is $|\mathcal{V}| = n^{O(r)}$ and the number of edges is $|\mathcal{E}| = n^{O(r)}$. The sizes of the label sets are $|L_j| = 2^{O(r)}$ for all $j \in [r]$, and

$|C| = 2^{O(r)}$. Further, we have $|\mathcal{V}_j| = |\mathcal{V}_{j'}|$ for all $j, j' \in [r]$, and each vertex $v \in \mathcal{V}$ has the same degree $r|\mathcal{E}|/|\mathcal{V}|$.

We use a *partition system* that is motivated by the hardness proof of the Set Cover problem [28] but uses a different construction.

Definition 4.4 (Partition System)**:** Let $r \in \mathbb{N}$ and let C be any finite set. An (r, C)-partition system is a pair $(Z, \{p_c\}_{c \in C})$, where Z is an arbitrary (ground) set, such that the following properties hold.

- (Partition) For all $c \in C$, $p_c = (A_c^1, \ldots, A_c^r)$ is a partition of Z, that is $\bigcup_{j=1}^r A_c^j = Z$, and $A_c^{j'} \cap A_c^j = \emptyset$ for all $j, j' \in [r], j \neq j'$.
- (r-intersecting) For any r *distinct* indices $c_1, \ldots, c_r \in C$ and *not-necessarily distinct* indices $j_1, \ldots, j_r \in [r]$, we have that $\bigcap_{i=1}^r A_{c_i}^{j_i} \neq \emptyset$. In particular, $A_c^j \neq \emptyset$ for all c and j.

In order to achieve a good lower bound on the approximation factor, we need partition systems with *small* ground sets. The most obvious way to build a partition system is to form an r-hypercube: Let $Z = [r]^{|C|}$, and for each $c \in C$ and $j \in [r]$, let A_c^j be the set of all elements in Z whose c-th component is j. It can easily be verified that this is an (r, C)-partition system with $|Z| = r^{|C|}$. With this construction, however, we would only get a hardness of $\Omega(\log \log m)$ for our problem. The following lemma shows that it is possible to construct an (r, C)-partition system probabilistically with $|Z| = r^{O(r)} \log |C|$.

Lemma 4.5: There is an (r, C)-partition system with $|Z| = r^{O(r)} \log |C|$ elements. Further, such a partition system can be constructed efficiently with high probability.

Proof. Let Z be any set of $r^{O(r)} \log |C|$ elements. We build a partition system $(Z, \{p_c\}_{c \in C})$ as described in Algorithm 2. By construction each p_c is a partition of Z, i.e. the first property stated in Definition 4.4 is satisfied. We bound the probability that the second property is violated.

Algorithm 2: A randomized construction of an (r, C)-partition system.

Input: A ground set Z, a parameters $r \in \mathbb{N}$, and a set C.
foreach $c \in C$ **do**
  `/* Construct the partition` $p_c = (A_c^1, \ldots, A_c^r)$ `*/`
  Initialize A_c^j to the empty set for all $j \in [r]$
  foreach *ground element* $e \in Z$ **do**
    Pick a $j \in [r]$ independently and uniformly at random and add e to A_c^j

Fix any choice of r *distinct* indices $c_1, \ldots, c_r \in C$ and *not necessarily distinct* indices $j_1, \ldots, j_r \in [r]$. We say that a *bad event* occurs when the intersection of the corresponding sets is empty, i.e. $\bigcap_{i=1}^r A_{c_i}^{j_i} = \emptyset$. To upper bound the probability of a bad

event, we focus on events of the form $E_{e,i}$ – this occurs when an element $e \in Z$ is included in a set $A^{j_i}_{c_i}$. Since the indices $c_1 \dots c_r$ are distinct, it follows that the events $\{E_{e,i}\}$ are mutually independent. Furthermore, note that we have $\Pr[E_{e,i}] = 1/r$ for all $e \in Z, i \in [r]$. Hence, the probability that an element $e \in Z$ does not belong to the intersection $\bigcap_{i=1}^{r} A^{j_i}_{c_i}$ is given by $1 - \Pr[\bigcap_{i=1}^{r} E_{e,i}] = 1 - 1/r^r$. Accordingly, the probability that no element $e \in Z$ belongs to the intersection, which defines the bad event, is equal to $(1 - 1/r^r)^{|Z|}$.

Now, the number of choices for r distinct indices $c_1, \dots, c_r$ and r not-necessarily distinct indices $j_1, \dots, j_r$ is equal to $\binom{|C|}{r} \cdot r^r$. Hence, by a union-bound over all bad events, the second property stated in Definition 4.4 is violated with probability at most $\binom{|C|}{r} \cdot r^r \cdot (1 - r^r)^{|Z|} \leq (|C|r)^r \cdot \exp(-|Z|/r^r)$. If we set $|Z| = d \cdot r^{d \cdot r} \log |C|$ with sufficiently large constant d, the property is satisfied with high probability. □

4.3 Hardness of Robust k-Median on Uniform Metrics

First, we define *Minimum Congestion Set Packing* (MCSP), and then show a reduction from MCSP to Robust k-Median on uniform metrics. In Sect. 4.3.2, we will then show that MCSP is hard to approximate by reducing HGLC to MCSP.

Definition 4.6 (Minimum Congestion Set Packing (MCSP))**:** An instance of this problem is a triple $(U, \mathcal{X}, t)$, where U is a universe of m elements, i.e. $|U| = m$, $\mathcal{X}$ is a collection of sets $\mathcal{X} = \{X \subseteq U\}$ such that $\bigcup_{X \in \mathcal{X}} X = U$, and $t \in \mathbb{N}$ and $t \leq |\mathcal{X}|$. The objective is to find a collection $\mathcal{X}' \subseteq \mathcal{X}$ of size t that minimizes $\textsc{Cong}(\mathcal{X}') = \max_{e \in U} \textsc{Cong}(e, \mathcal{X}')$. Here, $\textsc{Cong}(\mathcal{X}')$ refers to the *congestion* of the solution $\mathcal{X}'$, and $\textsc{Cong}(e, \mathcal{X}') = |\{X \in \mathcal{X}' : e \in X\}|$ is the congestion of the element $e \in U$ under the solution $\mathcal{X}'$.

Lemma 4.7: Given any MCSP instance $(U, \mathcal{X}, t)$, we can construct a Robust k-Median instance $(V, \mathcal{S}, d)$ with the same objective value in $poly(|U|, |\mathcal{X}|)$ time, such that $|U| = |\mathcal{S}|$, $|\mathcal{X}| = |V|$, d is a uniform metric, and $k = |V| - t$.

Proof. We construct the Robust k-Median instance $(V, \mathcal{S}, d)$ as follows. For every $e \in U$ we create a set of clients $S(e)$, and for each $X \in \mathcal{X}$ we create a location $v(X)$. Thus, we get $V = \{v(X) : X \in \mathcal{X}\}$, and $\mathcal{S} = \{S(e) : e \in U\}$. We place the clients in $S(e)$ at the locations of the sets that contain e, i.e. $S(e) = \{v(X) : X \in \mathcal{X}, e \in X\}$ for all $e \in U$. The distance is defined as $d(u, v) = 1$ for all $u, v \in V, u \neq v$, and $d(v, v) = 0$. Finally, we set $k \leftarrow |V| - t$.

Now, it is easy to verify that the Robust k-Median instance $(V, \mathcal{S}, d)$ has a solution with objective ρ iff the corresponding MCSP instance $(U, \mathcal{X}, t)$ has a solution with objective ρ. The intuition is that a location $v(X) \in V$ is *not* included in the solution F to the Robust k-Median instance iff the corresponding set X is included in the solution $\mathcal{X}'$ to the MCSP instance. Indeed, let F be any subset of $\mathcal{X}$ of size k (= the set of open facilities) and let $\mathcal{X}' = \mathcal{X} - F$. Further, let $[X \in \mathcal{X}']$ be an indicator variable that is set to 1 iff $X \in \mathcal{X}'$. Then

$$\begin{aligned}\textsc{Cong}(\mathcal{X}') &= \max_{e \in U} \textsc{Cong}(e, \mathcal{X}') = \max_{e \in U} \sum_{X; e \in X} [X \in \mathcal{X}'] \\ &= \max_{e \in U} \sum_{X; e \in X} \min_{Y \in F} d(X, Y) = \max_{S(e) \in \mathcal{S}} \sum_{v(X) \in S(e)} d(v(X), F).\end{aligned}$$

□

We devote the rest of Sect. 4.3 to MCSP and show that it is $\Omega(\log |U| / \log\log |U|)$ hard to approximate. This, in turn, will imply a $\Omega(\log |\mathcal{S}| / \log\log |\mathcal{S}|)$ hardness of approximation for Robust k-Median on uniform metrics. We will prove the hardness result via a reduction from HGLC.

4.3.1 Integrality Gap

Before proceeding to the hardness result, we show that a natural LP relaxation for the MCSP problem [4] has an integrality gap of $\Omega(\log m / \log\log m)$, where $m = |U|$ is the size of the universe of elements. In the LP, we have a variable $y(X)$ indicating that the set $X \in \mathcal{X}$ is chosen, and a variable z which represents the maximum congestion among the elements. The formulation is as follows:

$$\begin{aligned} \min \quad & z \\ \text{s.t.} \quad & \sum_{X \in \mathcal{X} : e \in X} y(X) \leq z \text{ for all } e \in U \\ & \sum_{X \in \mathcal{X}} y(X) = t \end{aligned}$$

The Instance: Now, we construct a bad integrality gap instance $(U, \mathcal{X}, t)$. Let d be the intended integrality gap, let $\eta = d^2$, and let $U = \{I : I \subseteq [\eta], |I| = d\}$ be all subsets of $[\eta]$ of size d. The collection $\mathcal{X}$ consists of η sets $X_1, \ldots, X_\eta$, where $X_i = \{I : I \in U \text{ and } i \in I\}$. Note that the universe U consists of $|U| = m = \binom{\eta}{d}$ elements, and each element I is contained in exactly d sets, namely $I \in X_i$ if and only if $i \in I$. Finally, we set $t \leftarrow \eta / d$.

Analysis: The fractional solution simply assigns a value of $1/d$ to each variable $y(X_i)$; this ensures that the total (fractional) number of sets selected is $\eta / d = t$. Furthermore, each element is contained (fractionally) in exactly one set, so the fractional solution has cost one. Any integral solution must choose $\eta / d = d$ sets, say $X_{i_1} \ldots X_{i_d}$. Then $I = \{i_1, \ldots, i_d\} \in X_{i_\lambda}$ for all $\lambda \in [d]$ and hence the congestion of I is d, and this also means that any integral solution has cost at least d. Finally, since $|U| = m \leq \eta^d \leq (d^2)^d$, we have $d = \Omega(\log m / \log\log m)$.

Tightness of the result: The bound on the hardness and integrality gap is tight for the uniform metric case, as there is a simple $O(\log m / \log\log m)$-approximation algorithm. Pick each set X with probability equal to $\min(1, 2y(X))$. The expected congestion is $2z$ for each element. By Chernoff's bound [35], an element is covered by no more than $z \cdot O(\log m / \log\log m)$ sets with high probability. A similar algorithm gives the same approximation guarantee for Robust k-Median on uniform metrics.

4.3.2 Reduction from r-Hypergraph Label Cover to Minimum Congestion Set Packing

The input is an instance (G, π, r) of r-HGLC (Definition 4.2). From this we construct the following instance $(U, \mathcal{X}, t)$ of MCSP (Definition 4.6).

- We define the universe U as a union of disjoint sets. For each edge $h \in \mathcal{E}$ in the hypergraph we have a set U_h. All these sets have the same size m^* and are pairwise disjoint, i.e. $U_h \cap U_{h'} = \emptyset$ for all $h, h' \in \mathcal{E}$, $h' \neq h$. The universe U is then the union of these sets $U = \bigcup_{h \in \mathcal{E}} U_h$. Since the U_h are mutually disjoint, we have $m = |U| = |\mathcal{E}| \cdot m^*$. Recall that C is the target set of π. Each set U_h is the ground set of an (r, C)-partition system (Definition 4.4) as given by Lemma 4.5. In particular we have $m^* = r^{O(r)} \log |C|$. We denote the r-partitions associated with U_h by $\{p_c(h)\}_{c \in C}$, where $p_c(h) = \left(A_c^1(h), \ldots, A_c^r(h)\right)$.
- We construct the collection of sets $\mathcal{X}$ as follows. For each $j \in [r]$, $v \in \mathcal{V}_j$ and $\ell \in L_j$, $\mathcal{X}$ contains the set $X(v, \ell)$, where $X(v, \ell) = \bigcup_{h : v \in h} A^j_{\pi^j_h(\ell)}(h)$. That is, $X(v, \ell) \cap U_h$ is empty if $v \notin h$ and is equal to $A^j_{\pi^j_h(\ell)}(h)$ if $v \in h$. Intuitively, choosing the set $X(v, \ell)$ corresponds to assigning label ℓ to the vertex v.
- We define $t \leftarrow |\mathcal{V}|$. Intuitively, this means each vertex in $\mathcal{V}$ gets one label.

We assume for the sequel that the r-HGLC instance is chosen according to Theorem 4.3. We assume that the parameter r satisfies $r^7 2^{-\gamma r} < 1$. In the proof of the main theorem, we will fix r to a specific value.

4.3.3 Analysis

We show that the reduction from HGLC to MCSP satisfies two properties. In Lemma 4.8, we show that for Yes-Instances (see Theorem 4.3) the corresponding MCSP instance admits a solution with congestion one. For No-Instances, Lemma 4.13 shows that any solution to the corresponding MCSP instance has congestion at least r.

Lemma 4.8 (Yes-Instance)**:** If the HGLC instance (G, π, r) admits a labeling that strongly satisfies every edge, then the MCSP instance $(U, \mathcal{X}, t)$ as in Sect. 4.3.2 admits a solution where the congestion of every element in U is exactly one.

Proof. Suppose that there is a labeling σ that strongly satisfies every edge $h \in \mathcal{E}$. We will show how to pick $t = |\mathcal{V}|$ sets from $\mathcal{X}$ such that each element in U is contained in exactly one set. This implies that the maximum congestion is one. For each $j \in [r]$ and each vertex $v \in \mathcal{V}_j$, we choose the set $X(v, \sigma(v))$. Thus, the total number of sets chosen is exactly $|\mathcal{V}|$.

To see that the congestion is one, we concentrate on the elements in U_h, where $h = (v_1, \ldots, v_r)$, $v_j \in \mathcal{V}_j$ for all $j \in [r]$, is one of the edges in $\mathcal{E}$. The picked sets that intersect U_h are $X(v_j, \sigma(v_j))$, where $j \in [r]$. Since h is strongly satisfied, π_h maps all vertex labels in h to a common $c \in C$, i.e. $\pi^j_h(\sigma(v_j)) = c$ for all $j \in [r]$. Thus $U_h \cap X(v_j, \sigma(v_j)) = A^j_c(h)$. By definition (Definition 4.4), the sets $A^1_c(h) \ldots A^r_c(h)$ partition the elements in U_h. This completes the proof. □

Proofing something similar for the no-case (Lemma 4.13) requires some preparation. Towards this end, we fix a collection $\mathcal{X}' \subseteq \mathcal{X}$ of size t and show that some element in U has congestion at least r under $\mathcal{X}'$. The intuition being that many edges in

$G = (\mathcal{V}, \mathcal{E})$ are not even weakly satisfied, and the elements in U corresponding to those edges incur large congestion. Recall that for a $v \in \mathcal{V}$, we define $j(v) \in \mathbb{N}$ to be such that $v \in \mathcal{V}_{j(v)}$.

Claim 4.9: For $v \in \mathcal{V}$, let $\mathcal{L}_v = \left\{\ell \in L_{j(v)} : X(v,\ell) \in \mathcal{X}'\right\}$. For $h \in \mathcal{E}$, let $\Lambda_h = \{X(v,\ell) \in \mathcal{X}' : v \in h\}$ and $\lambda(h) = |\Lambda_h|$. If the solution $\mathcal{X}'$ has congestion less than r then $|\mathcal{L}_v| < r^2$ and $|\Lambda_h| < r^3$.

Proof. Since $\Lambda_h = \bigcup_{v \in h} \mathcal{L}_v$, it suffices to prove $|\mathcal{L}_v| < r^2$ for all v. Assume otherwise, i.e., $|\mathcal{L}_v| \geq r^2$ for some $v \in \mathcal{V}_j$, $j \in [r]$. Let h be any hyper-edge with $v \in h$. Consider the images of the labels in $\mathcal{L}_v$ under π_h^j. Either there are at least r distinct images or at least r elements in L_v are mapped to the same $c \in C$.

In the former case, we have r pairwise distinct labels ℓ_1 to ℓ_r in $\mathcal{L}_v$ and r pairwise distinct labels c_1 to c_r in C such that $\pi_h^j(\ell_i) = c_i$ for $i \in [r]$. The set $X(v,\ell_i)$ contains $A_{c_i}^j(h)$ and $\bigcap_{i\in[r]} A_{c_i}^j(h) \neq \emptyset$ by property (2) of partition systems (Definition 4.4). Thus some element has congestion at least r.

In the latter case, we have r pairwise distinct labels ℓ_1 to ℓ_r in $\mathcal{L}_v$ and a label c in C such that $\pi_h^j(\ell_i) = c$ for $i \in [r]$. The set $X(v,\ell_i)$ contains $A_c^j(h)$ and hence every element in this non-empty set (property (2) of partition systems) has congestion at least r. □

Definition 4.10 (Colliding Edge)**:** We say that an edge $h \in \mathcal{E}$ is *colliding* iff there are sets $X(v,\ell), X(v',\ell') \in \mathcal{X}'$ with $v, v' \in h$, $v \neq v'$, and $\pi_h^{j(v)}(\ell) = \pi_h^{j(v')}(\ell')$.

Claim 4.11: Suppose that the solution $\mathcal{X}'$ has congestion less than r, and more than a $r^4 2^{-\gamma r}$ fraction of the edges in $\mathcal{E}$ are colliding. Then there is a labeling σ for G that weakly satisfies at least a $2^{-\gamma r}$ fraction of the edges in $\mathcal{E}$.

Proof. For each $v \in \mathcal{V}$, we define $\mathcal{L}_v = \{\ell \in L_{j(v)} : X(v,\ell) \in \mathcal{X}'\}$. Then $|\mathcal{L}_v| < r^2$ by Claim 4.9. We construct a labeling function σ using Algorithm 3.

Now we bound the expected fraction of weakly satisfied edges under σ from below. Take any colliding edge $h \in \mathcal{E}$. Then there are vertices $v \in \mathcal{V}_j$, $v' \in \mathcal{V}_{j'}$ with $j \neq j'$, and colors $\ell \in \mathcal{L}_v$, $\ell' \in \mathcal{L}_{v'}$ such that $v, v' \in h$ and $\pi_h^j(\ell) = \pi_h^{j'}(\ell')$. By Claim 4.9, $|\mathcal{L}_v|$ and $|\mathcal{L}_{v'}|$ are both at most r^2. Since the colors $\sigma(v)$ and $\sigma(v')$ are chosen uniformly and independently at random from their respective palettes $\mathcal{L}_v$ and $\mathcal{L}_{v'}$, we have $\Pr[\sigma(v) = \ell \text{ and } \sigma(v') = \ell'] \geq 1/r^4$. In other words, every colliding

Algorithm 3: An algorithm for constructing a labeling function.

```
foreach vertex v ∈ V do
    if L_v ≠ ∅ then
        Pick a color σ(v) uniformly and independently at random from
        L_v
    else
        Pick an arbitrary color σ(v) from L_{j(v)}
```

edge is weakly satisfied with probability at least $1/r^4$. Since more than a $r^4 2^{-\gamma r}$ fraction of the edges in $\mathcal{E}$ are colliding, from linearity of expectation we infer that the expected fraction of edges weakly satisfied by σ is at least $2^{-\gamma r}$. □

Claim 4.12: Let $\Lambda_h = \{X(v,\ell) \in \mathcal{X}' : v \in h\}$ and $\lambda(h) = |\Lambda_h|$. Then $\sum_{h\in\mathcal{E}} \lambda(h) = r|\mathcal{E}|$.

Proof. This is a simple counting argument. Consider a bipartite graph H with vertex set $A \cup B$, where each vertex in A represents a set $X(v,\ell)$, and each vertex in B represents an edge $h \in \mathcal{E}$. There is an edge between two vertices iff the set $X(v,\ell)$ contains some element in U_h. The quantity $\sum_{h\in\mathcal{E}} \lambda(h)$ counts the number of edges in H where one endpoint is included in the solution $\mathcal{X}'$. Since $\mathcal{X}'$ picks $t = |\mathcal{V}|$ sets and each set has degree $r|\mathcal{E}|/|\mathcal{V}|$ in the H (Theorem 4.3), the total number of edges that are chosen is exactly $|\mathcal{V}| \times (r|\mathcal{E}|/|\mathcal{V}|) = r|\mathcal{E}|$. □

Let $\mathcal{E}' \subseteq \mathcal{E}$ denote the set of colliding edges, and define $\mathcal{E}'' = \mathcal{E} - \mathcal{E}'$. Suppose that we are dealing with a No-Instance (Theorem 4.3), i.e. the solution $\mathcal{X}'$ has congestion less than r and every labeling weakly satisfies at most a $2^{-\gamma r}$ fraction of the edges in $\mathcal{E}$. Then $\lambda(h) \le r^3$ for all $h \in \mathcal{E}$ by Claim 4.9, and no more than $r^4 2^{-\gamma r}|\mathcal{E}|$ edges are colliding, i.e. $|\mathcal{E}'| \le r^4 2^{-\gamma r}|\mathcal{E}|$, by Claim 4.11. Using these facts we conclude that $\sum_{h\in\mathcal{E}'} \lambda(h) \le r^7 2^{-\gamma r}|\mathcal{E}| < |\mathcal{E}|$, as by assumption $r^7 2^{-\gamma r} < 1$. Now, applying Claim 4.12, we get $\sum_{h\in\mathcal{E}''} \lambda(h) = r|\mathcal{E}| - \sum_{h\in\mathcal{E}'} \lambda(h) > (r-1)|\mathcal{E}|$. In particular, there is an edge $h \in \mathcal{E}''$ with $\lambda(h) \ge r$.

Recall that $\Lambda_h = \{X(v,\ell) \in \mathcal{X}' : v \in h\}$ are the sets in $\mathcal{X}'$ that intersect U_h and note that $|\Lambda_h| = \lambda(h) \ge r$. Let $\mathcal{X}^* \subseteq \Lambda_h$ be a *maximal* collection of sets with the following property: For every two distinct sets $X(v,\ell), X(v',\ell') \in \mathcal{X}^*$ we have $\pi_h^{j(v)}(\ell) \neq \pi_h^{j(v')}(\ell')$. Hence, from the definition of a partition system (Definition 4.4), it follows that the intersection of the sets in $\mathcal{X}^*$ and the set U_h is non-empty.

Now, consider any set $X(v,\ell) \in \Lambda_h - \mathcal{X}^*$. Since the collection $\mathcal{X}^*$ is maximal, there must be at least one set $X(v',\ell')$ in $\mathcal{X}^*$ with $\pi_h^{j(v)}(\ell) = \pi_h^{j(v')}(\ell')$. Since h is not colliding, we must have $j(v) = j(v')$. Consequently we get $X(v,\ell) \cap U_h = X(v',\ell') \cap U_h$. In other words, for every set $X \in \Lambda_h - \mathcal{X}^*$, there is some set $X' \in \mathcal{X}^*$ where $X \cap U_h = X' \cap U_h$. Thus, $U_h \cap (\bigcap_{X\in\Lambda_h} X) = U_h \cap (\bigcap_{X\in\mathcal{X}^*} X) \neq \emptyset$. Every element in the intersection of the sets in Λ_h and U_h will have congestion $|\Lambda_h| \ge r$. This leads to the following lemma.

Lemma 4.13 (No-Instance)**:** If every labeling weakly satisfies at most a $2^{-\gamma r}$ fraction of the edges in the hypergragph Label Cover instance (G, π, r), for some universal constant γ and that $r^7 2^{-\gamma r} < 1$ then the congestion incurred by every solution to the MCSP instance $(U, \mathcal{X}, t)$ constructed in Sect. 4.3.2 is at least r.

We are now ready to prove the main theorem of this section.

Theorem 4.14: Robust k-Median $(V, \mathcal{S}, d)$ is $\Omega(\log m / \log\log m)$ hard to approximate on uniform metrics, where $m = |\mathcal{S}|$, unless $\mathsf{NP} \subseteq \bigcap_{\delta>0} \mathsf{DTIME}(2^{n^\delta})$.

Proof. Assume that there is a polynomial time algorithm for Robust k-Median that guarantees an approximation ratio in $o(\log|\mathcal{S}|/\log\log|\mathcal{S}|)$. Then, by Lemma 4.7,

there is an approximation algorithm for the Minimum Congestion Set Packing problem with approximation guarantee $o(\log |U| / \log\log |U|)$.

Let $\delta > 0$ be arbitrary and set $r = \lfloor n^{\delta} \rfloor$, where n is the number of variables in the 3-SAT instance (Theorem 4.3). Then $r^7 2^{-\gamma r} < 1$ for all sufficiently large n. We first bound the size of the MCSP instance $(U, \mathcal{X}, t)$ constructed in Sect. 4.3.2. By Lemma 4.5, the size of an (r, C)-partition system is $|Z| = r^{O(r)} \log |C|$. By Theorem 4.3, we have $|C| = 2^{O(r)}$. So each set U_h has cardinality at most $r^{O(r)} \cdot r = r^{O(r)}$. Also recall that the number of sets in the MCSP instance is $|\mathcal{X}| = \sum_{j \in [r]} |\mathcal{V}_j| \cdot |L_j| = n^{O(r)}$, and that the number of elements is $|U| = m = |\mathcal{E}| \cdot r^{O(r)} \leq (nr)^{O(r)} = n^{O(r)} = n^{O(n^{\delta})} = 2^{O(r \log r)}$. Thus $r \geq \Omega(\log m / \log\log m)$.

The gap in the optimal congestion between the Yes-Instance and the No-Instance is at least r (Theorem 4.3 and Lemmas 4.8, 4.13). More precisely, for Yes-instances the congestion is at most one and for No-instances it is at least r. Since the approximation ratio of the alleged algorithm is $o(\log m / \log\log m)$, it is better than r for all sufficiently large n and hence it can be used to decide SAT.

The running time is polynomial in the size of the MCSP instance, i.e., is $\text{poly}(n^{O(n^{\delta})}) = n^{O(n^{\delta})} = 2^{O(n^{2\delta})}$. Since $\delta > 0$ is arbitrary, the theorem follows. □

4.4 Hardness of Robust k-Median on Line Metrics

We modify the reduction from r-HGLC to Minimum Congestion Set Packing (MCSP) to give a $\Omega(\log m / \log\log m)$ hardness of approximation for Robust k-Median on line metrics as well, where $m = |\mathcal{S}|$ is the number of client-sets. For this section, it is convenient to assume that the label-sets are the initial segments of the natural numbers, i.e., $L_j = \{1, \ldots, |L_j|\}$ and $C = \{1, \ldots, |C|\}$.

Given a HGLC instance (G, π, r), we first construct a MCSP instance $(U, \mathcal{X}, t)$ in accordance with the procedure outlined in Sect. 4.3.2. Next, from this MCSP instance, we construct a Robust k-Median instance $(V, \mathcal{S}, d)$ as described below.

- We create a location in V for every set $X(v, \ell) \in \mathcal{X}$. To simplify the notation, the symbol $X(v, \ell)$ will represent both a set in the instance $(U, \mathcal{X}, t)$, and a location in the instance $(V, \mathcal{S}, d)$. Thus, we have $V = \{X(v, \ell) \in \mathcal{X}\}$. Furthermore, we create a set of clients $S(e)$ for every element $e \in U$, which consists of all the locations whose corresponding sets in the MCSP instance contain the element e. Thus, we have $\mathcal{S} = \{S(e) : e \in U\}$, where $S(e) = \{X(v, \ell) \in \mathcal{X} : e \in X(v, \ell)\}$ for all $e \in U$. This step is same as in Lemma 4.7.
- We now describe how to embed the locations in V on a given line. For every vertex $v \in \mathcal{V}_j$, $j \in [r]$, the locations $X(v, 1), \ldots, X(v, |L_j|)$ are placed next to one another in sequence, in such a way that the distance between any two consecutive locations is exactly one. Formally, this gives $d(X(v, \ell), X(v, \ell')) = |\ell' - \ell|$ for all $\ell, \ell' \in L_j$. Furthermore, we ensure that any two locations corresponding to two different vertices in $\mathcal{V}$ are *not close to each other*. To be more specific, we have the following guarantee: $d(X(v, \ell), X(v', \ell')) \geq 2$ whenever $v \neq v'$. It is easy to verify that d is a line metric.
- We define $k \leftarrow |\mathcal{X}| - t$.

Note that as $k = |\mathcal{X}| - t$, there is a one to one correspondence between the solutions to the MCSP instance and the solutions to the Robust k-Median instance. Specifically, a set in $\mathcal{X}$ is picked by a solution to the MCSP instance iff the corresponding location is *not* picked in the Robust k-Median instance.

Lemma 4.15 (Yes-Instance)**:** Suppose that there is a labeling strategy σ that strongly satisfies every edge in the HGLC instance (G, π, r). Then there is a solution to the Robust k-Median instance $(V, \mathcal{S}, d)$ with objective one.

Proof. Recall the proof of Lemma 4.8. We construct a solution $\mathcal{X}' \subseteq \mathcal{X}$, $|\mathcal{X}'| = t$, to the MCSP instance $(U, \mathcal{X}, t)$ as follows. For every $v \in \mathcal{V}_j, j \in [r]$, the solution $\mathcal{X}'$ contains the set $X(v, \sigma(v))$. Now, focus on the corresponding solution $F_{\mathcal{X}'} \subseteq V$ to the Robust k-Median instance, which picks a location X iff $X \notin \mathcal{X}'$. Hence, for every vertex $v \in \mathcal{V}_j$, $j \in [r]$, all but one of the locations $X(v, 1), \ldots, X(v, |L_j|)$ are included in $F_{\mathcal{X}'}$. Since any two consecutive locations in such a sequence are unit distance away from each other, the cost of connecting any location in V to the set $F_{\mathcal{X}'}$ is either zero or one, i.e., $d(X, F_{\mathcal{X}'}) \in \{0, 1\}$ for all $X \in V = \mathcal{X}$.

For the rest of the proof, fix any set of clients $S(e) \in \mathcal{S}$, $e \in U$. The proof of Lemma 4.8 implies that the element e incurs congestion one under $\mathcal{X}'$. Hence, the element belongs to exactly one set in $\mathcal{X}'$, say X^*. Again, comparing the solution $\mathcal{X}'$ with the corresponding solution $F_{\mathcal{X}'}$, we infer that $S(e) - F_{\mathcal{X}'} = \{X^*\}$. In other words, every location in $S(e)$, except X^*, is present in the set $F_{\mathcal{X}'}$. The clients in such locations require zero cost for getting connected to $F_{\mathcal{X}'}$. Thus, the total cost of connecting the clients in $S(e)$ to the set $F_{\mathcal{X}'}$ is at most: $\sum_{X \in S(e)} d(X, F_{\mathcal{X}'}) = d(X^*, F_{\mathcal{X}'}) \leq 1$.

Thus, we see that every set of clients in $\mathcal{S}$ requires at most unit cost for getting connected to $F_{\mathcal{X}'}$. So the solution $F_{\mathcal{X}'}$ to the Robust k-Median instance indeed has objective one. □

Lemma 4.16 (No-Instance)**:** If every labeling weakly satisfies at most a $2^{-\gamma r}$ fraction of the edges in the HGLC instance (G, π, r), for some constant γ then every solution to the Robust k-Median instance $(V, \mathcal{S}, d)$ has objective at least r.

Proof. Fix any solution $F \subseteq V$ to the Robust k-Median instance $(V, \mathcal{S}, d)$, and let $\mathcal{X}'_F \subseteq \mathcal{X}$ denote the corresponding solution to the MCSP instance $(U, \mathcal{X}, t)$. By Lemma 4.13 there is some element $e \in U$ with congestion at least r under $\mathcal{X}'_F$. In other words, there are at least r sets $X_1, \ldots, X_r \in \mathcal{X}'_F$ that contain the element e. The locations corresponding to these sets are not picked by the solution F. Furthermore, the way the locations have been embedded on a line ensures that the distance between any location and its nearest neighbor is at least one. Hence, we have $d(X_i, F) \geq 1$ for all $i \in [r]$. Summing over these distances, we infer that the total cost of connecting the clients in $S(e)$ to F is at least $\sum_{i \in [r]} d(X_i, F) \geq r$. Thus, the solution F to the Robust k-Median instance has objective at least r. □

Finally, applying Lemmas 4.15, 4.16, and an argument similar to the proof of Theorem 4.14, we get the following result.

Theorem 4.17: The Robust k-Median problem $(V, \mathcal{S}, d)$ is $\Omega(\log m / \log \log m)$ hard to approximate even on line metrics, where $m = |\mathcal{S}|$, unless $\mathsf{NP} \subseteq \cap_{\delta > 0} \mathsf{DTIME}(2^{n^\delta})$.

4.5 Heuristics

Robust k-Median is a real-world problem and as such needs to be solved as well as possible despite its hardness of approximation. In this section, we complement our negative theoretical results with an experimental evaluation of different simple heuristics for Robust k-Median. In particular we look at two variants of a greedy strategy and two variants of a local search approach. We consider a slight generalization of the problem where clients and facilities are at separate locations. This is more realistic and no easier than the original problem, as one can simply place a facility at every client position to solve an instance of the problem as defined in Definition 4.1.

This is by no means an exhaustive exploration of the possible solution space. However, the results we obtain indicate that a heuristic treatment of Robust k-Median can yield surprisingly good solutions, even if the heuristics are very naive.

For our experiments we consider instances in the plane, as these are closest to the real-world motivation for the problem. We wanted to check how the structure of the instance influences the performance of the heuristics. We suspected that instances where the clients are distributed uniformly are easy, as intuitively a solution that is good for one group of clients is good for all groups.

The robust version of k-median is considered because often the exact set of clients is not known before choosing facility locations and one wants to perform well even if the worst set of possible clients turns out to be realized. It is reasonable to assume that every group of clients has something in common, for example that they come from a similar region, like a city. Therefore more realistic instances for Robust k-Median have the groups form clusters in space. We also generate such instances for testing our heuristics.

4.5.1 Methods

Since solving Robust k-Median instances to optimality is infeasible for the instances we consider[1], we compare the performance of the various heuristics to the value of a LP-relaxation. We have a variable x_j for each possible median location and variables y_{ij} that indicate whether client i is served by facility j. The LP is then as follows.

$$
\begin{aligned}
\min \quad & T \\
s.t. \quad & y_{ij} - x_j \leq 0 && \forall i, j \\
& \sum_j y_{ij} \geq 1 && \forall i \\
& \sum_{i \in g} \sum_j d(i,j) \cdot y_{ij} \leq T && \forall \text{ groups of clients } g \\
& \sum_j x_j \leq k \quad \text{and} \quad 0 \leq x_j \leq 1 && \forall j \\
& 0 \leq y_{ij} \leq 1 && \forall i, j
\end{aligned}
$$

[1] We attempted solving three instances optimally, see Figure 4.2, but gave up on the third after nearly half a year of CPU time was consumed.

To solve the LP we use the Gurobi solver [33], version 5.5.0, on a 64-bit Linux system.

Note that the assignment of the y_{ij} variables is immediately clear from the assignment of the x_j. For location i, let $j_1, j_2, \ldots j_n$ be the locations ordered by increasing distance. Then $y_{ij_\ell} = \min(x_{j_\ell}, 1 - (y_{ij_1} + \ldots + y_{ij_{\ell-1}}))$. The constraint $y_{ij_\ell} \leq \min(\,,\,)$ is already expressed by the first two constraints. It could however be put into the objective via the big M-method. Consider a minimization problem min T subject to $x = \min(b, c)$. Let M be large integer and consider min $T + Mt$ subject to $x \leq b$, $x \leq c$, $t \leq b - x$, and $t \leq c - x$. Observe that $t = \min(b, c) - x$ in an optimal solution. One needs to choose M big enough so that t must be zero in an optimal vertex solution. It is however unclear whether this will speed up the solution. We have not tried this method.

We implemented and compared the following heuristics:

Greedy Upwards. Initialize all facilities as closed. Open the facility that reduces the cost maximally. Repeat until k facilities are open.

Greedy Downwards. Initialize all facilities as open. Close the facility that increases the cost minimally. Repeat until k facilities are open.

Local Search. Open k random facilities. Compare all solutions that can be obtained from the current solution by closing ℓ facilities and opening ℓ facilities. Replace the current solution by the best solution found. Repeat until the current solution is a local optimum. In the experiments we use $\ell = 2$.

Randomized Local Search. Same as Local Search, but instead of considering *all* solutions in the neighborhood, sample only a random subset. The size of the subset is an additional parameter to the heuristic. In the experiments we use $\ell = 3$ and 200 random neighbors.

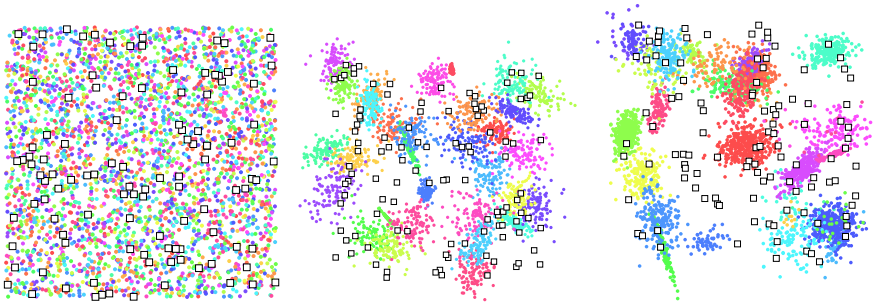
Note that taking the solution of one of the greedy algorithms as starting point for a local search is an obvious improvement, but this would prevent us from comparing the local search algorithm with the greedy heuristic.

The local search heuristic is closely related to Lloyd's algorithm for k-means. In Lloyd's algorithm, a random set of centers is chosen and iteratively updated by moving the centers to the centroids of the clients that fall in their voronoi cell. This improves the total distance from the centers to all clients in every iteration.

In our setting, we want to reduce the cost of the group of clients that currently incurs the maximal cost. This can be done by moving a facility closer to this group of clients, that is, closing one facility and opening another that reduces the objective function. The local search algorithm, by closing and opening more than one facility at a time, does this at least as well.

We create instances in the plane and use the euclidean distance. We create two types of instances. In the first type the clients and facilities are uniformly distributed in a 100×100 square. We call these instances the *uniform* instances. In these instances all groups of clients contain the same number of clients. The k we use for the experiments is 7.

The second kind uses random gaussian distributions to sample client positions. To generate the gaussian distributions we sample a 2×2-matrix M with v_1, v_2 on



(a) Uniform (b) Gauss-Const (c) Gauss-Exp

Figure 4.1: Examples for the kind of instances we generate. Circles are clients, squares are facilities, colors indicate group membership.

the diagonal, where the two values are chosen uniformly at random from $[0, 50]$. This matrix describes the major axes of the gaussian ellipse. It is then rotated by a uniformly random angle. The result is the covariance matrix of the gaussian distribution. The mean is a random point in a 100×100 square. These instances we call *gauss*. We generate two subgroups of instances, in the first subgroup, *gauss-const*, all groups of clients have the same number of clients, in the second subgroup, *gauss-exp*, the number of clients in a group is sampled from an exponential distribution. Figure 4.1 shows examples for the different kind of instances we generate.

As we didn't put much effort into optimizing our heuristics for speed (for example we do not use spatial search structures to find nearest neighbors), we do not report execution time and focus solely on solution quality. Nevertheless it is clear that the greedy strategies are much simpler to implement and much faster than the local search heuristics.

We report average performance on instances where the solution is worse than the LP value, as small, easy instances otherwise skew the results. To conclude relative performance advantages between heuristics we use a Wilcoxon signed-rank test as implemented in SciPy 0.12.0.

All computer code we wrote to run the experiments and analyze the results, as well as the instances we solved, is available online at `http://resources.mpi-inf.mpg.de/robust-k-median/code-data.7z`.

4.5.2 Results

Table 4.1 summarizes the results of the experiments, Table 4.2 shows the performance for the different instance sizes for the Greedy Upwards and the Local Search heuristic. The performance differences in Table 4.1 are statistically significant with a very small two-sided p-value, except for the difference between Greedy Downward and Randomized Local Search on Uniform and Gauss-Const instances. In these cases the p-value is 0.66, respectively 0.08.

Since we use an LP relaxation as a comparison point, it is not immediately clear what these numbers mean for the actual quality of the solution. If the heuristic and

Heuristic	Uniform	Gauss-Const	Gauss-Exp
Greedy Up	1.65 (1.49)	5.18 (5.24)	6.63 (5.94)
Greedy Down	1.45 (1.42)	2.92 (2.92)	2.12 (2.05)
Local Search	1.13 (1.12)	1.63 (1.62)	1.41 (1.39)
Randomized Local Search	1.53 (1.48)	2.15 (2.29)	2.37 (2.36)

Table 4.1: Mean Performance as a multiple of the LP relaxation value, rounded to three digits. In parentheses we provide the median. 1654 uniform instances, 1009 Gauss-Const instances, and 2029 Gauss-Exp instances of varying sizes were solved. The reported performance is over the instances where the heuristics perform worse that the LP relaxation.

the LP have similar costs, we know that the LP bound was good (as the heuristics produce integer solutions). However, we do not know whether the instances where the heuristics find a worse solution are actually hard for the heuristics or whether the LP relaxation provides weak bound. To investigate this we had a closer look at instances where both Greedy down and Local Search perform badly. For three instances we attempted to solve the integer linear program and succeeded for two of them. In Figure 4.2 we see different solutions. For these instances at least it was indeed the case that the LP relaxation yielded a bad bound. This suggests that the heuristics work even better than the numbers in Table 4.1 indicate.

4.6 Conclusion

We show a logarithmic lower bound for Robust k-median on the uniform and line metrics. Despite this result, heuristics perform very well empirically.

As expected instances where the *robust* nature of Robust k-Median are not as important because groups are distributed uniformly are easier than the more realistic instances where groups form clusters. For the two better heuristics, Greedy Downwards and Local Search, also perform better on instances with uneven group sizes. Here too, one can speculate that few groups dominate the problem, and finding a solution that minimizes maximum costs becomes easier.

The good performance of these simple heuristics indicate that although Robust k-Median is hard to approximate in the worst case, a heuristic treatment can effectively find a very good approximation. Moreover, these results suggest that Robust k-Median might become easier to approximate if some natural restrictions are assumed.

For instance, if we assume that the diameter of each set S_i is at most an ϵ fraction of the diameter $\Delta = \max_{u,v} d(u, v)$, can we obtain a constant approximation factor? This case captures the notion of "locality" of the communities. We note that in our hardness instances the diameter of each set S_i is Δ for uniform metric and at least $\Delta/2$ in the line metric, so these hard instances would not arise if we have the locality assumption. Another interesting case is a random instance where the sets S_i are randomly generated by an unknown distribution.

The problem is also interesting in a parametrized complexity setting. In particular, can we obtain an $O(1)$ approximation algorithm in time $g(k)\,\mathrm{poly}(n)$?

Clients	Facilities									
	10		110		210		310		410	
	Greedy	Local Search	GD	LS	GD	LS	GD	LS	GD	LS
10	1.00	1.00	1.12	1.00	1.31	1.01	1.39	1.02	1.40	1.01
160	1.01	1.01	1.6	1.17	1.63	1.17	1.68	1.15	1.63	1.15
310	1.01	1.01	1.64	1.21	1.69	1.19	1.70	1.19	1.75	1.18
460	1.01	1.01	1.68	1.22	1.73	1.21	1.71	1.21	1.73	1.21
110	1.00	1.00	1.17	1.01	1.22	1.01	1.25	1.01	1.24	1.01
1760	1.0	1.0	1.28	1.06	1.33	1.06	1.34	1.06	1.34	1.06
3410	1.0	1.0	1.3	1.07	1.33	1.07				

(a) Uniform

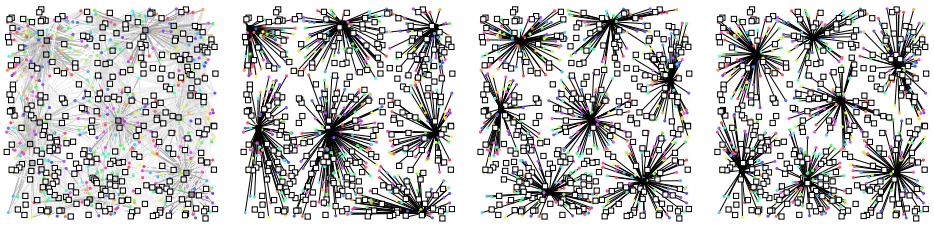
Clients	Facilities									
	10		110		210		310		410	
	Greedy	Local Search	GD	LS	GD	LS	GD	LS	GD	LS
10	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
160	1.0	1.0	2.74	1.64	3.05	1.6	3.33	1.62	3.33	1.57
310	1.0	1.0	2.76	1.70	3.07	1.66	3.32	1.64		
110	1.0	1.0	1.0	1.0	1.0*	1.0*				
3410	1.01	1.0	2.74	1.65	3.02*	1.63*				

(b) Gauss-Const

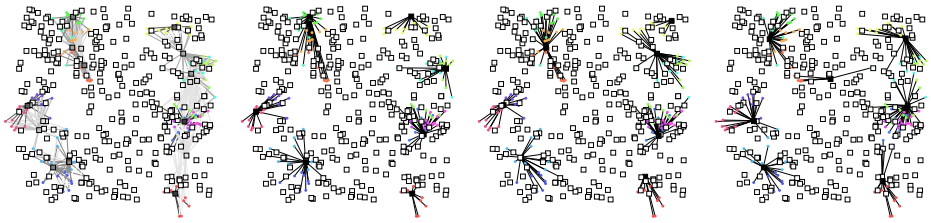
Clients	Facilities									
	10		110		210		310		410	
	Greedy	Local Search	GD	LS	GD	LS	GD	LS	GD	LS
10					1.0*	1.0*	1.0	1.0	1.0	1.0
110	1.0	1.0	1.34	1.16	1.66	1.28	1.65	1.26	1.91	1.34
210	1.0	1.0	1.9	1.41	2.14	1.45	2.31	1.46	2.46	1.49
310	1.0	1.0	2.23	1.48	2.6	1.48	2.69	1.51	2.78	1.50
110	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.01	1.0
1210	1.0	1.0	1.38	1.21	1.56	1.23	1.73	1.29	1.77	1.29
2310	1.0	1.0	1.94	1.38	2.09	1.44	2.48	1.41	2.29	1.44
3410	1.0	1.0	2.17	1.51	2.48	1.48	2.8	1.55		

(c) Gauss-Exp

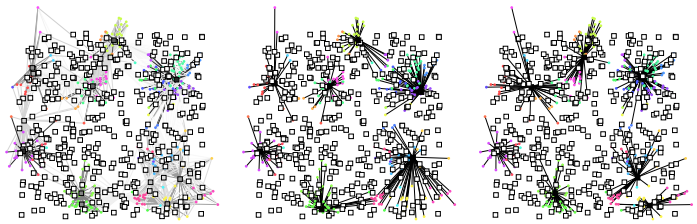
Table 4.2: Performance depending on instance size for the Greedy Downwards and Local Search heuristics. All values are averages over 50 instances, except for those marked by *. For Gauss-Exp instances the number of clients is the mean of the exponential distribution times the number of groups. Values above the horizontal line come from instances with 10 clients per group, below the line instances have 110 clients per group. Missing entries took too long to solve, or aren't interesting.



(a) Uniform: LP value 2806.4, Greedy value 5982.39, Local Search value 3426.43, OPT 3230.19.



(b) Gauss-Const: LP value 1360.26, Greedy value 8307.48, Local Search value 2541.21, OPT 2505.26



(c) Gauss-Exp: LP value 2362.06, Greedy value 10624.4, Local Search value 4354.54, 4192.31 $\leq$ OPT $\leq$ 4354.54

Figure 4.2: Solutions of the different algorithms on particularly hard instances. From left to right, the LP solution, the Greedy downwards solution, the Local Search solution and the ILP solution. Darkness of facilities indicates "how open" they are in the LP relaxation. In 4.2c we stopped the ILP solver, after having consumed 177 days of CPU time and 46GB of memory.

Appendices

A Computing a Spanning Subgraph of an Overlap Graph

We say two intervals $[a, b]$ and $[c, d]$ overlap if $a < c < b < d$ or vice versa. Given a set of intervals this relation induces a graph, the *overlap graph*. This section explain how to compute a spanning subgraph of this graph in linear time. We use this result in Chapter 2 to compute an ordering in which to add chains to a subdivision.

We first assume that the endpoints of all intervals are pairwise distinct. We will later show how to remove this assumption by perturbation.

For every interval $I = [a, b]$ define its set of left and right neighbors:

$$L(I) = \{I' = [a', b']; a' < a < b' < b\},$$
$$R(I) = \{I' = [a', b']; a < a' < b < b'\}.$$

If the set of left neighbors is non-empty, let the interval $I' \in L(I)$ with the rightmost right endpoint be the immediate left neighbor of I. Similarly, if the set of right neighbors is non-empty, the immediate right neighbor of I is the interval in $R(I)$ with the leftmost left endpoint.

Lemma .18: The graph G' formed by connecting each interval to its immediate left and right neighbor (if any) forms a spanning subgraph of the overlap graph G and has exactly the same connected components.

Proof. Clearly, every edge of G' is also an edge of G and hence connected components of G' are subsets of connected components of G.

For the other direction, assume I and I' are overlapping intervals that are not connected in G' and for which the left endpoint of I is as small as possible. Then $a < a' < b < b'$, where $I = [a, b]$ and $I' = [a', b']$. Since $I' \in R(I)$, but I and I' are not connected, I has an immediate right neighbor $J \neq I'$. The left endpoint of J must thus be smaller than a' and the right endpoint of J must be larger than b (since I overlaps with J). If the right endpoint is smaller than b', J and I' overlap. By repeating this argument for J and I', we must reach an interval $U = [c, d]$ containing I'. Thus

$$a < c < a' < b < b' < d.$$

Starting from I' and going to left neighbors, we obtain in the same fashion an interval $U' = [c', d']$ with

$$c' < a < a' < b < d' < b'.$$

Algorithm 4: Finding a spanning forest of a overlap graph

Data: $I = \{[a_0, a_0'], \ldots, [a_\ell, a_\ell']\}$
stack = [ ]
sort I lexicographically in descending order
for $[l, r]$ *in* I **do**
  while *stack not empty and* $r >$ *top(stack) right endpoint* **do**
    pop(stack)
  if *stack not empty and* $r \geq$ *top(stack) left endpoint* **then**
    connect $[l, r]$, top(stack)
  push(stack, $[l, r]$)
stack = [ ]
sort I lexicographically in ascending order where the key for $[l, r]$ is $[r, l]$
for $[l, r]$ *in* I **do**
  while *stack not empty and l<top(stack) left endpoint* **do**
    pop(stack)
  if *stack not empty and* $l \leq$ *top(stack) right endpoint* **then**
    connect $[l, r]$, top(stack)
  push(stack, $[l, r]$)

We conclude that U' and U overlap, but are not connected in G'. Since the left endpoint of U' is to the left of the left endpoint of I, this contradicts the choice of I and I'. □

It is easy to determine all immediate right neighbors by a linear time sweep over all intervals. We sort the intervals in decreasing order of left endpoint and then sweep over the intervals starting with the interval with rightmost left endpoint. We maintain a stack S of intervals, initially empty. If $I_1 = [a_1, b_1], \ldots, I_k = [a_k, b_k]$ are the intervals on the stack with I_1 being on the top of the stack, then $a_1 < a_2 < \ldots < a_k$ and $b_1 < b_2 < \ldots < b_k$, I_1 is the last interval processed, and $I_{\ell+1}$ is the immediate right neighbor of I_ℓ if I_ℓ has right neighbors. If I_ℓ does not have right neighbors, $a_{\ell+1} > b_\ell$. Let $I = [a, b]$ be the next interval to be processed. Its immediate right neighbor is the topmost interval I_ℓ on the stack with $b_\ell > b$ (if any). Hence we pop intervals I_ℓ from the stack while $b > b_\ell$ and then connect I to the topmost interval if $b > a_\ell$, and push I. The determination of immediate left neighbors is symmetric.

It remains to deal with intervals with equal endpoints. We do so by perturbation. It is easy to see that the following rules preserve the overlaps-relation and eliminate equal endpoints.

(1) if a left and a right endpoint are at the same coordinate, then the left endpoint precedes the right endpoint.

(2) if two left endpoints are equal, the one belonging to the shorter interval is smaller.

(3) if two right endpoints are equal, the one belonging to the shorter interval is larger.

(4) if two intervals are equal, one is slightly shifted to the right.

In other words, the endpoints of an interval $I_i = [a, b]$ are replaced by $((a, -1, b - a, i)$ and $(b, 1, b - a, i))$ and comparisons are lexicographic. The perturbation need not be made explicitly, it can be incorporated into the sorting order and the conditions under which edges are added, as described in Algorithm 4.

B Hypergraph Label Cover

An instance of r- is equivalent to the r-Prover system as used by Feige [28] in proving the hardness of approximation for Set Cover. We discuss the equivalence in this section. We use very similar techniques in the proof of Theorem 4.3 in Chapter 4.

In the r-prover system, there are r provers $P_1, \ldots, P_r$ and a verifier V. Each prover is associated with a codeword of length r in such a way that the hamming distance between any pair P_i, P_j is at least $\mathsf{ham}(P_i, P_j) = r/2$; this is possible if r is a power of two because we can use Hadamard code. Given an input 3-SAT formula ϕ, the verifier selects r clauses uniformly and independently at random. Call these clauses $C_1, \ldots, C_r$. From each such clause, the verifier selects a variable uniformly and independently at random. These variables are called $x_1, \ldots, x_r$. Prover P_i receives a clause C_j if the jth bit of its codeword is 0; otherwise, it receives variable x_j. The property of Hadamard code guarantees that each prover would receive $r/2$ clauses and $r/2$ variables.

Then each prover P_i is expected to give an assignment to all involved variables it receives and sends this assignment to the verifier. The verifier then looks at the answers from r provers and has two types of acceptance predicates.

- (Weak acceptance) At least one pair of answers is consistent.
- (Strong acceptance) All pairs of answers are consistent.

Applying parallel repetition theorem [62], Feige argues the following.

Theorem .19: ([28, Lemma 2.3.1]) If Φ is a satisfiable 3-SAT(5) formula, then there is provers' strategy that always causes the verifier to accept. Otherwise, the verifier weakly accepts with probability at most $r^2 2^{-\gamma r}$ for some universal constant $\gamma > 0$.

Now we show how Theorem 4.3 follows by constructing the instance of (V, E) based on the r-prover system. For each prover j, we create a set V_j consisting of vertices v that correspond to possible query sent to prover j, so we have $|V_j| = (5n/3)^{r/2} n^{r/2}$. For each possible random string x, we have an edge h_x that contains r vertices, corresponding to queries sent to the provers. It can be checked that the total number of possible random strings is $(5n)^r$, and the degree of each vertex is $3^{r/2}5^{r/2} = 15^{r/2}$; notice that this is equal to $r|E|/|V|$. A prover strategy corresponds to the label of vertices, and the acceptance probability is exactly the fraction of satisfied edges. Moreover, for each possible query, the number of possible answers is at most 7^r (for each clause, there are 7 ways to satisfy it). This implies that $|L_j| \leq 7^r$.

Bibliography

[1] *A lost interview with ENIAC co-inventor J. Presper Eckert*. URL: https://web.archive.org/web/20140912140810/http://www.computerworld.com/article/2561813/computer-hardware/q-a--a-lost-interview-with-eniac-co-inventor-j--presper-eckert.html.

[2] L. Ahlroth, O. Pottonen, and A. Schumacher. "Approximately Uniform Online Checkpointing with Bounded Memory". In: *Algorithmica* 67.2 (2013), pp. 234–246. DOI: 10.1007/s00453-013-9772-5.

[3] E. Alkassar et al. "A Framework for the Verification of Certifying Computations". English. In: *Journal of Automated Reasoning* 52.3 (2014), pp. 241–273. ISSN: 0168-7433. DOI: 10.1007/s10817-013-9289-2.

[4] B. M. Anthony et al. "A Plant Location Guide for the Unsure: Approximation Algorithms for Min-Max Location Problems". In: *Math. Oper. Res.* 35.1 (2010 (Also in SODA 2008)), pp. 79–101.

[5] S. Arora. "Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems". In: *J. ACM* 45.5 (1998), pp. 753–782.

[6] S. Arora et al. "Proof Verification and the Hardness of Approximation Problems". In: *J. ACM* 45.3 (1998), pp. 501–555.

[7] V. Arya et al. "Local Search Heuristics for k-Median and Facility Location Problems". In: *SIAM J. Comput.* 33.3 (2004), pp. 544–562.

[8] N. Bansal et al. "On generalizations of network design problems with degree bounds". In: *Math. Program.* 141.1-2 (2013), pp. 479–506.

[9] D. W. Barnette and B. Grünbaum. "On Steinitz's theorem concerning convex 3-polytopes and on some properties of 3-connected graphs". In: *Many Facets of Graph Theory*. 1969, pp. 27–40.

[10] M. W. Bern et al. "On-Line Algorithms for Locating Checkpoints". In: *Algorithmica* 11.1 (1994), pp. 33–52.

[11] S. Bhattacharya et al. “New Approximability Results for the Robust k-Median Problem”. In: *Algorithm Theory - SWAT 2014 - 14th Scandinavian Symposium and Workshops, Copenhagen, Denmark, July 2-4, 2014. Proceedings*. 2014, pp. 50–61. DOI: 10.1007/978-3-319-08404-6_5. URL: http://dx.doi.org/10.1007/978-3-319-08404-6_5.

[12] R. E. Bixby. “Solving Real-World Linear Programs: A Decade and More of Progress”. In: *Oper. Res.* 50.1 (Jan. 2002), pp. 3–15. ISSN: 0030-364X. DOI: 10.1287/opre.50.1.3.17780.

[13] M. Blum. “Program Result Checking: A New Approach to Making Programs More Reliable”. In: *Automata, Languages and Programming, 20nd International Colloquium, ICALP'93, Lund, Sweden, July 5-9, 1993, Proceedings*. 1993, pp. 1–14. DOI: 10.1007/3-540-56939-1_57.

[14] M. Blum and S. Kannan. “Designing Programs That Check Their Work”. In: *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*. 1989, pp. 86–97. DOI: 10.1145/73007.73015.

[15] J. A. Bondy and U. S. R. Murty. *Graph Theory*. Springer, 2008.

[16] *Breakthrough performance by Pocket Fritz 4 in Buenos Aires*. URL: http://en.chessbase.com/post/breakthrough-performance-by-pocket-fritz-4-in-buenos-aires.

[17] K. Bringmann et al. “Online Checkpointing with Improved Worst-Case Guarantees”. In: *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I*. 2013, pp. 255–266. DOI: 10.1007/978-3-642-39206-1_22.

[18] K. M. Chandy and C. V. Ramamoorthy. “Rollback and Recovery Strategies for Computer Programs”. In: *IEEE Transactions on Computers* C-21 (1972), pp. 546–556. ISSN: 0018-9340. DOI: 10.1109/TC.1972.5009007.

[19] M. Charikar and S. Guha. “Improved Combinatorial Algorithms for the Facility Location and k-Median Problems”. In: *FOCS*. IEEE Computer Society, 1999, pp. 378–388.

[20] M. Charikar et al. “A Constant-Factor Approximation Algorithm for the k-Median Problem”. In: *J. Comput. Syst. Sci.* 65.1 (2002), pp. 129–149.

[21] L. Chen and A. Avizienis. “N-Version Programming: A fault-tolerance approach to reliability of software operation”. In: *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*. 1978, pp. 3–9.

[22] J. N. Corcoran, U. Schneider, and H.-B. Schüttler. “Perfect Stochastic Summation in High Order Feynman Graph Expansions”. In: *International Journal of Modern Physics C* 17.11 (2006), pp. 1527–1549.

[23] J. Dean. *Underneath the Covers at Google: Current Systems and Future Directions*. 2008. URL: https://sites.google.com/site/io/underneath-the-covers-at-google-current-systems-and-future-directions.

[24] *Deep Blue*. English. IBM. URL: https://web.archive.org/web/20140822073803/http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/.

[25] F. Dehne et al. "The cluster editing problem: Implementations and experiments". In: *Parameterized and Exact Computation* (2006), pp. 13–24.

[26] E. Dinits, A. Karzanov, and M. Lomonosov. "On the structure of a family of minimal weighted cuts in graphs". In: *Studies in Discrete Mathematics (in Russian)*. 1976, pp. 290–306.

[27] E. N. M. Elnozahy et al. "A survey of rollback-recovery protocols in message-passing systems". In: *ACM Computing Surveys* 34.3 (2002), pp. 375–408. ISSN: 0360-0300.

[28] U. Feige. "A threshold of ln n for approximating set cover". In: *J. ACM* 45.4 (1998), pp. 634–652.

[29] T. Fleiner and A. Frank. *A quick proof for the cactus representation of mincuts*. Tech. rep. QP-2009-03. Egerváry Research Group, Budapest, 2009.

[30] H. N. Gabow. "Path-based depth-first search for strong and biconnected components". In: *Inf. Process. Lett.* 74.3-4 (2000), pp. 107–114. ISSN: 0020-0190. DOI: http://dx.doi.org/10.1016/S0020-0190(00)00051-X.

[31] Z. Galil and G. F. Italiano. "Reducing edge connectivity to vertex connectivity". In: *SIGACT News* 22.1 (1991), pp. 57–61. ISSN: 0163-5700. DOI: http://doi.acm.org/10.1145/122413.122416.

[32] E. Gelenbe. "On the Optimum Checkpoint Interval". In: *J. ACM* 26.2 (1979), pp. 259–270. ISSN: 0004-5411. DOI: 10.1145/322123.322131.

[33] I. Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2013. URL: http://www.gurobi.com.

[34] C. Gutwenger and P. Mutzel. "A Linear Time Implementation of SPQR-trees". In: *Proceedings of the 8th International Symposium on Graph Drawing (GD'00)*. 2001, pp. 77–90. ISBN: 3-540-41554-8.

[35] T. Hagerup and C. Rüb. "A guided tour of Chernoff bounds". In: *Information Processing Letters* 33.6 (1990), pp. 305 –308. ISSN: 0020-0190. DOI: http://dx.doi.org/10.1016/0020-0190(90)90214-I.

[36] T. L. Heath. *A History of Greek Mathematics*. Vol. 1.

[37] V. Heuveline and A. Walther. "Online Checkpointing for Parallel Adjoint Computation in PDEs: Application to Goal-Oriented Adaptivity and Flow Control". In: *Euro-Par 2006 Parallel Processing*. Vol. 4128. Lecture Notes in Computer Science. Springer, 2006, pp. 689–699. ISBN: 978-3-540-37783-2. DOI: 10.1007/11823285_72.

[38] M. Hilbert and P. López. "The World's Technological Capacity to Store, Communicate, and Compute Information". In: *Science* 332.6025 (2011), pp. 60–65. DOI: 10.1126/science.1200970. eprint: http://www.sciencemag.org/content/332/6025/60.full.pdf.

[39] J. Hopcroft and R. Tarjan. "Efficient planarity testing". In: *J. ACM* 21.4 (1974), pp. 549–568.

[40] J. E. Hopcroft and R. E. Tarjan. "Dividing a graph into triconnected components". In: *SIAM J. Comput.* 2.3 (1973), pp. 135–158.

[41] D. R. Karger. "Minimum cuts in near-linear time". In: *J. ACM* 47.1 (2000), pp. 46–76. ISSN: 0004-5411. DOI: 10.1145/331605.331608.

[42] S. G. Kolliopoulos and S. Rao. "A nearly linear-time approximation scheme for the Euclidean k-median problem". In: *Algorithms-ESA'99*. Springer, 1999, pp. 378–389.

[43] D. Kratsch et al. "Certifying algorithms for recognizing interval graphs and permutation graphs". In: *SIAM J. Comput.* 36.2 (2006), pp. 326–353.

[44] S. Li and O. Svensson. "Approximating k-median via pseudo-approximation". In: *STOC*. ACM, 2013, pp. 901–910. ISBN: 978-1-4503-2029-0.

[45] J.-H. Lin and J. S. Vitter. "Approximation Algorithms for Geometric Median Problems". In: *Information Processing Letters* 44.5 (1992), pp. 245–249.

[46] N. Linial, L. Lovász, and A. Wigderson. "Rubber bands, convex embeddings and graph connectivity". In: *Combinatorica* 8.1 (1988), pp. 91–102. DOI: http://dx.doi.org/10.1007/BF02122557.

[47] L. Lovász. "Computing ears and branchings in parallel". In: *Proceedings of the 26th Annual Symposium on Foundations of Computer Science (FOCS'85)*. 1985. DOI: http://doi.ieeecomputersociety.org/10.1109/SFCS.1985.16.

[48] C. Lund and M. Yannakakis. "On the Hardness of Approximating Minimization Problems". In: *J. ACM* 41.5 (1994), pp. 960–981.

[49] W. Mader. "A Reduction Method for Edge-Connectivity in Graphs". In: *Advances in Graph Theory*. Vol. 3. Annals of Discrete Mathematics. 1978, pp. 145–164. DOI: 10.1016/S0167-5060(08)70504-1.

[50] R. M. McConnell et al. "Certifying algorithms". In: *Computer Science Review* 5.2 (2011), pp. 119–161. ISSN: 1574-0137. DOI: DOI:10.1016/j.cosrev.2010.09.009.

[51] K. Mehlhorn. “Nearly Optimal Binary Search Trees”. In: *Acta Informatica* 5 (1975), pp. 287–295.

[52] K. Mehlhorn, S. Näher, and C. Uhrig. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[53] K. Mehlhorn and S. Näher. “LEDA a library of efficient data types and algorithms”. In: *Mathematical Foundations of Computer Science 1989*. Springer. 1989, pp. 88–106.

[54] K. Mehlhorn, A. Neumann, and J. M. Schmidt. “Certifying 3-Edge-Connectivity”. In: *Graph-Theoretic Concepts in Computer Science - 39th International Workshop, WG 2013, Lübeck, Germany, June 19-21, 2013, Revised Papers*. 2013, pp. 358–369. DOI: 10.1007/978-3-642-45043-3_31.

[55] H. Nagamochi and T. Ibaraki. “A linear time algorithm for computing 3-edge-connected components in a multigraph”. In: *Japan Journal of Industrial and Applied Mathematics* 9 (2 1992), pp. 163–180. ISSN: 0916-7005.

[56] H. Nagamochi and T. Ibaraki. *Algorithmic Aspects of Graph Connectivity (Encyclopedia of Mathematics and its Applications)*. Cambridge University Press, 2008.

[57] A. Neumann. “Implementation of Schmidt’s Algorithm for Certifying Triconnectivity Testing”. MA thesis. Universität des Saarlandes and Graduate School of CS, Germany, 2011.

[58] W. D. Nordhaus. “Two Centuries of Productivity Growth in Computing”. In: *The Journal of Economic History* 67 (01 Mar. 2007), pp. 128–159. ISSN: 1471-6372. DOI: 10.1017/S0022050707000058.

[59] S. Olariu and A. Y. Zomaya. “A Time- and Cost-Optimal Algorithm for Interlocking Sets – With Applications”. In: *IEEE Trans. Parallel Distrib. Syst.* 7.10 (1996), pp. 1009–1025. ISSN: 1045-9219. DOI: http://dx.doi.org/10.1109/71.539733.

[60] F. Österlind et al. “Sensornet Checkpointing: Enabling Repeatability in Testbeds and Realism in Simulations”. In: *Wireless Sensor Networks*. Vol. 5432. Lecture Notes in Computer Science. Springer, 2009, pp. 343–357. ISBN: 978-3-642-00223-6. DOI: 10.1007/978-3-642-00224-3_22.

[61] V. Ramachandran. “Parallel Open Ear Decomposition with Applications to Graph Biconnectivity and Triconnectivity”. In: *Synthesis of Parallel Algorithms*. 1993, pp. 275–340.

[62] R. Raz. “A Parallel Repetition Theorem”. In: *SIAM J. Comput.* 27.3 (1998), pp. 763–803.

[63] J. M. Schmidt. "A Simple Test on 2-Vertex- and 2-Edge-Connectivity". In: *Information Processing Letters* 113.7 (2013), pp. 241–244. DOI: 10.1016/j.ipl.2013.01.016.

[64] J. M. Schmidt. *Contractions, Removals and Certifying 3-Connectivity in Linear Time*. Tech. Report B 10-04. Freie Universität Berlin, Germany, 2010.

[65] J. M. Schmidt. "Contractions, Removals and Certifying 3-Connectivity in Linear Time". In: *SIAM Journal on Computing* 42(2) (2013), pp. 494–535.

[66] J. M. Schmidt. "Structure and Constructions of 3-Connected Graphs". PhD thesis. Freie Universität Berlin, Germany, 2011.

[67] C. E. Shannon. "Programming a computer for playing chess". In: *Philosophical magazine* 41.314 (1950), pp. 256–275.

[68] P. Stumm and A. Walther. "New Algorithms for Optimal Online Checkpointing". In: *SIAM Journal on Scientific Computing* 32.2 (2010), pp. 836–854. DOI: 10.1137/080742439. eprint: http://epubs.siam.org/doi/pdf/10.1137/080742439.

[69] G. F. Sullivan and G. M. Masson. "Using certification trails to achieve software fault tolerance". In: *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*. IEEE. 1990, pp. 423–431.

[70] S. Taoka, T. Watanabe, and K. Onaga. "A linear time algorithm for computing all 3-edge-connected components of a multigraph". In: *IEICE Trans. Fundamentals E75* 3 (1992), pp. 410–424.

[71] S. Toueg and Ö. Babaoglu. "On the Optimum Checkpoint Selection Problem". In: *SIAM Journal on Computing* 13.3 (1984), pp. 630–649. DOI: 10.1137/0213039. eprint: http://epubs.siam.org/doi/pdf/10.1137/0213039.

[72] Y. H. Tsin. "A Simple 3-Edge-Connected Component Algorithm". In: *Theor. Comp. Sys.* 40.2 (2007), pp. 125–142. ISSN: 1432-4350. DOI: http://dx.doi.org/10.1007/s00224-005-1269-4.

[73] Y. H. Tsin. "Yet another optimal algorithm for 3-edge-connectivity". In: *J. of Discrete Algorithms* 7.1 (2009), pp. 130–146. ISSN: 1570-8667. DOI: http://dx.doi.org/10.1016/j.jda.2008.04.003.

[74] A. M. Turing. "Faster than Thought". In: Pitman & Sons Ltd., 1953. Chap. Digital computers applied to games.

[75] K.-P. Vo. "Finding triconnected components of graphs". In: *Linear and Multilinear Algebra* 13 (1983), pp. 143–165.

[76] K.-P. Vo. "Segment graphs, depth-first cycle bases, 3-connectivity, and planarity of graphs". In: *Linear and Multilinear Algebra* 13 (1983), pp. 119–141.

[77] S. Yi, D. Kondo, and A. Andrzejak. "Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud". In: *IEEE 3rd International Conference on Cloud Computing (CLOUD 2010)*. 2010, pp. 236–243. DOI: 10.1109/CLOUD.2010.35.