On the Analysis of Two Fundamental Randomized Algorithms

Multi-Pivot Quicksort and Efficient Hash Functions

Dissertation zur Erlangung des akademischen Grades
Doctor rerum naturalium (Dr. rer. nat.)

vorgelegt der Fakultät für Informatik und Automatisierung
der Technischen Universität Ilmenau

von

Dipl.-Inf. Martin Aumüller

geboren am 12.01.1986 in Gera, Deutschland

Eingereicht am 1. Dezember 2014

---

1. Gutachter:
2. Gutachter:
3. Gutachter:

Abstract

Randomization is a central technique in the design and analysis of algorithms and data structures. This thesis investigates the analysis of two fundamental families of randomized algorithms in two different branches of research.

The first part of this thesis considers the sorting problem, i. e., the problem of putting elements of a sequence into a certain order. This is one of the most fundamental problems in computer science. Despite having a large set of different sorting algorithms to choose from, the "quicksort" algorithm turned out to be implemented as the standard sorting algorithm in practically all programming libraries. This divide-and-conquer algorithm has been studied for decades, but the idea of using more than one pivot was considered to be impractical. This changed in 2009, when a quicksort algorithm using two pivots due to Yaroslavskiy replaced the well-engineered quicksort-based sorting algorithm in Oracle's Java 7. The thesis presents a detailed study of multi-pivot quicksort algorithms. It explains the underlying design choices for dual-pivot quicksort algorithms with respect to the comparisons they make when sorting the input. Moreover, it describes two easy to implement dual-pivot quicksort algorithms that are comparison-optimal, i. e., make as few comparisons as possible on average. These algorithms make about $1.8n \ln n$ key comparisons on average when sorting an input of n distinct elements in random order, improving on the $1.9n \ln n$ key comparisons in Yaroslavskiy's algorithm. To analyze quicksort algorithms using more than two pivots, only slight modifications to the dual-pivot case are necessary. This thesis also considers the theoretical analysis of the memory and cache behavior of multi-pivot quicksort algorithms. It will be demonstrated that using more than one pivot makes it possible to improve over the cache behavior of classical quicksort. If no additional space is allowed, using three or five pivots provides the best choice for a multi-pivot quicksort algorithm with respect to cache behavior. Otherwise, significant improvements are possible by using 127 pivots. A large-scale study on the empirical running time of multi-pivot quicksort algorithms suggests that theoretical improvements translate into practice.

The second part of this thesis considers the use of hash functions in algorithms and data structures. Hash functions are applied in many different situations, e. g., when building a hash table or when distributing jobs among machines in load balancing. Traditionally, the analysis of a particular hashing-based algorithm or data structure assumes that a hash function maps keys independently and uniformly at random to a range. Such functions are unrealistic, since their space complexity is huge. Consequently, the task is to construct explicit hash functions providing provable theoretical guarantees. The thesis describes such a construction. The considered hash functions provide sufficient randomness properties for running many different applications, such as cuckoo hashing with a stash, the construction of a perfect hash function, the simulation of a

uniform hash function, load balancing, and generalized cuckoo hashing in a sparse setting with two alternative insertion algorithms. The main contribution of this part of the thesis is a unified framework based on the first moment method. This framework makes it possible to analyze a hashing-based algorithm or data structure only using random graph theory, without exploiting details of the hash function. The hash functions are easy to implement and turn out to be practical while providing strong randomness guarantees.

Zusammenfassung

Die Nutzung von Zufall ist eine wichtige Technik in der Entwicklung und Analyse von Algorithmen und Datenstrukturen. Diese Arbeit beschäftigt sich mit der Analyse von zwei grundlegenden Familien randomisierter Algorithmen in zwei unterschiedlichen Forschungsbereichen.

Im ersten Teil der vorliegenden Arbeit wird das Sortierproblem betrachtet, also das Problem, die Elemente einer Sequenz in eine bestimmte Ordnung zu bringen. Dieses Problem ist eines der grundlegendsten Probleme der Informatik. In den letzten Jahrzehnten wurden eine Vielzahl unterschiedlicher Algorithmen zur Lösung des Sortierproblems vorgestellt. Quicksort, ein auf dem Prinzip von Teile-und-Herrsche basierender Algorithmus, ist dabei der am häufigsten in Programmierbibliotheken genutzte Sortieralgorithmus. Die Idee mehr als ein Pivotelement im Quicksort-Algorithmus zu nutzen, erschien über viele Jahre als unpraktikabel. Dies änderte sich im Jahr 2009, in dem eine elegante Quicksort-Variante von V. Yaroslavskiy zum Standard-Sortierverfahren in Oracles Java 7 wurde. Überraschenderweise basierte dieser Algorithmus auf der Idee zwei Pivotelemente zu nutzen. Die vorliegende Arbeit stellt eine detaillierte Studie von sogenannten Multi-Pivot-Quicksort-Algorithmen dar, also Varianten des Quicksort-Algorithmus, die mit mehr als einem Pivotelement arbeiten. Sie beschreibt dabei die Konstruktionsprinzipien von 2-Pivot-Quicksort-Algorithmen in Bezug auf die Schlüsselvergleiche, die bei der Sortierung einer Eingabe nötig sind. Ein Ergebnis dieser Untersuchung sind zwei leicht zu implementierende 2-Pivot-Quicksort-Algorithmen, die optimal bezüglich der Anzahl an Schlüsselvergleichen sind: Die Kosten stimmen mit einer in dieser Arbeit entwickelten unteren Schranke für die kleinstmöglichen durchschnittlichen Kosten eines 2-Pivot-Quicksort-Algorithmus überein. Auf einer Eingabe, die aus n paarweise verschiedenen Elementen in zufälliger Reihenfolge besteht, werden diese Algorithmen durchschnittlich ungefähr $1.8n \ln n$ Schlüssel miteinander vergleichen, was einen großen Vorteil gegenüber den $1.9n \ln n$ Schlüsselvergleichen im Algorithmus von Yaroslavskiy darstellt. Die Verallgemeinerung der Resultate auf Quicksort-Varianten mit mindestens drei Pivotelementen benötigt nur kleine Anpassungen des entwickelten Modells. Diese Arbeit betrachtet außerdem die theoretische Analyse von Kostenmaßen, die es ermöglichen, Multi-Pivot-Quicksort-Algorithmen hinsichtlich ihres Speicher- und Cacheverhaltens zu vergleichen. Es zeigt sich dabei, dass Ansätze mit mehreren Pivotelementen große Vorteile im Bezug auf diese Kostenmaße gegenüber Standard-Quicksort haben. Wenn es nicht erlaubt ist zusätzlichen Speicher zu allokieren, dann haben Verfahren mit drei oder fünf Pivotelementen das beste Cacheverhalten. Andernfalls können Algorithmen auf Basis vieler Pivotelemente, z. B. 127 Pivotelemente, zu deutlichen Verbesserungen hinsichtlich des Cacheverhaltens führen. Eine umfangreiche Studie der Laufzeit von Multi-Pivot-Quicksort-Algorithmen deutet darauf hin, dass diese theoretischen Vorteile auch in der Praxis zu schnelleren Sortieralgorithmen führen.

Der zweite Teil der vorliegenden Arbeit beschäftigt sich mit dem Einsatz von Hashfunktionen im Rahmen der Entwicklung von Algorithmen und Datenstrukturen. Hashfunktionen bilden eine Kernkomponente vieler Anwendungen, z. B. beim Aufbau einer Hashtabelle oder bei der Verteilung von Jobs auf Maschinen im Rahmen der Lastbalancierung. In der Literatur wird dabei eine praktisch oft nicht zu begründende Annahme getätigt: Die Abbildung von Elementen auf Hashwerte sei voll zufällig; Hashwerte seien also unabhängig und uniform im Wertebereich der Hashfunktion verteilt. Die Speicherplatzkomplexität, die die Beschreibung einer solchen Funktion benötigt, ist für den praktischen Einsatz für gewöhnlich unverhältnismäßig hoch. Das Ziel ist es also, einfache Konstruktionen zu finden, deren Zufallseigenschaften ausreichen, um sie mit beweisbaren theoretischen Garantien praktisch einsetzen zu können. Diese Arbeit beschreibt eine solche einfache Konstruktion von Hashfunktionen, die in einer Vielzahl von Anwendungen beweisbar gut ist. Zu diesen Anwendungen zählen Cuckoo Hashing mit einem sogenannten Stash, die Konstruktion einer perfekten Hashfunktion, die Simulation einer uniformen Hashfunktion, verschiedene Algorithmen zur Lastbalancierung und verallgemeinertes Cuckoo Hashing in einer leicht abgeschwächten Variante mit verschiedenen Einfügealgorithmen. Der zentrale Beitrag dieser Dissertation ist ein einheitliches Analysekonzept. Dieses ermöglicht es, eine auf Hashfunktionen basierende Datenstruktur oder einen auf Hashfunktionen basierenden Algorithmus nur mit Mitteln der Theorie von Zufallsgraphen zu analysieren, ohne Details der Hashfunktion offenzulegen. Die Analysetechnik ist dabei die sogenannte First-Moment-Methode, eine Standardanalysemethode innerhalb der randomisierten Algorithmen. Die Hashfunktionen zeigen gutes Cacheverhalten und sind praktisch einsetzbar.

Acknowledgements

This work would not have been possible without the help and generous contributions of others. I consider many people met along this way to be my dearest friends or greatest mentors.

First and foremost I would like to thank my supervisor Martin Dietzfelbinger. His lectures got me first interested in the area of algorithms and data structures. Over the years, he has been an incredible advisor. Collaborating with him was not only a pleasure but also a tremendous learning experience. Looking back, it seems like most ideas presented in this thesis probably came out from a meeting with him. His vast knowledge, patience, rigor, and generosity with his time and ideas are truly inspiring. He shaped my thinking far beyond the realm of theoretical computer science. If in the future I will ever get the chance to mentor students, I wish to become a mentor to them in the same way Martin Dietzfelbinger has been a mentor to me.

I am greatly honored that Rasmus Pagh and Philipp Woelfel agreed to review this thesis. Both had a direct or indirect impact on this work. Most of the algorithms in the second part of this thesis were first described in a paper co-authored by Rasmus. Philipp provided me with a manuscript that described the idea which eventually led to the framework that is now a foundation of this thesis. Thank you!

This work is greatly influenced by discussions with colleagues. I would like to thank Michael Rink for many interesting and helpful discussions on randomized algorithms, Sebastian Wild for sharing great discussions and ideas on quicksort algorithms, and Timo Bingmann for teaching me many aspects of algorithm engineering and experimental setups. Timo's "sqlplot-tools" saved me countless hours of evaluating and presenting experimental data. I also thank the audience at ESA 2012 and the people at the Dagstuhl seminar on "Data Structures and Advanced Models of Computation on Big Data" for encouraging and/or critical comments on my work.

I have spent the last five years in the company of students, office mates, and colleagues who have made this place feel like home. First, I would like to thank the members—past or present—of the Institute of Theoretical Computer Science, Petra Schüller, Jana Kopp, Dietrich Kuske, Manfred Kunde, Martin Huschenbett, Michael Brinkmeier, Sascha Grau, Ulf Schellbach, Roy Mennicke, Raed Jaberi, and Christopher Mattern for many interesting conversations over lunch and entertaining hiking, cycling, and rafting trips. In particular, I would like to thank Martin for many encouraging debates and coffee breaks, Petra for the relaxing chats, and Jana, for endless support with candy. Moreover, I thank Michael Rossberg for many interesting discussions and a glimpse of the world beyond theoretical computer science. I also enjoyed the company of my fellow conspirator Andre Puschmann.

Over the last years, somewhat unexpectedly, teaching became one of the most enjoyable experiences in my life. I thank all the students who visited my tutorials week after week. Specifically, I thank Steffen Hirte, Pascal Klaue, Tafil Kajtazi, Andreas Seifert, Markus Theil, and Martin Backhaus. In particular, I thank Pascal for allowing me to include some parts of his Master's thesis, which was prepared under my guidance, in this thesis.

The weekly board game evenings have been one of my favorite moments in the week. I thank my fellow "board game geeks" Sascha Grau, Adrian Grewe, Martin Huschenbett, Roy Mennicke, Raed Jaberi, Michael Brinkmeier, and Michael Rink for making these evenings truly entertaining. I also thank my friends Jaehsus, Anna, Felix, Fabian, and Martin for the rafting trips, cycling tours, and so many entertaining evenings over the last years.

Finally I thank my parents, Sabine and Andreas, for providing all the opportunities that allowed me to come this far, and my sister Annelie, for unconditional support. Moreover, I thank my grandfathers, Adolf and Walter, whom I owe many beliefs and ideas about this world. Most of all, I thank Sophia, for all of the above, love, and patience.

Martin Aumüller,
Ilmenau, November 28, 2014.

Contents

1 Outline & Motivation **1**

I Multi-Pivot Quicksort **7**

2 Introduction **9**

3 Basic Approach to Analyzing Dual-Pivot Quicksort **15**
3.1 Basic Setup 15
3.2 Analysis of a Partitioning Step 18
3.2.1 Analysis of the Additional Cost Term 19
3.3 Discussion 25

4 Classification Strategies For Dual-Pivot Quicksort **27**
4.1 Analysis of Some Known Strategies 27
4.2 (Asymptotically) Optimal Classification Strategies 29
4.2.1 Two Unrealistic (Asymptotically) Optimal Strategies 30
4.2.2 Two Realistic Asymptotically Optimal Strategies 35
4.3 Discussion 37

5 Choosing Pivots From a Sample **39**
5.1 Choosing the Two Tertiles in a Sample of Size 5 as Pivots 39
5.2 Pivot Sampling in Classical Quicksort and Dual-Pivot Quicksort 40
5.3 Optimal Segment Sizes for Dual-Pivot Quicksort 41

6 Generalization to Multi-Pivot Quicksort **45**
6.1 General Setup 45
6.2 The Average Comparison Count for Partitioning 49
6.3 Example: 3-pivot Quicksort 53
6.4 (Asymptotically) Optimal Classification Strategies 56
6.4.1 Choosing an Optimal Comparison Tree 56
6.4.2 The Optimal Classification Strategy and its Algorithmic Variant 56
6.4.3 An Oblivious Strategy and its Algorithmic Variant 58
6.5 Guesses About the Optimal Average Comparison Count of k-Pivot Quicksort . . 60

6.6 Discussion . . . 62

7 The Cost of Rearranging Elements 65
7.1 Why Look at Other Cost Measures Than Comparisons . . . 65
7.2 Problem Setting, Basic Algorithms and Related Work . . . 69
7.3 Algorithms . . . 69
7.3.1 Partitioning After Classification . . . 70
7.3.2 Partitioning During Classification . . . 70
7.4 Assignments . . . 73
7.5 Memory Accesses and Cache Misses . . . 80

8 Running Time Experiments 87
8.1 Running Times of Dual-Pivot Quicksort Algorithms . . . 87
8.2 Running Times of k-Pivot Quicksort Algorithms based on "Exchange$_k$" . . . 88
8.3 Running Times of k-Pivot Quicksort Algorithms based on "Permute$_k$" and "Copy$_k$" 91
8.4 Do Theoretical Cost Measures Help Predicting Running Time? . . . 94

9 Conclusion and Open Questions 97

II Hashing 99

10 Introduction 101

11 Basic Setup and Groundwork 107
11.1 The Hash Class . . . 108
11.2 Graph Properties and the Hash Class . . . 111
11.3 Bounding the Failure Term of Hash Class $\mathcal{Z}$ . . . 114
11.4 Step by Step Example: Analyzing Static Cuckoo Hashing . . . 119

12 Randomness Properties of $\mathcal{Z}$ on Leafless Graphs 125
12.1 A Counting Argument . . . 125
12.2 The Leafless Part of $G(S, h_1, h_2)$ . . . 126

13 Applications on Graphs 131
13.1 Cuckoo Hashing (with a Stash) . . . 131
13.2 Simulation of a Uniform Hash Function . . . 136
13.3 Construction of a (Minimal) Perfect Hash Function . . . 139
13.4 Connected Components of $G(S, h_1, h_2)$ are small . . . 143

14 Applications on Hypergraphs 147
14.1 Generalized Cuckoo Hashing . . . 148

14.2 Labeling-based Insertion Algorithms For Generalized Cuckoo Hashing 155
14.3 Load Balancing 164

15 A Generalized Version of the Hash Class **177**
15.1 The Generalized Hash Class 177
15.2 Application of the Hash Class 179
15.3 Discussion 179

16 Experimental Evaluation **181**
16.1 Setup and Considered Hash Families 181
16.2 Success Probability 184
16.3 Running Times 185

17 Conclusion and Open Questions **189**

Bibliography **190**

A Quicksort: Algorithms in Detail **205**

B Details of k-pivot Quicksort Experiments **215**

List of Figures **221**

List of Tables **223**

List of Algorithms **226**

Erklärung **227**

1. Outline & Motivation

Randomness is an ubiquitous tool in computer science. In the design and the analysis of algorithms and data structures, randomness is usually applied in two different ways. On the one hand, in *Average Case Analysis* we assume that the input is random and we make statements about the expected, i. e., average, behavior of a deterministic algorithm over all such inputs. On the other hand, randomness can be used to "cancel out" worst-case inputs. Then we consider the expected behavior of a randomized algorithm on an arbitrary, fixed input. This thesis uses both of these techniques and applies them to two different fundamental topics of computer science: sorting and hashing.

In the first part of this thesis, we consider the average case analysis of new variants of the well-known quicksort algorithm. The purpose of this algorithm is to sort a given input, i. e., to put the elements of a possibly unordered sequence into a particular order. Over the last decades, a great number of different sorting algorithms were developed. A standard textbook on algorithms and data structures like *"Introduction to Algorithms"* by Cormen, Leiserson, Rivest, and Stein [Cor+09] lists twelve different sorting algorithms in the index; most of them are covered extensively. Despite this variety of sorting algorithms, the quicksort algorithm (with its variants), as introduced by Hoare in [Hoa62], turned out to be used dominantly throughout almost all standard libraries of popular programming languages.

Following the divide-and-conquer paradigm, on an input consisting of n elements quicksort uses a pivot element to *partition* its input elements into two parts: the elements in one part being smaller than or equal to the pivot and the elements in the other part being larger than or equal to the pivot; then it uses recursion to sort these parts. This approach—with slight variants such as detecting worst-case inputs or choosing the pivot from a small sample of elements—found its way into practically all algorithm libraries.

This thesis considers variants of quicksort using more than one pivot element. Such variants were deemed to be impractical since Sedgewick's PhD thesis in 1975 [Sed75]. The approach of using more than one pivot was pioneered in the dual-pivot quicksort algorithm of Yaroslavskiy [Yar09] in 2009, which replaced the well-engineered quicksort algorithm in Oracle's *Java 7* shortly after its discovery. This algorithm initiated much research which is documented, e. g., in the papers [WN12; Wil+13; NW14; WNN13; WNM13; Kus+14; MNW15; AD13].

The goal of this thesis is to answer the following general question:

> *How good is multi-pivot quicksort?*

We will identify several advantages of multi-pivot quicksort algorithms over classical quicksort. At the beginning we consider the classical cost measure of counting the average number of key

comparisons between input keys made by a specific sorting algorithm. We will detail the design choices for developing a dual-pivot quicksort, i. e., an algorithm that uses two pivots p and q with $p < q$. This approach will make it possible to analyze the average comparison count of an arbitrary dual-pivot quicksort algorithm. It will turn out that a very simple property of a dual-pivot quicksort algorithm, the average number of times it compares an element to the smaller pivot p first, will describe its average comparison count up to lower order terms. This means that we do not have to care about things like the way pointers move through the input array to analyze the average comparison count of a dual-pivot quicksort algorithm. Next, we will show that there exist natural comparison-optimal dual-pivot quicksort algorithms, i. e., algorithms which achieve the minimum possible average comparison count. To do this, we will develop a lower bound for the average comparison count of dual-pivot quicksort. We will extend our theory to k-pivot quicksort, for $k \geq 3$. This will allow us to compare multi-pivot quicksort with other standard variants such as classical quicksort using the median in a sample of size $2k' + 1, k' \geq 0$, as pivot. (We will refer to this algorithm as "median-of-k'"). It will turn out that the improvements in the average comparison count when using comparison-optimal k-pivot quicksort algorithms can be achieved in much simpler ways, e. g., by using the median-of-k strategy. The algorithmic subproblems which have to be solved by optimal k-pivot quicksort algorithms will let us conclude that no practical improvements are to be expected from using such variants with more than two pivots.

However, there could be other advantages of multi-pivot quicksort apart from a lower average comparison count. In [Kus+14], Kushagra, López-Ortiz, Qiao, and Munro described a beautiful 3-pivot quicksort algorithm that was faster than Yaroslavskiy's algorithm in their experiments. Their algorithm makes much more comparisons on average than a comparison-optimal 3-pivot quicksort algorithm would make, but has a much simpler implementation. The authors of [Kus+14] conjectured that the improvements of multi-pivot quicksort are due to better cache behavior. (They provided experimental results to back this thesis in [LO14].) We will provide a theoretical study of partitioning algorithms, i. e., algorithms that solve the problem of partitioning the input with respect to the pivots. One of these algorithms will generalize the partitioning algorithm for classical quicksort. We will see that using more than one pivot makes it possible to decrease the average number of memory accesses to the input, which directly translates into better cache behavior. Another partitioning algorithm uses a two-pass approach and minimizes both the average number of element rearrangements and the average number of memory accesses to the input when used with many, e. g., 127 pivots. At the end, we will report on results of a large-scale study on the empirical running time of multi-pivot quicksort algorithms. When no additional space is allowed, variants using two or three pivots provide the best running time. If additional space can be allocated, significant improvements are possible by using 127 pivots.

The second part of this thesis considers the use of hash functions in algorithms and data structures. For a finite set U ("the universe") and a finite set R ("the range"), a hash function is a function mapping elements from U to R. Hash functions are used in many application: distributing keys to table cells in *hash tables*, distributing jobs among machines in *load balancing*,

and gathering statistics in *data streams*, to name just a few.

In the analysis of a hashing-based algorithm or data structure, the hash function is traditionally assumed to be "ideal", i. e., the mapping is fully random, it consumes no space, and its evaluation takes unit time. Such functions do not exist. In fact, the representation of a fully random hash functions takes $|U| \log |R|$ bits, which is inefficient since in hashing the universe U is assumed to be huge. Consequently, a large body of work has considered explicit, efficient hash functions, which are not fully random, but just good enough to allow running a specific application. The goal of the second part of this thesis is to detail exactly such a construction and show its use in different applications.

Traditionally, explicit hash function constructions build upon the work of Carter and Wegman [CW79]. They proposed a technique called *universal hashing*, in which the idea is to pick a hash function randomly from a set $\mathcal{H} \subseteq \{h \mid h\colon U \to R\}$. (We call such a set $\mathcal{H}$ a *hash family* or *hash class*.) They coined the notions of "universality" and "independence" of such sets $\mathcal{H}$ (to be defined rigorously in the respective part of this thesis). Both results mean the hash function behaves close to a fully random hash function with respect to the collision probability of two distinct elements or with respect to full randomness on small key sets of the universe. These two concepts were (and still are) central in the analysis of hashing-based algorithms. As examples, we mention the groundbreaking results of Alon, Matias, and Szegedy, who showed in [AMS99] that 4-wise independence suffices for frequency estimation, and Pagh, Pagh, and Ruciz [PPR09], who proved, only in 2009, that 5-wise independence suffices for running linear probing, the most often used hash table implementation. Finding a proof that a certain degree of independence allows running a specific application has the advantage that one can choose freely from the pool of available hash families that achieve the necessary degree of independence. If a faster hash family becomes known in future research, one can just switch to use this hash class.

In a different line of research, explicit properties of a hash class beyond its "universality" and "independence" were exploited to show that specific hash functions suffice to run a certain application with provable guarantees. Here, examples are the papers of Dietzfelbinger and Meyer auf der Heide [DM90] (dynamic hashing), Karp, Luby, and Meyer auf der Heide [KLM96] (PRAM simulations), Dietzfelbinger and Woelfel [DW03] (cuckoo hashing, uniform hashing, shared memory simulations) and Woelfel [Woe06a] (load balancing). In 2010, Pătraşcu and Thorup showed in [PT11] that a class of very simple tabulation hash functions allows running many important applications such as linear probing, static cuckoo hashing, frequency estimation and ε-minwise independent hashing. The same authors described in [PT13] a more involved tabulation class allowing for Chernoff-type bounds which guarantees robust execution times for a sequence of operations in linear probing and chained hashing. Currently, there is a lot of ongoing research devoted to the demonstration that explicit hash function constructions allow running certain applications. For tabulation-based hashing this is demonstrated by the recent papers of Dahlgaard and Thorup [DT14] and Dahlgaard, Knudsen, Rotenberg and Thorup [Dah+14]. A different hash class was presented by Celis, Reingold, Segev, and Wieder in [Cel+13]. They proved that their construction has strong randomness properties in the classical situation of throwing n balls into n bins. In [RRW14], Reingold, Rothblum, and Wieder showed that this hash class allows running

a modified version of cuckoo hashing (with a stash) and load balancing using two hash functions.

This thesis considers a generalization of the hash class described by Dietzfelbinger and Woelfel in [DW03]. We will prove that the hash class provides sufficient randomness properties to run many different applications, such as cuckoo hashing with a stash as introduced by Kirsch, Mitzenmacher, and Wieder [KMW09], the construction of a perfect hash function as described by Botelho, Pagh, and Ziviani [BPZ13], the simulation of a uniform hash function due to Pagh and Pagh [PP08], generalized cuckoo hashing as described by Fotakis, Pagh, Sanders, and Spirakis [Fot+05] in a sparse setting with two recent insertion algorithms introduced by Khosla [Kho13] and Eppstein, Goodrich, Mitzenmacher, and Pszona [Epp+14], and many different algorithms for load balancing as studied by Schickinger and Steger in [SS00]. The main contribution is a unified framework based on the first moment method. This framework allows us to analyze a hashing-based algorithm or data structure without exploiting details of the hash function. While our construction is more involved as the simple tabulation scheme of Pătraşcu and Thorup from [PT11], we show in experiments it is indeed practical.

How to Read This Thesis. This thesis consists of two independent parts. These two parts can be read independently of each other. Each part will contain its own introduction and conclusion with pointers to future work. Part 1 will explain the work on multi-pivot quicksort. There, Sections 3–5 (dual-pivot quicksort), Section 6 (multi-pivot quicksort), and Section 7 (additional cost measures for multi-pivot quicksort) do not dependent on each other. Part 2 will describe the explicit construction of a class of simple hash functions and its analysis. Section 11 presents the basic framework and is mandatory to understand the subsequent sections. Section 12 and Section 13 can be read independently of Section 14.

Publications. The first part of this thesis draws some content from the following published or submitted material:

- *"Optimal Partitioning For Dual Pivot Quicksort"*, Martin A., Martin Dietzfelbinger, appeared in *ICALP '13* [AD13]. *Full version of this paper submitted to ACM Transactions on Algorithms.*

The second part of this thesis builds upon a manuscript of Woelfel [Woe05]. Furthermore, it includes some parts of the following published material:

- *"Simple Hash Functions Suffice for Cuckoo Hashing with a Stash"*, Martin A., Martin Dietzfelbinger, Philipp Woelfel, in *ESA '12* [ADW12]. Full version [ADW14] to appear in *Algorithmica 70* (2014) as part of an issue on selected papers from *ESA '12*.

Notation. We fix some notation that we are going to use throughout this thesis. For $m \in \mathbb{N}$, we let $[m] := \{0, \ldots, m-1\}$. We assume the reader is familiar with basics of discrete probability theory. A good treatment of the topic with respect to this thesis are the books [MR95; MU05]. In this work, probabilities are denoted by $\Pr$, expectations are denoted by E. When considering

different probability spaces, we add the probability space as a subscript to $\Pr$ or E. Random variables will always be referred to by an uppercase character. Events considered in this work often depend on an integer n. If an event E_n occurs with probability at least $1 - O(1/n^\alpha)$, for some constant $\alpha > 0$, we say that E_n occurs *with high probability*, often abbreviated by "w.h.p.".

Experimental Setup. All experiments were carried out on an Intel i7-2600 (4 physical cores, 3.4 GHz, 32 KB L1 instruction cache, 32 KB L1 data cache, 256 KB L2 cache and 8 MB L3 cache) with 16 GB RAM running Ubuntu 13.10 with kernel version 3.11.0. The C++ source code and the measurements from experiments can be accessed via the web page that accompanies this thesis. It is available at `http://eiche.theoinf.tu-ilmenau.de/maumueller-diss/`.

Part I | Multi-Pivot Quicksort

2. Introduction

Quicksort [Hoa62] is a thoroughly analyzed classical sorting algorithm, described in standard textbooks such as [Cor+09; Knu73; SF96] and with implementations in practically all algorithm libraries. Following the divide-and-conquer paradigm, on an input consisting of n elements quicksort uses a pivot element to partition its input elements into two parts, the elements in one part being smaller than or equal to the pivot, the elements in the other part being larger than or equal to the pivot, and then uses recursion to sort these parts. It is well known that if the input consists of n elements with distinct keys in random order and the pivot is picked by just choosing an element, then on average quicksort uses $2n \ln n + O(n)$ comparisons between elements from the input. (Pseudocode of a standard implementation of quicksort can be found in Appendix A.)

In 2009, Yaroslavskiy announced[1] that he had found an improved quicksort implementation, the claim being backed by experiments. After extensive empirical studies, in 2009 Yaroslavskiy's algorithm became the new standard quicksort algorithm in Oracle's Java 7 runtime library. This algorithm employs two pivots to split the elements. If two pivots p and q with $p \leq q$ are used, the partitioning step partitions the remaining $n - 2$ elements into three parts: elements smaller than or equal to p, elements between p and q, and elements larger than or equal to q, see Fig. 2.1. (In accordance with tradition, we assume in this theoretical study that all elements have different keys. Of course, in implementations equal keys are an important issue that requires a lot of care [Sed77].) Recursion is then applied to the three parts. As remarked in [WN12], it came as a surprise that two pivots should help, since in his thesis [Sed75] Sedgewick had proposed and analyzed a dual-pivot approach inferior to classical quicksort. Later, Hennequin in his thesis [Hen91] studied the general approach of using $k \geq 1$ pivot elements. According to [WN12], he found only slight improvements which would not compensate for the more involved partitioning procedure.

In [WN12] (full version [WNN13]), Nebel and Wild formulated and thoroughly analyzed a simplified version of Yaroslavskiy's algorithm. They showed that it makes $1.9n \ln n + O(n)$ key comparisons on average, in contrast to the $2n \ln n + O(n)$ of standard quicksort and the $\frac{32}{15} n \ln n + O(n)$ of Sedgewick's dual-pivot algorithm. On the other hand, Yaroslavskiy's algorithm requires $0.6n \ln n + O(n)$ swap operations on average, which is much higher than the $0.33n \ln n + O(n)$ swap operations in classical quicksort. As an important future research direction, they proposed to explain how Yaroslavskiy's algorithm can compensate for the large number of extra swaps it makes.

[1] An archived version of the relevant discussion in a Java newsgroup can be found at `http://permalink.gmane.org/gmane.comp.java.openjdk.core-libs.devel/2628`. Also see [WN12].

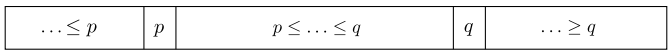

Figure 2.1.: Result of the partition step in dual-pivot quicksort schemes using two pivots p, q with $p \leq q$. Elements left of p are smaller than or equal to p, elements right of q are larger than or equal to q. The elements between p and q are at least as large as p and at most as large as q.

In the discussion referenced in Footnote 1, Jon Bentley, one of the authors of the seminal paper [BM93] describing engineering steps for a sorting algorithm used in programming libraries, is quoted as saying:

> *It would be horrible to put the new code [Yaroslavskiy's algorithm] into the library, and then have someone else come along and speed it up by another 20% by using standard techniques.*

This thesis considers what is possible when using more than one pivot and whether or not improvements beyond Yaroslavskiy's algorithm are to be expected from multi-pivot quicksort algorithms.

In the first part of our study on multi-pivot quicksort algorithms, we will detail the design choices we have for developing a dual-pivot quicksort algorithm that, on average, makes as few key comparisons as possible. Let us take some time to understand the general idea.

The first observation is that everything depends on the cost, i. e., the comparison count, of the partitioning step. This is not new at all. Actually, in Hennequin's thesis [Hen91] the connection between partitioning cost and overall cost for quicksort variants with more than one pivot is analyzed in detail. The result relevant for us is that if two pivots are used and the (average) partitioning cost for n elements is $a \cdot n + O(1)$, for a constant a, then the average cost for sorting n elements is

$$\frac{6}{5} a \cdot n \ln n + O(n). \tag{2.1}$$

Throughout this part of the thesis all that interests us is the constant factor with the leading term. (The reader should be warned that for real-life n the linear term, which may even be negative, can have a big influence on the average number of comparisons. We shall see that this is indeed the case in the empirical verification.)

The second observation is that the partitioning cost depends on certain details of the partitioning procedure. This is in contrast to standard quicksort with one pivot where partitioning always takes $n - 1$ comparisons. In [WN12] it is shown that Yaroslavskiy's partitioning procedure uses $\frac{19}{12}n + O(1)$ comparisons on average, while Sedgewick's uses $\frac{16}{9}n + O(1)$ many. The analysis of these two algorithms is based on the study of how certain pointers move through the array, at which positions elements are compared to the pivots, which of the two pivots is used for the first comparison, and how swap operations exchange two elements in the array. For understanding what is going on, however, it is helpful to forget about concrete implementations with loops in

which pointers sweep across arrays and entries are swapped, and look at partitioning with two pivots in a more abstract way. For simplicity, we shall always assume the input to be a permutation of $\{1, \dots, n\}$. Now pivots p and q with $p < q$ are chosen. The task is to *classify* the remaining $n - 2$ elements into classes "small" ($s = p - 1$ many), "medium" ($m = q - p - 1$ many), and "large" ($\ell = n - p$ many), by comparing these elements one after the other with the smaller pivot or the larger pivot, or both of them if necessary. Note that for symmetry reasons it is inessential in which order the elements are treated. The only choice the algorithm can make is whether to compare the current element with the smaller pivot or the larger pivot first. Let the random variable S_2 denote the number of small elements compared with the larger pivot first, and let L_2 denote the number of large elements compared with the smaller pivot first. Then, the total number of comparisons is $n - 2 + m + S_2 + L_2$.

Averaging over all inputs and all possible choices of the pivots the term $n - 2 + m$ will lead to $\frac{4}{3}n + O(1)$ key comparisons on average, independently of the algorithm. Let $W = S_2 + L_2$ be the number of elements compared with the "wrong" pivot first. Then $\mathrm{E}(W)$ is the only quantity influenced by a particular partitioning procedure.

In this thesis, we will first devise an easy method to calculate $\mathrm{E}(W)$. The result of this analysis will lead to an (asymptotically) optimal strategy. The basic approach is the following. Assume a partitioning procedure is given, and assume p, q and hence $s = p - 1$ and $\ell = n - q$ are fixed, and let $w_{s,\ell} = \mathrm{E}(W \mid s, \ell)$. Denote the average number of elements compared to the smaller and larger pivot first by $f^{\mathrm{p}}_{s,\ell}$ and $f^{\mathrm{q}}_{s,\ell}$, respectively. If the elements to be classified were chosen to be small, medium, and large independently with probabilities $s/(n-2)$, $m/(n-2)$, and $\ell/(n-2)$, resp., then the average number of small elements compared with the large pivot first would be $f^{\mathrm{q}}_{s,\ell} \cdot s/(n-2)$, similarly for the large elements. Of course, the actual input is a sequence with exactly s, m, and ℓ small, medium, and large elements, respectively, and there is no independence. Still, we will show that the randomness in the order is sufficient to guarantee that

$$w_{s,\ell} = f^{\mathrm{q}}_{s,\ell} \cdot s/n + f^{\mathrm{p}}_{s,\ell} \cdot \ell/n + o(n). \tag{2.2}$$

The details of the partitioning procedure will determine $f^{\mathrm{p}}_{s,\ell}$ and $f^{\mathrm{q}}_{s,\ell}$, and hence $w_{s,\ell}$ up to $o(n)$.

This seemingly simple insight has two consequences, one for the analysis and one for the design of dual-pivot algorithms:

(i) In order to *analyze* the average comparison count of a dual-pivot algorithm (given by its partitioning procedure) up to lower order terms, determine $f^{\mathrm{p}}_{s,\ell}$ and $f^{\mathrm{q}}_{s,\ell}$ for this partitioning procedure. This will give $w_{s,\ell}$ up to lower order terms, which must then be averaged over all s, ℓ to find the average number of comparisons in partitioning. Then apply (2.1).

(ii) In order to *design* a good partitioning procedure w.r.t. the average comparison count, try to make $f^{\mathrm{q}}_{s,\ell} \cdot s/n + f^{\mathrm{p}}_{s,\ell} \cdot \ell/n$ small.

We shall demonstrate both approaches in Section 4. An example: As explained in [WN12], if s and ℓ are fixed, in Yaroslavskiy's algorithm we have $f^{\mathrm{q}}_{s,\ell} \approx \ell$ and $f^{\mathrm{p}}_{s,\ell} \approx s + m$. By (2.2) we get

$w_{s,\ell} = (\ell s + (s+m)\ell)/n + o(n)$. This must be averaged over all possible values of s and ℓ. The result is $\frac{1}{4}n + o(n)$, which together with $\frac{4}{3}n + O(1)$ gives $\frac{19}{12}n + o(n)$, close to the result from [WN12].

Principle (ii) will be used to identify an (asymptotically) optimal partitioning procedure that makes $1.8n \ln n + o(n \ln n)$ key comparisons on average. In brief, such a strategy should achieve the following: If $s > \ell$, compare (almost) all entries with the smaller pivot first ($f^{\mathrm{p}}_{s,\ell} \approx n$ and $f^{\mathrm{q}}_{s,\ell} \approx 0$), otherwise compare (almost) all entries with the larger pivot first ($f^{\mathrm{p}}_{s,\ell} \approx 0$ and $f^{\mathrm{q}}_{s,\ell} \approx n$). Of course, some details have to be worked out: How can the algorithm decide which case applies? In which technical sense is this strategy optimal? These questions will be answered in Section 4.2.

Following our study on dual-pivot quicksort, we will consider the case of using k pivots $p_1, \dots, p_k$ in a quicksort algorithm, for $k \geq 1$. We shall see that the model for dual-pivot quicksort algorithms extends nicely to this general situation. First, instead of having "small", "medium", and "large" elements, there are $k+1$ different groups $\mathrm{A}_0, \dots, \mathrm{A}_k$. An element x belongs to group $\mathrm{A}_i, 0 \leq i \leq k$, if $p_i < x < p_{i+1}$. (For ease of discussion, we set $p_0 = 0$ and $p_{k+1} = n+1$.) The classification of a single element becomes more involved when at least three pivots are used. Naturally, it is done by comparing the element against the pivots in some order. This order is best visualized using a *comparison tree*, which is a binary tree with $k+1$ leaves labeled $\mathrm{A}_0, \dots, \mathrm{A}_k$ from left to right and k inner nodes labeled $\mathrm{p}_1, \dots, \mathrm{p}_k$ according to inorder traversal. The classification of an element can then be read off from the leaf that is reached in the obvious way. The design choice of a multi-pivot quicksort algorithm for classifying an element is then to pick a certain pivot order, i. e., a certain comparison tree. To find out how many key comparisons a multi-pivot quicksort algorithm makes on average it suffices to multiply the average number of times a certain comparison tree is used with a certain cost term which describes how many comparisons a comparison tree will require on average for a fixed pivot choice, summed up over all comparison trees and pivot choices.

In implementations of quicksort, the pivot is usually chosen as the median from a small sample of $2k+1$ elements with $k \geq 0$. To speed up the selection of the pivot, other strategies such as the "quasi-median-of-nine", i. e., the median of three medians of samples of size 3, have been suggested [BM93]. Intuitively, this yields more balanced (and thus fewer) subproblems. This idea already appeared in Hoare's original publication [Hoa62] without an analysis, which was later supplied by van Emden [Emd70]. The complete analysis of this variant was given by Martínez and Roura in [MR01] in 2001. They showed that the optimal sample size is $\Theta(\sqrt{n})$. For this sample size the average comparison count of quicksort matches the lower-order bound of $n \log n + O(n)$ comparisons. In practice, one usually uses a sample of size 3. Theoretically, this decreases the average comparison count from $2n \ln n + O(n)$ to $1.714n \ln n + O(n)$. We will see that choosing the median of a sample of k elements yields about the same improvement to the average comparison count as using k pivots in a comparison-optimal multi-pivot quicksort algorithm.

It seems hard to believe that key comparisons are the single dominating factor to the running

time of a sorting algorithm. This is especially true when key comparisons are cheap, e. g., for comparing 32-bit integers. Two important performance bottlenecks in modern computers are *branch mispredictions* and *cache behavior* [HP12]. In very recent work, Martínez, Nebel, and Wild [MNW15] analyzed branch mispredictions in classical quicksort and Yaroslavksiy's algorithm. According to their paper, the running time differences observed in practice cannot be explained by this cost measure.

Consequently, we have to consider the cost of rearranging the input elements using partitioning algorithms for multi-pivot quicksort in Section 7. This section draws some ideas from the fast three-pivot quicksort algorithm of Kushagra *et al.* [Kus+14]. The classification strategy of their algorithm is very simple. It compares a new element with the middle pivot first, and then with one of the two others. While the general idea of this algorithm had been known (see, e.g., [Hen91; Tan93]), they provided a smart way of moving elements around to produce the partition. Building upon the work of Ladner and LaMarca [LL99], they demonstrated that their algorithm is very cache efficient. Hence, they conjectured that the observed running time behavior is largely due to cache-efficiency, and not primarily influenced by comparisons or swaps. We will extend their study to partitioning algorithms for multi-pivot quicksort.

In which sense can the cache-efficiency of classical quicksort be improved? It is often assumed that the standard partitioning procedure of quicksort, in which two pointers move towards each other and exchange misplaced elements along the way, see Algorithm A.1 in Appendix A, is "optimal" with respect to cache-efficiency. There are only two places in the array which are needed in memory at any given point in time and it is easy to predict the array segments that are going to be used next. This makes prefetching of these array segments easy. To improve cache behavior, we have to consider the whole sorting process. Intuitively, using more than one pivot decreases the size of subproblems and thus reduces the depth of the recursion stack. Since we have to read about the whole array on each level of the recursion, reduced depth means the input has to be read fewer times. On the other hand, using more than one pivot increases the number of elements that have to be exchanged because they are at a wrong position in the input. This makes partitioning more complicated. So, using more than one pivot yields two effects working in opposite directions: increased cost by more complicated partitioning, and decreased cost because the input has to be read fewer times. It will turn out that, in some sense, minimal partitioning cost will be achieved when using five pivots.

Moreover, we will consider partitioning strategies that decouple classification and partitioning, i. e., use two passes to produce a partition. This technique was pioneered by Sanders and Winkel with their "super scalar sample sort algorithm" [SW04]. This approach will prove to have much better cache behavior because partitioning does not become more difficult with many pivots. However, additional space is needed to obtain algorithms that are faster in practice than algorithms using only one pass.

Summary and Outline. We will study multi-pivot quicksort algorithms and show how well they perform with respect to different cost measures.

In Section 3, we study the average comparison count of dual-pivot quicksort algorithms. To this end, we introduce a model which covers dual-pivot quicksort algorithms. Then, we describe how to calculate the average comparison count of a given algorithm. Next, we use this result to re-analyze previously known dual-pivot quicksort algorithms like Yaroslavskiy's algorithm and Sedgewick's algorithm in Section 4. In the same section, we present optimal algorithms, i. e., algorithms which achieve the minimum possible average comparison count for dual-pivot quicksort. Optimal dual-pivot quicksort algorithms make $1.8n \ln n + O(n)$ comparisons on average, improving on the $1.9n \ln n + O(n)$ comparisons Yaroslavskiy's algorithm makes on average. In the subsequent section, we consider the well known technique of choosing pivots from a small sample. We prove that choosing the tertiles of the sample as the two pivots, as, e. g., done in the Java implementation of Yaroslavskiy's algorithm, is not optimal for dual-pivot quicksort, and describe optimal sampling strategies.

After understanding the case with two pivots, we consider quicksort algorithms using more than two pivots in Section 6. We are going to see that our theory for dual-pivot quicksort generalizes nicely to this case. Again, we describe how to calculate the average comparison count for an arbitrary multi-pivot quicksort strategy and how comparison-optimal algorithms for dual-pivot quicksort extend to the case of using more than two pivots. As we will find out, calculating the average comparison count is hard even for a few, say, four pivots. From a practical perspective, comparison-optimal multi-pivot quicksort will turn out to be slow and not competitive.

Consequently, we will follow a different approach in Section 7. We restrict ourselves to use some fixed comparison tree for each classification—ignoring the average comparison count—, and think only about moving elements around in order to produce the partition. This will help to understand in which sense a multi-pivot quicksort approach allows more efficient algorithms than classical quicksort.

Finally, Section 8 reports on a study of empirical running times of different quicksort algorithms. We shall see that many variants are faster than classical quicksort. Furthermore, we will investigate whether or not our theoretical cost measures help predicting observed running times. In brief, the cache-efficiency of an algorithm provides the best prediction for differences in running times. However, it cannot explain observed running times in detail.

3. Basic Approach to Analyzing Dual-Pivot Quicksort

In this section we formalize the notion of a "dual-pivot quicksort algorithm", give the basic assumptions of the analysis and show how to calculate the average comparison count of an arbitrary dual-pivot quicksort algorithm.

In Section 6 we will generalize this approach to "k-pivot quicksort", for $k \geq 1$. Of course, this generalization includes the dual-pivot quicksort case. However, the algorithmic approach to dual-pivot quicksort is much easier to understand. Furthermore, we are able to prove some tight results analytically only in the dual-pivot quicksort case.

3.1. Basic Setup

We assume the input sequence $(a_1, \ldots, a_n)$ to be a random permutation of $\{1, \ldots, n\}$, each permutation occurring with probability $(1/n!)$. If $n \leq 1$, there is nothing to do; if $n = 2$, sort by one comparison. Otherwise, choose the first element a_1 and the last element a_n as the set of pivots, and set $p = \min(a_1, a_n)$ and $q = \max(a_1, a_n)$. Partition the remaining elements into elements smaller than p ("small" elements), elements between p and q ("medium" elements), and elements larger than q ("large" elements), see Fig. 2.1. Then apply the procedure recursively to these three groups. Clearly, each pair p, q with $1 \leq p < q \leq n$ appears as set of pivots with probability $1/\binom{n}{2}$. Our cost measure is the number of key comparisons needed to sort the given input. Let C_n be the random variable counting this number. Let P_n denote the number of key comparisons necessary to partition the $n-2$ non-pivot elements into the three groups, and let R_n denote the number of key comparisons made in the recursion. Since elements are only compared with the two pivots, the randomness of subarrays is preserved. Thus, in the recursion we may always assume that the input is arranged randomly. The average number of key comparisons

$\mathrm{E}(C_n)$ obeys the following recurrence:

$$\begin{aligned}\mathrm{E}(C_n) &= \sum_{1\le p<q\le n} \Pr(p,q \text{ pivots}) \cdot \mathrm{E}(P_n + R_n \mid p,q) \\ &= \sum_{1\le p<q\le n} \frac{2}{n(n-1)} \cdot \mathrm{E}(P_n + C_{p-1} + C_{q-p-1} + C_{n-q} \mid p,q) \\ &= \mathrm{E}(P_n) + \frac{2}{n(n-1)} \cdot 3 \sum_{k=0}^{n-2} (n-k-1) \cdot \mathrm{E}(C_k). \end{aligned} \tag{3.1}$$

Solving the Recurrence for Dual-Pivot Quicksort. We now solve this recurrence using the Continuous Master Theorem of Roura [Rou01], whose statement we will review first.

Theorem 3.1.1 ***([Rou01, Theorem 18])***
Let F_n be recursively defined by

$$F_n = \begin{cases} b_n, & \text{for } 0 \le n < N, \\ t_n + \sum_{j=0}^{n-1} w_{n,j} F_j, & \text{for } n \ge N, \end{cases}$$

where the toll function t_n satisfies $t_n \sim K n^\alpha \log^\beta(n)$ as $n \to \infty$ for constants $K \neq 0, \alpha \ge 0, \beta > -1$. Assume there exists a function $w\colon [0,1] \to \mathbb{R}$ such that

$$\sum_{j=0}^{n-1} \left| w_{n,j} - \int_{j/n}^{(j+1)/n} w(z)\, \mathrm{d}z \right| = O(n^{-d}), \tag{3.2}$$

for a constant $d > 0$. Let $H := 1 - \int_0^1 z^\alpha w(z)\, \mathrm{d}z$. Then we have the following cases:[a]

1. If $H > 0$, then $F_n \sim t_n / H$.
2. If $H = 0$, then $F_n \sim (t_n \ln n)/\hat{H}$, where
$$\hat{H} := -(\beta+1) \int_0^1 z^\alpha \ln(z) w(z)\, \mathrm{d}z.$$
3. If $H < 0$, then $F_n \sim \Theta(n^c)$ for the unique $c \in \mathbb{R}$ with
$$\int_0^1 z^c w(z)\, \mathrm{d}z = 1.$$

[a] Here, $f(n) \sim g(n)$ means that $\lim_{n\to\infty} f(n)/g(n) = 1$.

Theorem 3.1.2
Let $\mathcal{A}$ be a dual-pivot quicksort algorithm which has for each subarray of length n partitioning cost $\mathrm{E}(P_n) = a \cdot n + o(n)$. Then

$$\mathrm{E}(C_n) = \frac{6}{5} a n \ln n + o(n \ln n). \tag{3.3}$$

Proof. Recurrence (3.1) has weight

$$w_{n,j} = \frac{6(n-j-1)}{n(n-1)}$$

We define the shape function $w(z)$ as suggested in [Rou01] by

$$w(z) = \lim_{n\to\infty} n \cdot w_{n,zn} = 6(1-z).$$

Now we have to check (3.2) to see whether the shape function is suitable. We calculate:

$$\begin{aligned}
&\sum_{j=0}^{n-1} \left| w_{n,j} - \int_{j/n}^{(j+1)/n} w(z)\,\mathrm{d}z \right| \\
&= 6 \sum_{j=0}^{n-1} \left| \frac{n-j-1}{n(n-1)} - \int_{j/n}^{(j+1)/n} (1-z)\,\mathrm{d}z \right| \\
&= 6 \sum_{j=0}^{n-1} \left| \frac{n-j-1}{n(n-1)} + \frac{2j+1}{2n^2} - \frac{1}{n} \right| \\
&< 6 \sum_{j=0}^{n-1} \left| \frac{1}{2n(n-1)} \right| = O(1/n).
\end{aligned}$$

Thus, w is a suitable shape function. By calculating

$$H := 1 - 6 \int_0^1 (z - z^2)\,\mathrm{d}z = 0,$$

we conclude that the second case of Theorem 3.1.1 applies for our recurrence. Consequently, we calculate

$$\hat{H} := -6 \int_0^1 (z - z^2) \ln z\,\mathrm{d}z,$$

which—using a standard computer algebra system—gives $\hat{H} = 5/6$. The theorem follows. □

This generalizes the result of Hennequin [Hen91] who proved that for average partitioning cost $a \cdot n + O(1)$ for n elements, for a constant a, the average cost for sorting n elements is

$$\frac{6}{5} a \cdot n \ln n + O(n). \tag{3.4}$$

Remark 3.1.3. Most of our algorithms have partitioning cost $a \cdot n + o(n)$, for a constant a. Thus, we cannot apply (3.4) directly. In the paper [AD13] we give an alternative proof of Theorem 3.1.2 only based on (3.4), see [AD13, Theorem 1].

Handling Small Subarrays. One of our algorithms will make a decision based on a small sampling step. For very small subarrays of size $n_0 \leq n^{1/\ln \ln n}$, this decision will be wrong with a too high probability, making the partitioning cost larger than $a \cdot n' + o(n')$. We will now argue that the total contribution of these small subarrays to the average comparison count is $o(n \ln n)$, and can hence be neglected.

To see this, wait until the algorithm has created a subarray of size $n' < n_0$. Note that the partitioning cost of dual-pivot quicksort on input size n' is at most $2n'$. Using this simple observation and combining it with (3.3), the cost for the whole recursion starting from this input is at most $12/5 \cdot n' \ln n' + o(n' \ln n')$. To calculate the total contribution of all small subarrays we must then sum $12/5 \cdot n_i \ln n_i + o(n_i \ln n_i)$ over a sequence of disjoint subarrays of length $n_1, \ldots, n_k$. Since all n_i are smaller than n_0, $n_1 + \ldots + n_k \leq n$, and since $x \mapsto x \ln x$ is a convex function, this sums up to no more than $\frac{n}{n_0} \cdot \frac{12}{5} n_0 \ln n_0 + \frac{n}{n_0} \cdot o(n_0 \ln n_0) = o(n \ln n)$.

Thus, in the remainder of this work we will ignore the contribution of such small subarrays to the total sorting cost.

3.2. Analysis of a Partitioning Step

The main consequence of Theorem 3.1.2 is that it is sufficient to study the cost of partitioning.

Abstracting from moving elements around in arrays, we arrive at the following "classification problem": Given a random permutation $(a_1, \ldots, a_n)$ of $\{1, \ldots, n\}$ as the input sequence and a_1 and a_n as the two pivots p and q, with $p < q$, classify each of the remaining $n - 2$ elements as being small, medium, or large. Note that there are exactly $s := p - 1$ small elements, $m := q - p - 1$ medium elements, and $\ell := n - q$ large elements. Although this classification does not yield an actual partition of the input sequence, a classification algorithm can be turned into a partition algorithm only by rearranging the input elements after classification, without additional key comparisons.

We make the following observations (and fix notation) for all classification algorithms. One key comparison is needed to decide which of the elements a_1 and a_n is the smaller pivot p and which is the larger pivot q. For classification, each of the remaining $n - 2$ elements has to be compared against p or q or both. Each *medium* element has to be compared to p *and* q. On average, there are $(n-2)/3$ medium elements. Let S_2 denote the number of small elements that

are compared to the larger pivot first, i. e., the number of small elements that need 2 comparisons for classification. Analogously, let L_2 denote the number of large elements compared to the smaller pivot first. Conditioning on the pivot choices, and hence the values of s and ℓ, we may calculate $\mathrm{E}(P_n)$ as follows:

$$\mathrm{E}(P_n) = (n-1) + (n-2)/3 + \frac{1}{\binom{n}{2}} \sum_{s+\ell \le n-2} \mathrm{E}(S_2 + L_2 \mid s, \ell). \tag{3.5}$$

Here, "s, ℓ" denotes the event that the pivots $p = s + 1$ and $q = n - \ell$ are chosen. We call the third summand the *additional cost term (ACT)*, as it is the only value that depends on the actual classification algorithm.

3.2.1. Analysis of the Additional Cost Term

We will use the following formalization of a partitioning procedure: A *classification strategy* is given as a three-way decision tree T with a root and $n-2$ levels of inner nodes as well as one leaf level. The root is on level 0. Each node v is labeled with an index $i(v) \in \{2, \dots, n-1\}$ and an element $l(v) \in \{\mathrm{p}, \mathrm{q}\}$. If $l(v)$ is p, then at node v element $a_{i(v)}$ is compared with the smaller pivot first; otherwise, i. e., $l(v) = \mathrm{q}$, it is compared with the larger pivot first. The three edges out of a node are labeled σ, μ, λ, resp., representing the outcome of the classification as small, medium, large, respectively. The label of edge e is called $c(e)$. The three children of a node v are called the σ-, μ-, and λ-child of this node. On each of the 3^{n-2} paths each index occurs exactly once. Each input determines exactly one path w from the root to a leaf in the obvious way; the classification of the elements can then be read off from the node and edge labels along this path. The labeled reached in this way contains this classification. We call such a tree a *classification tree*.

Identifying a path π from the root to a leaf lf by the sequence of nodes and edges on it, i. e., $\pi = (v_1, e_1, v_2, e_2, \dots, v_{n-2}, e_{n-2}, \mathrm{lf})$, we define the cost c_π as

$$c_\pi = \big|\{j \in \{1, \dots, n-2\} \mid c(e_j) \neq \mu, l(v_j) \neq c(e_j)\}\big|.$$

For a given input, the cost of the path associated with this input exactly describes the number of additional comparisons on this input. An example for such a classification tree is given in Figure 3.1.

For a random input, we let S_2^T [L_2^T] denote the random variable that counts the number of small [large] elements classified in nodes with label q [p]. We now describe how we can calculate the ACT of a classification tree T. First consider fixed s and ℓ and let the input excepting the pivots be arranged randomly. For a node v in T, we let s_v, m_v, and ℓ_v, resp., denote the number of edges labeled σ, μ, and λ, resp., from the root to v. By the randomness of the input, the probability that the element classified at v is "small", i. e., that the edge labeled σ is used, is exactly $(s - s_v)/(n - 2 - \mathrm{level}(v))$. The probability that it is "medium" is $(m - m_v)/(n - 2 - \mathrm{level}(v))$,

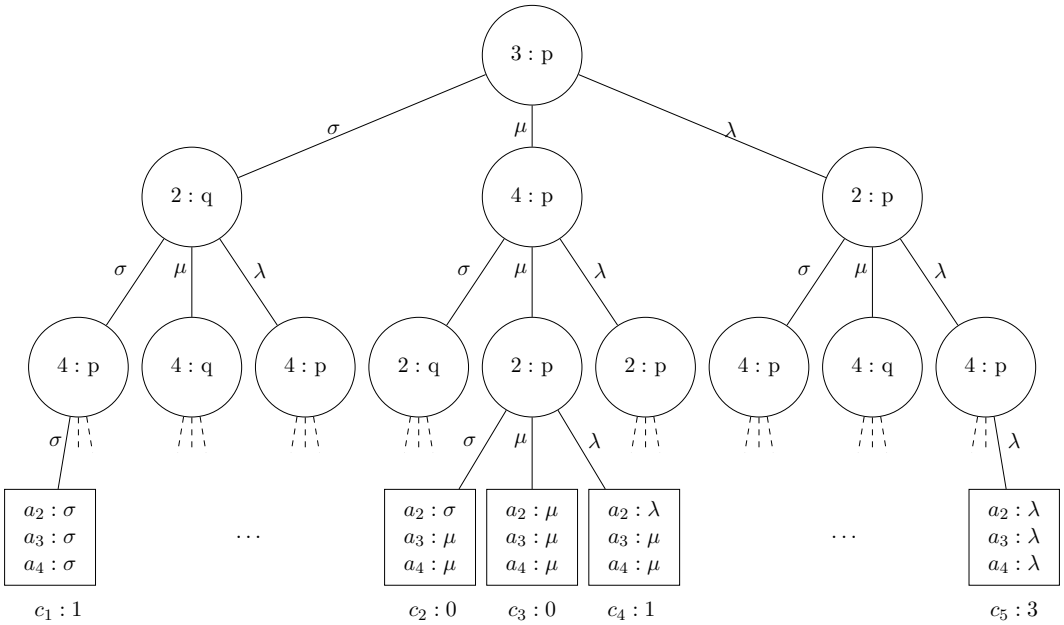

Figure 3.1.: An example for a decision tree to classify three elements a_2, a_3, and a_4 according to the pivots a_1 and a_5. Five out of the 27 leaves are explicitly drawn, showing the classification of the elements and the costs c_i of the specific paths.

and that it is "large" is $(\ell - \ell_v)/(n - 2 - \text{level}(v))$. The probability $p^v_{s,\ell}$ that node v in the tree is reached is then just the product of all these edge probabilities on the unique path from the root to v. The probability that the edge labeled σ out of a node v is used can then be calculated as $p^v_{s,\ell} \cdot (s - s_v)/(n - 2 - \text{level}(v))$. Similarly, the probability that the edge labeled λ is used is $p^v_{s,\ell} \cdot (\ell - \ell_v)/(n - 2 - \text{level}(v))$. Note that all this is independent of the actual ordering in which the classification tree inspects the elements. We can thus always assume some fixed ordering and forget about the label $i(v)$ of node v.

By linearity of expectation, we can sum up the contribution to the additional comparison count for each node separately. Thus, we may calculate

$$\mathrm{E}(S_2^T + L_2^T \mid s, \ell) = \sum_{\substack{v \in T \\ l(v) = \mathrm{q}}} p^v_{s,\ell} \cdot \frac{s - s_v}{n - 2 - \text{level}(v)} + \sum_{\substack{v \in T \\ l(v) = \mathrm{p}}} p^v_{s,\ell} \cdot \frac{\ell - \ell_v}{n - 2 - \text{level}(v)}. \tag{3.6}$$

The setup developed so far makes it possible to describe the connection between a classification tree T and its average comparison count in general. Let F^T_p resp. F^T_q be two random variables that denote the number of elements that are compared with the smaller resp. larger pivot first when using T. Then let $f^\mathrm{q}_{s,\ell} = \mathrm{E}\left(F^T_\mathrm{q} \mid s, \ell\right)$ resp. $f^\mathrm{p}_{s,\ell} = \mathrm{E}\left(F^T_\mathrm{p} \mid s, \ell\right)$ denote the average

number of comparisons with the larger resp. smaller pivot first, given s and ℓ. Now, if it was decided in each step by independent random experiments with the correct expectations $s/(n-2)$, $m/(n-2)$, and $\ell/(n-2)$, resp., whether an element is small, medium, or large, it would be clear that for example $f^{\mathrm{q}}_{s,\ell} \cdot s/(n-2)$ is the average number of small elements that are compared with the larger pivot first. We will show that one can indeed use this intuition in the calculation of the average comparison count, excepting that one gets an additional $o(n)$ term due to the elements tested not being independent.

Before we can show this, we first have to introduce the basic probability theoretical argument which will be used throughout the analysis of different lemmas and theorems.

Let s_i, m_i, ℓ_i, resp., be the random variables which counts the number of elements classified as small, medium, and large, resp., in the first i classification steps. Our goal is to show concentration of these random variables. This would be a trivial application of the Chernoff bound if the tests to which group elements belong to were independent. But when pivots are fixed, the probability that the i-th considered element is small depends on s_{i-1} and i. To deal with these dependencies, we will use the following theorem, commonly known as "the method of averaged bounded differences".[1]

Theorem 3.2.1 ***([DP09, Theorem 5.3])***
Let $X_1, \ldots, X_n$ be an arbitrary set of random variables and let f be a function satisfying the property that for each $i \in \{1, \ldots, n\}$ there is a non-negative c_i such that

$$|\mathrm{E}(f \mid X_1, \ldots, X_i) - \mathrm{E}(f \mid X_1, \ldots, X_{i-1})| \leq c_i.$$

Then

$$\Pr(f > \mathrm{E}(f) + t) \leq \exp\left(-\frac{t^2}{2c}\right)$$

and

$$\Pr(f < \mathrm{E}(f) - t) \leq \exp\left(-\frac{t^2}{2c}\right),$$

where

$$c := \sum_{i=1}^{n} c_i^2.$$

[1] We remark that our statement corrects a typo in [DP09, Theorem 5.3] where the bound reads $\exp(-2t^2/c)$ instead of $\exp(-t^2/(2c))$.

Lemma 3.2.2
Let the two pivots p and q be fixed. Let s_i be defined as above. For each i with $1 \leq i \leq n-2$ we have that

$$\Pr\left(|s_i - \mathrm{E}(s_i)| > n^{2/3}\right) \leq 2\exp\left(-n^{1/3}/2\right).$$

Proof. Define the indicator random variable $X_j = [\text{the } j\text{-th element is small}]$. Of course, $s_i = \sum_{1 \leq j \leq i} X_j$. We let

$$c_j := \left|\mathrm{E}\left(s_i \mid X_1, \ldots, X_j\right) - \mathrm{E}\left(s_i \mid X_1, \ldots, X_{j-1}\right)\right|.$$

Using linearity of expectation we may calculate

$$\begin{aligned}
c_j &= \left|\mathrm{E}(s_i \mid X_1, \ldots, X_j) - \mathrm{E}(s_i \mid X_1, \ldots, X_{j-1})\right| \\
&= \left|X_j + \sum_{k=j+1}^{i} \left(\mathrm{E}(X_k \mid X_1, \ldots, X_j) - \mathrm{E}(X_k \mid X_1, \ldots, X_{j-1})\right) - \mathrm{E}(X_j \mid X_1, \ldots, X_{j-1})\right| \\
&= \left|X_j - \frac{s - s_{j-1}}{n-j-1} + (i-j)\left(\frac{s-s_j}{n-j-2} - \frac{s-s_j+X_j}{n-j-1}\right)\right| \\
&= \left|X_j\left(1 - \frac{i-j}{n-j-1}\right) - \frac{s-s_{j-1}}{n-j-1} + \frac{(i-j)(s-s_j)}{(n-j-2)(n-j-1)}\right| \\
&\leq \left|X_j\left(1 - \frac{i-j}{n-j-1}\right) - \frac{s-s_j+X_j}{n-j-1} + \frac{s-s_j}{n-j-1}\right| \\
&= \left|X_j\left(1 - \frac{i-j-1}{n-j-1}\right)\right| \leq 1.
\end{aligned}$$

Applying Theorem 3.2.1 now gives us

$$\Pr(|s_i - \mathrm{E}(s_i)| > n^{2/3}) \leq 2\exp\left(\frac{-n^{4/3}}{2i}\right),$$

which is not larger than $2\exp(-n^{1/3}/2)$. □

Of course, we get analogous results for the random variables m_i and ℓ_i.

This allows us to prove the following lemma.

Lemma 3.2.3
Let T be a classification tree. Let $\mathrm{E}(P_n^T)$ be the average number of key comparisons for classifying an input of n elements using T. Then

$$\mathrm{E}(P_n^T) = \frac{4}{3}n + \frac{1}{\binom{n}{2}\cdot(n-2)} \sum_{s+\ell\le n-2} \left(f^{\mathrm{q}}_{s,\ell}\cdot s + f^{\mathrm{p}}_{s,\ell}\cdot \ell \right) + o(n).$$

Proof. Fix p and q (and thus s, m, and ℓ). We will show that

$$\mathrm{E}(S_2^T + L_2^T \mid s,\ell) = \frac{f^{\mathrm{q}}_{s,\ell}\cdot s + f^{\mathrm{p}}_{s,\ell}\cdot \ell}{n-2} + o(n). \tag{3.7}$$

(The lemma then follows by substituting this into (3.5).)

We call a node v in T *good* if

$$\begin{aligned} l(v) = \mathrm{q} \text{ and } \left| \frac{s}{n-2} - \frac{s-s_v}{n-\mathrm{level}(v)-2} \right| \le \frac{1}{n^{1/12}} \qquad &\text{or} \\ l(v) = \mathrm{p} \text{ and } \left| \frac{\ell}{n-2} - \frac{\ell-\ell_v}{n-\mathrm{level}(v)-2} \right| \le \frac{1}{n^{1/12}}. \qquad & \end{aligned} \tag{3.8}$$

Otherwise we call v *bad*. We first obtain an upper bound. Starting from (3.6), we calculate:

$$\begin{aligned} \mathrm{E}(S_2^T + L_2^T \mid s,\ell) &= \sum_{v\in T, l(v)=\mathrm{q}} p^v_{s,\ell}\cdot \frac{s-s_v}{n-2-\mathrm{level}(v)} + \sum_{v\in T, l(v)=\mathrm{p}} p^v_{s,\ell}\cdot \frac{\ell-\ell_v}{n-2-\mathrm{level}(v)} \\ &= \sum_{v\in T, l(v)=\mathrm{q}} p^v_{s,\ell}\cdot \frac{s}{n-2} + \sum_{v\in T, l(v)=\mathrm{p}} p^v_{s,\ell}\cdot \frac{\ell}{n-2} + \\ &\quad \sum_{v\in T, l(v)=\mathrm{q}} p^v_{s,\ell}\left(\frac{s-s_v}{n-2-\mathrm{level}(v)} - \frac{s}{n-2} \right) + \\ &\quad \sum_{v\in T, l(v)=\mathrm{p}} p^v_{s,\ell}\left(\frac{\ell-\ell_v}{n-2-\mathrm{level}(v)} - \frac{\ell}{n-2} \right) \\ &\le \sum_{v\in T, l(v)=\mathrm{q}} p^v_{s,\ell}\cdot \frac{s}{n-2} + \sum_{v\in T, l(v)=\mathrm{p}} p^v_{s,\ell}\cdot \frac{\ell}{n-2} + \\ &\quad \sum_{\substack{v\in T, l(v)=\mathrm{q} \\ v \text{ good}}} \frac{p^v_{s,\ell}}{n^{1/12}} + \sum_{\substack{v\in T, l(v)=\mathrm{q} \\ v \text{ bad}}} p^v_{s,\ell} + \sum_{\substack{v\in T, l(v)=\mathrm{p} \\ v \text{ good}}} \frac{p^v_{s,\ell}}{n^{1/12}} + \sum_{\substack{v\in T, l(v)=\mathrm{p} \\ v \text{ bad}}} p^v_{s,\ell}, \end{aligned} \tag{3.9}$$

where the last step follows by separating good and bad nodes and using (3.8). (For bad nodes we use that the left-hand side of the inequalities in (3.8) is at most 1.) For the sums in the last line of

(3.9), consider each level of the classification tree separately. Since the probabilities $p^v_{s,\ell}$ for nodes v on the same level sum up to 1, the contribution of the $1/n^{1/12}$ terms is bounded by $O(n^{11/12})$. Using the definition of $f^{\mathrm{q}}_{s,\ell}$ and $f^{\mathrm{p}}_{s,\ell}$, we continue as follows:

$$\begin{aligned}
\mathrm{E}(S_2^T + L_2^T \mid s,\ell) &\leq \sum_{v\in T, l(v)=\mathrm{q}} p^v_{s,\ell} \cdot \frac{s}{n-2} + \sum_{v\in T, l(v)=\mathrm{p}} p^v_{s,\ell} \cdot \frac{\ell}{n-2} + \sum_{\substack{v\in T\\ v \text{ bad}}} p^v_{s,\ell} + o(n) \\
&= \frac{f^{\mathrm{q}}_{s,\ell}\cdot s + f^{\mathrm{p}}_{s,\ell}\cdot \ell}{n-2} + \sum_{v\in T, v \text{ bad}} p^v_{s,\ell} + o(n) \\
&= \frac{f^{\mathrm{q}}_{s,\ell}\cdot s + f^{\mathrm{p}}_{s,\ell}\cdot \ell}{n-2} + \sum_{i=0}^{n-3} \Pr(\text{a bad node on level } i \text{ is reached}) + o(n), \quad (3.10)
\end{aligned}$$

where in the last step we just rewrote the sum to consider each level in the classification tree separately. So, to show (3.7) it remains to bound the sum in (3.10) by $o(n)$.

To see this, consider a random input that is classified using T. We will show that with very high probability we do not reach a bad node in the classification tree in the first $n - n^{3/4}$ levels. Intuitively, this means that it is highly improbable that underway the observed fraction of small elements deviates very far from the average $s/(n-2)$. In the following, we will only consider nodes which are labeled with "p". Analogously, these statements are valid for nodes labeled with "q".

Let s_i be the random variable that counts the number of small elements classified in the first i classification steps. By Lemma 3.2.2, with very high probability we have that $|s_i - \mathrm{E}(s_i)| \leq n^{2/3}$. Suppose this events occurs.

We may calculate

$$\left|\frac{s}{n-2} - \frac{s-s_i}{n-2-i}\right| \leq \left|\frac{s}{n-2} - \frac{s(1-i/(n-2))}{n-2-i}\right| + \left|\frac{n^{2/3}}{n-2-i}\right| = \frac{n^{2/3}}{n-2-i}.$$

That means that for each of the first $i \leq n - n^{3/4}$ levels with very high probability we are in a *good node* on level i, because the deviation from the ideal case that we see a small element with probability $s/(n-2)$ is $n^{2/3}/(n-2-i) \leq n^{2/3}/n^{3/4} = 1/n^{1/12}$. Thus, for the first $n - n^{3/4}$ levels the contribution of the sums of the probabilities of bad nodes in (3.10) is $o(n)$. For the last $n^{3/4}$ levels of the tree, we use that the contribution of the probabilities that we reach a bad node on level i is at most 1 for a fixed level.

This shows that the contribution of he sum in (3.10) is $o(n)$. This finishes the proof of the upper bound. The calculations for the lower bound are similar and are omitted here. □

3.3. Discussion

Lemma 3.2.3 and Theorem 3.1.2 tell us that for the analysis of the average comparison count of a dual-pivot quicksort algorithm we just have to find out what $f^{\mathrm{p}}_{s,\ell}$ and $f^{\mathrm{q}}_{s,\ell}$ are for this algorithm. Moreover, to design a good algorithm (w.r.t. the average comparison count), we should try to make $f^{\mathrm{q}}_{s,\ell} \cdot s + f^{\mathrm{p}}_{s,\ell} \cdot \ell$ small for each pair s, ℓ.

To model dual-pivot quicksort algorithms and study their average comparison count, we introduced the concept of a *classification strategy*, which is a three-way decision tree. The reader might wonder whether this model is general enough or not. We describe two ideas for possible generalizations, and show how they are covered by our model.

Randomized Classifications. Of course, one could also think of allowing random choices inside the nodes of the decision tree, e. g., *"flip a coin to choose the pivot used in the first comparison."* This, however, can just be seen as a probability distribution on deterministic classification strategies. But this means that for every randomized classification strategy there is also a deterministic strategy which is at least as good on average.

Postponing Classifications. In our model we enforce that an element has to be classified as soon as it is inspected for the first time. Of course, it can be allowed that an element is left unclassified (i. e., that its group is not determined after the first classification) and reconsidered later. Intuitively, this should not help with respect to lowering the comparison count: If the element is left unclassified then one more comparison is needed later to determine its group. Moreover, one could think that not classifying an element is a disadvantage, since it could be useful for future decisions to know about previous classifications. Neither is the case and we make this statement precise in the following way. Naturally, postponing classifications can be modeled by a decision tree which allows inner nodes that either have two or three children. But given such a decision tree T one can build an equivalent decision tree T' in which each inner node has degree 3. We sketch one way to do this transformation. Let v be a node with only two children in T. Let $p = (v_0, v_1, \ldots, v_t = v)$ be the unique path from the root v_0 of T to v. If there exists a node v_j with $i(v_j) = i(v)$ and $j < t$, then identify v with v_j. Wlog. assume that $l(v) = \mathrm{p}$. Then construct a decision tree T' from T in the following way: Let T_v be the subtree rooted at v in T. Let T_μ be the tree we obtain from T_v by taking each non-root node v' with $i(v') = i(v)$ and change the edge pointing to it from its parent to point to its μ-child. (The node v' and its λ-child are thus removed from T_μ.) Analogously, let T_λ be the tree we obtain from T_v by taking each non-root node v' with $i(v') = i(v)$ and change the edge pointing to it from its parent to point to its λ-child. Now construct T' in the obvious way: First, let $T' = T$. Then, replace the subtree reached from v by following the edge labeled μ with the subtree from T_μ that is rooted at the μ-child of v. Finally, add the subtree reached by following the edge labeled μ of T_λ as a child to v; label the edge from v to the root of this tree with λ. Starting with an arbitrary decision tree with nodes with two and three children, this process is iterated until there

is no node with two children left. Thus, postponing classifications does not help with respect to improving the average comparison count.

4. Classification Strategies For Dual-Pivot Quicksort

This section is devoted to the study of different classification strategies. In the first section, we will analyze the average comparison count of some known strategies. Then, we will study classification algorithms which minimize the average comparison count. Pseudocode for actual dual-pivot algorithms using these classification strategies is provided in Appendix A. The reader is invited to look at the pseudocode to see the simplicity of dual-pivot quicksort algorithms.

4.1. Analysis of Some Known Strategies

Oblivious Strategies. We will first consider strategies that do not use information of previous classifications for future classifications. To this end, we call a classification tree *oblivious* if for each level all nodes v on this level share the same label $l(v) \in \{\mathrm{p}, \mathrm{q}\}$. This means that these algorithms do not react to the outcome of previous classifications, but use a fixed sequence of pivot choices. Examples for such strategies are, e. g.,

- always compare to the larger pivot first (we refer to this strategy by the letter "$\mathcal{L}$"),
- always compare to the smaller pivot first,
- alternate the pivots in each step.

Let T be an oblivious classification tree. Let f_n^{q} denote the average number of comparisons to the larger pivot first. By assumption this value is independent of s and ℓ. Hence these strategies make sure that $f_{s,\ell}^{\mathrm{q}} = f_n^{\mathrm{q}}$ and $f_{s,\ell}^{\mathrm{p}} = n - 2 - f_n^{\mathrm{q}}$ for all pairs of values s, ℓ.

Applying Lemma 3.2.3 gives us

$$\begin{aligned}
\mathrm{E}(P_n) &= \frac{4}{3}n + \frac{1}{\binom{n}{2} \cdot (n-2)} \cdot \sum_{s+\ell \le n-2} (f_n^{\mathrm{q}} \cdot s + (n - 2 - f_n^{\mathrm{q}}) \cdot \ell) + o(n) \\
&= \frac{4}{3}n + \frac{f_n^{\mathrm{q}}}{\binom{n}{2} \cdot (n-2)} \cdot \left(\sum_{s+\ell \le n-2} s \right) + \frac{n - 2 - f_n^{\mathrm{q}}}{\binom{n}{2} \cdot (n-2)} \cdot \left(\sum_{s+\ell \le n-2} \ell \right) + o(n) \\
&= \frac{4}{3}n + \frac{1}{\binom{n}{2}} \cdot \left(\sum_{s+\ell \le n-2} s \right) + o(n) = \frac{5}{3}n + o(n).
\end{aligned}$$

Using Theorem 3.1.2 we get $\mathrm{E}(C_n) = 2n \ln n + o(n \ln n)$—the leading term being the same as in standard quicksort. So, for each strategy that does not adapt to the outcome of previous classifications, there is no difference to the average comparison count of classical quicksort. We believe that this is one reason why dual-pivot quicksort seemed inferior to classical quicksort for a long time.[1]

Yaroslavskiy's Algorithm. Following [WN12, Section 3.2], Yaroslavskiy's algorithm is an implementation based on the following strategy $\mathcal{Y}$: *Compare ℓ elements to q first, and compare the other elements to p first.*[2]

We get that $f^{\mathrm{q}}_{s,\ell} = \ell$ and $f^{\mathrm{p}}_{s,\ell} = s + m$. Applying Lemma 3.2.3, we calculate

$$\mathrm{E}\left(P_n^{\mathcal{Y}}\right) = \frac{4}{3}n + \frac{1}{\binom{n}{2}} \sum_{s+\ell \le n-2} \left(\frac{s\ell}{n-2} + \frac{(s+m)\ell}{n-2} \right) + o(n).$$

Of course, it is possible to evaluate this sum by hand. We used Maple® to obtain $\mathrm{E}\left(P_n^{\mathcal{Y}}\right) = \frac{19}{12}n + o(n)$. Using Theorem 3.1.2 gives $\mathrm{E}\left(C_n\right) = 1.9n \ln n + o(n \ln n)$, as in [WN12].

Sedgewick's Algorithm. Following [WN12, Section 3.2], Sedgewick's algorithm amounts to an implementation of the following strategy $\mathcal{S}$: *Compare (on average) a fraction of $s/(s+\ell)$ of the keys with q first, and compare the other keys with p first.* We get $f^{\mathrm{q}}_{s,\ell} = (n-2) \cdot s/(s+\ell)$ and $f^{\mathrm{p}}_{s,\ell} = (n-2) \cdot \ell/(s+\ell)$. Plugging these values into Lemma 3.2.3, we calculate

$$\mathrm{E}\left(P_n^{\mathcal{S}}\right) = \frac{4}{3}n + \frac{1}{\binom{n}{2}} \sum_{s+\ell \le n-2} \left(\frac{s^2}{s+\ell} + \frac{\ell^2}{s+\ell} \right) + o(n) = \frac{16}{9}n + o(n).$$

Applying Theorem 3.1.2 gives $\mathrm{E}(C_n) = 2.133\ldots \cdot n \ln n + o(n \ln n)$, as known from [WN12].

Obviously, this is worse than the oblivious strategies considered before.[3] This is easily explained intuitively: If the fraction of small elements is large, it will compare many elements with q first. But this costs two comparisons for each small element. Conversely, if the fraction of large elements is large, it will compare many elements to p first, which is again the wrong decision.

Since Sedgewick's strategy seems to do exactly the opposite of what one should do to lower the

[1] This does not mean that oblivious strategies do not have other advantages over classical quicksort. The simple strategy $\mathcal{L}$ will be among the fastest algorithms in our experiments.

[2] The idea behind this is simple: By default, we compare against p first. But whenever we classify an element as being large, the next classification is started by comparing against q first. We see that this is slightly different to strategy $\mathcal{Y}$: It makes $\ell - 1$ comparisons to the larger pivot first, if the element classified last is large. Otherwise, it makes ℓ comparisons to the larger pivot first. So, we get $f^{\mathrm{q}}_{s,\ell} = \ell - \alpha$ and $f^{\mathrm{p}}_{s,\ell} = s + m + \alpha$, for some $0 \le \alpha \le 1$. The difference of this strategy and strategy $\mathcal{Y}$ with regard to the average comparison count for classification vanishes in the $o(n)$ term. Thus, we disregard this detail in the discussion.

[3] We remark that in his thesis Sedgewick [Sed75] focused on the average number of swaps, not on the comparison count.

comparison count, we consider the following modified strategy $\mathcal{S}'$: *For given p and q, compare (on average) a fraction of $s/(s+\ell)$ of the keys with p first, and compare the other keys with q first.* ($\mathcal{S}'$ simply uses p first when $\mathcal{S}$ would use q first and vice versa.)

Using the same analysis as above, we get $\mathrm{E}(P_n) = \frac{14}{9}n + o(n)$, which yields $\mathrm{E}(C_n) = 1.866\ldots \cdot n \ln n + o(n \ln n)$—improving on the standard quicksort algorithm and even on Yaroslavskiy's algorithm! Note that this has been observed by Wild in his Master's Thesis as well [Wil13].

Remark. Swapping the first comparison of p with q and vice versa as in the strategy described above is a general technique. In fact, if the leading coefficient of the average number of comparisons for a fixed rule for choosing p or q first is α, e. g., $\alpha = 2.133\ldots$ for strategy $\mathcal{S}$, then the leading coefficient of the strategy that does the opposite is $4 - \alpha$, e. g., $4 - 2.133\ldots = 1.866\ldots$ as in strategy $\mathcal{S}'$.

To make this precise, let $\mathcal{A}$ be a strategy that uses a fixed choice of $f^{\mathrm{p}}_{s,\ell}$ and $f^{\mathrm{q}}_{s,\ell}$. Let $\mathcal{A}'$ be a strategy that uses $g^{\mathrm{p}}_{s,\ell} = f^{\mathrm{q}}_{s,\ell}$ and $g^{\mathrm{q}}_{s,\ell} = f^{\mathrm{p}}_{s,\ell}$. (Such a strategy is easily obtained by exchanging the labels $l(v)$ in the decision tree that corresponds to strategy $\mathcal{A}$.) Since $f^{\mathrm{p}}_{s,\ell} = (n - 2 - f^{\mathrm{q}}_{s,\ell})$, summing up to the additional cost terms of $\mathcal{A}$ and $\mathcal{A}'$ in Lemma 3.2.3 leads to

$$\begin{aligned}
&\frac{1}{\binom{n}{2}}\Bigg(\sum_{s+\ell\le n-2}\left(\frac{f^{\mathrm{q}}_{s,\ell}\cdot s}{n-2}+\frac{f^{\mathrm{p}}_{s,\ell}\cdot \ell}{n-2}\right)+\sum_{s+\ell\le n-2}\left(\frac{g^{\mathrm{q}}_{s,\ell}\cdot s}{n-2}+\frac{g^{\mathrm{p}}_{s,\ell}\cdot \ell}{n-2}\right)\Bigg)+o(n)\\
=&\frac{1}{\binom{n}{2}}\sum_{s+\ell\le n-2}(s+\ell)+o(n)=\frac{2}{3}(n-2)+o(n).
\end{aligned}$$

So, if the additional cost term of $\mathcal{A}$ is $b \cdot n + o(n)$, for a constant b, then the additional cost term of $\mathcal{A}'$ is $(2/3 - b) \cdot n + o(n)$. Now let $\alpha = 6/5 \cdot (4/3 + b)$, i. e., $\mathrm{E}(C^{\mathcal{A}}_n) = \alpha \cdot n \ln n + o(n \ln n)$. Using (3.3), we obtain by a standard calculation

$$\begin{aligned}
\mathrm{E}(C^{\mathcal{A}'}_n) &= \frac{6}{5}\cdot\left(\frac{4}{3}+\frac{2}{3}-b\right)\cdot n\ln n + o(n\ln n)\\
&= (4-\alpha)n\ln n + o(n\ln n),
\end{aligned}$$

which precisely describes the influence of exchanging $f^{\mathrm{p}}_{s,\ell}$ and $f^{\mathrm{q}}_{s,\ell}$ to the leading term of the average comparison count.

4.2. (Asymptotically) Optimal Classification Strategies

We now present *optimal* classification strategies, that means, classification strategies which achieve the minimum average comparison count in our model of dual-pivot quicksort algorithms. At first, we will consider two different strategies whose optimality proof is surprisingly simple. However, they require that after the pivots p and q are chosen, the algorithm knows s and ℓ. We call them

"improper" classification strategies, because a classification strategy uses only a single classification tree. In the second part of this subsection, we will slightly change these two strategies and obtain "real" classification strategies. The main task is then to show that these changes do not affect the dominating term of the average comparison count.

4.2.1. Two Unrealistic (Asymptotically) Optimal Strategies

We consider the following strategy $\mathcal{O}$: *Given s and ℓ, the comparison at node v is with the smaller pivot first if $s - s_v > \ell - \ell_v$, otherwise it is with the larger pivot first.*[4] (For the definition of s_v and ℓ_v, see Page 19.)

Theorem 4.2.1
Strategy $\mathcal{O}$ is optimal, i. e., its ACT (see (3.5)) is at most as large as ACT_T for every single classification tree T. When using $\mathcal{O}$ in a dual-pivot quicksort algorithm, we get $\mathrm{E}(C_n^{\mathcal{O}}) = 1.8n \ln n + O(n)$.

Proof. Fix the two pivots. We will prove each statement separately.

First statement: According to (3.6), the contribution of an arbitrary node v in the decision tree to the additional cost term is at least

$$p_{s,\ell}^v \cdot \frac{\min\{s - s_v, \ell - \ell_v\}}{n - 2 - \mathrm{level}(v)}.$$

Strategy $\mathcal{O}$ chooses the label of each node in the decision tree such that this minimum contribution is achieved, and hence minimizes the additional cost term in (3.6).

Second statement: We first derive an upper bound of $1.8n \ln n + O(n)$ for the average number of comparisons, and then show that this is tight.

For the first part, let an input with n entries and two pivots be given, so that there are s small and ℓ large elements. Assume $s \geq \ell$. Omit all medium elements to obtain a reduced input $(a_1, \ldots, a_{n'})$ with $n' = s + \ell$. For $0 \leq i \leq n'$ let s_i and ℓ_i denote the number of small resp. large elements remaining in $(a_{i+1}, \ldots, a_{n'})$. Let $D_i = s_i - \ell_i$. Of course we have $D_0 = s - \ell$ and $D_{n'} = 0$. Let $i_1 < i_2 < \ldots < i_k$ be the list of indices i with $D_i = 0$. (In particular, $i_k = n'$.) Rounds i with $D_i = 0$ are called *zero-crossings*. Consider some j with $D_{i_j} = D_{i_{j+1}} = 0$. The numbers $D_{i_j+1}, \ldots . D_{i_{j+1}-1}$ are nonzero and have the same positive [or negative] sign. The algorithm compares $a_{i_j+2}, \ldots, a_{i_{j+1}}$ with the smaller [or larger] pivot first, and a_{i_j+1} with the larger pivot first. Since $\{a_{i_j+1}, \ldots, a_{i_{j+1}}\}$ contains the same number of small and large elements, the contribution of this segment to the additional comparison count is $\frac{1}{2}(i_{j+1} - i_j) - 1$ $\big[$or $\frac{1}{2}(i_{j+1} - i_j)\big]$.

If $D_0 > 0$, i. e., $s > \ell$, all elements in $\{a_1, \ldots, a_{i_1}\}$ are compared with the smaller pivot first, and this set contains $\frac{1}{2}(i_1 - (s - \ell))$ large elements (and $\frac{1}{2}(i_1 + (s - \ell))$ small elements), giving

[4] This strategy was suggested to us by Thomas Hotz (personal communication).

a contribution of $\frac{1}{2}(i_1 - (s - \ell))$ to the additional comparison count. Overall, the additional comparison count of strategy $\mathcal{O}$ on the considered input is

$$\frac{i_1 - (s-\ell)}{2} + \sum_{j=1}^{k-1} \frac{i_{j+1} - i_j}{2} - k^* = \frac{n' - (s-\ell)}{2} - k^* = \ell - k^*,$$

for some correction term $k^* \in \{0, \dots, k\}$.

Averaging the upper bound ℓ over all pivot choices, we see that the additional cost term of strategy $\mathcal{O}$ is at most

$$\frac{1}{\binom{n}{2}} \cdot \left(2 \cdot \sum_{\substack{s+\ell \le n \\ \ell < s}} \ell + \sum_{\ell \le n/2} \ell \right), \tag{4.1}$$

which gives an average number of at most $1.5n + O(1)$ comparisons. For such a partitioning cost we can use (2.1) and obtain an average comparison count for sorting via strategy $\mathcal{O}$ of at most $1.8n \ln n + O(n)$.

It remains to show that this is tight. This follows by a lengthy calculation. We shall see that the essential step in this analysis is to show that the average (over all inputs) of the number of *zero-crossings* (the number k from above) is $O(\log n)$. Again, we temporarily omit medium elements to simplify calculations, i. e., we assume that the number of small and large elements together is n. Let Z_n be the random variable that denotes the number of zero-crossings for an input of n elements. We calculate:

$$\begin{aligned}
\mathrm{E}(Z_n) &= \sum_{1 \le i \le n/2} \Pr(\text{there is a zero-crossing at position } n - 2i) \\
&= \frac{2}{n} \sum_{i=1}^{n/2} \sum_{s=i}^{n/2} \Pr(\text{there is a zero-crossing at position } n - 2i \mid s \text{ small elements}) \\
&= \frac{2}{n} \sum_{i=1}^{n/2} \sum_{s=i}^{n/2} \frac{\binom{2i}{i} \cdot \binom{n-2i}{s-i}}{\binom{n}{s}}.
\end{aligned}$$

By using the well-known estimate $\binom{2i}{i} = \Theta(2^{2i}/\sqrt{i})$ (which follows directly from Stirling's

approximation), we continue by

$$\begin{aligned}\mathrm{E}(Z_n) &= \Theta\left(\frac{1}{n}\right)\sum_{i=1}^{n/2}\frac{2^{2i}}{\sqrt{i}}\sum_{s=s}^{n/2}\frac{\binom{n-2i}{n-s}}{\binom{n}{s}}\\ &= \Theta\left(\frac{1}{n}\right)\sum_{i=1}^{n/2}\frac{2^{2i}}{\sqrt{i}}\sum_{s=i}^{n/2}\frac{(n{-}2i)\cdot\ldots\cdot(n{-}i{-}s{+}1)\cdot s\cdot\ldots\cdot(s{-}i{+}1)}{n\cdot\ldots\cdot(n{-}s{+}1)}\\ &= \Theta\left(\frac{1}{n}\right)\sum_{i=1}^{n/2}\frac{n+1}{\sqrt{i}(n{-}2i{+}1)}\sum_{j=0}^{n/2-i}\prod_{k=0}^{i-1}\frac{(n{+}2j{-}2k)(n{-}2j{-}2k)}{(n{-}2k{+}1)(n{-}2k)},\end{aligned} \tag{4.2}$$

where the last step follows by an index transformation using $j = s - i$ and multiplying 2^{2i} into the terms of the right-most fraction. We now obtain an upper bound for the right-most product:

$$\prod_{k=0}^{i-1}\frac{(n{+}2j{-}2k)(n{-}2j{-}2k)}{(n{-}2k{+}1)(n{-}2k)} \le \prod_{k=0}^{i-1}\left(1-\left(\frac{2j}{n-2k}\right)^2\right) \le \left(1-\left(\frac{2j}{n}\right)^2\right)^i.$$

We substitute this bound into (4.2) and bound the right-most sum by an integral:

$$\mathrm{E}(Z_n) = O\left(\frac{1}{n}\right)\sum_{i=1}^{n/2}\frac{n+1}{\sqrt{i}(n-2i+1)}\left(\int_0^{n/2}\left(1-\left(\frac{2t}{n}\right)^2\right)^i \mathrm{d}t + 1\right). \tag{4.3}$$

We now obtain a bound on the integral as follows:

$$\begin{aligned}\int_0^{n/2}\left(1-\left(\frac{2t}{n}\right)^2\right)^i \mathrm{d}t = \frac{n}{2}\int_0^1\left(1-t^2\right)^i \mathrm{d}t &\overset{(1)}{=} \frac{n}{2}\sum_{k=0}^{i}(-1)^k\binom{i}{k}\frac{1}{2k+1}\\ &\overset{(2)}{=} \frac{n}{2}\cdot\frac{\Gamma\left(\frac{1}{2}\right)\cdot\Gamma\left(i+1\right)}{\Gamma\left(i+\frac{3}{2}\right)} \overset{(3)}{=} \Theta\left(\frac{n}{\sqrt{i}}\right),\end{aligned}$$

involving the Gamma function $\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}\mathrm{d}t$. Here, (1) follows according to the Binomial theorem, (2) is a well-known identity for the sum, see, e.g., [Gou72, Identity (1.40)], and (3) follows by Stirling's approximation and the identity

$$\Gamma\left(i+\frac{1}{2}\right) = \frac{(2i)!}{i!\cdot 4^i}\cdot\sqrt{\pi},$$

which can be checked by induction using $\Gamma\left(\frac{1}{2}\right) = \sqrt{\pi}$ and $\Gamma(x) = (x-1)\cdot\Gamma(x-1)$.

So, we can continue our calculation at (4.3) and obtain

$$\mathrm{E}(Z_n) = O\left(\sum_{i=1}^{n/2} \frac{n+1}{i(n-2i+1)}\right) = O\left(\sum_{i=1}^{n/4} \frac{1}{i} + \sum_{i=n/4+1}^{n/2} \frac{1}{n-2i+1}\right) = O(\log n).$$

Now we consider the case that the input contains medium elements. Fix the number of small elements and the number of large elements. Medium elements have no influence on the value of the additional cost term, so we can remove them and get the same number of additional comparisons. This means that we should not consider a round i to be a zero crossing, when there was a zero-crossing in round $i-1$, for $1 \leq i \leq n-2$. Thus, we have that

$$\mathrm{E}\left(Z_{s+m+\ell} \mid s \text{ small}, m \text{ medium}, \ell \text{ large elements}\right) = \mathrm{E}\left(Z_{s+\ell} \mid s \text{ small}, \ell \text{ large elements}\right),$$

i. e., it equals the average number of zero-crossings for a smaller input containing $s+\ell$ elements.

We can now simply calculate the average number of zero-crossing for an arbitrary input as follows:

$$\mathrm{E}(Z_n) = \frac{1}{\binom{n}{2}} \sum_{s+\ell \leq n} \mathrm{E}(Z_n \mid s, \ell) = \frac{1}{\binom{n}{2}} \sum_{s+\ell \leq n} O(\log(s+\ell)) = O(\log n).$$

So, the difference between the upper bound on the additional cost term shown in (4.1) and the actual additional cost term is $O(\log n)$. It remains to show that the influence of these $O(\log n)$ terms to the total average sorting cost is bounded by $O(n)$. By linearity of expectation, we consider these terms in the average sorting cost of (3.1) separately. So, assume that the cost associated with a partitioning step involving a subarray of length n is $c \cdot \log n$ for a constant c.

We show by induction on the input size that the contributions of the $c \log n$ terms sum up to at most $O(n)$ for the total average comparison count. Let $\mathrm{E}(A_n)$ denote the sum of the error terms in the average comparison count. We will show that

$$\mathrm{E}(A_n) \leq C \cdot n - D \ln n, \tag{4.4}$$

for suitable constants C and D.

Let $D \geq c/5$. For the base case, let $n_0 \in \mathbb{N}$ and set C such that $\mathrm{E}(A_n) \leq C \cdot n - D \ln n$ for all $n < n_0$. As the induction hypothesis, assume that (4.4) holds for all $n' < n$. For the induction

step, we calculate:

$$\begin{aligned}\mathrm{E}(A_n) &= \frac{1}{\binom{n}{2}} \sum_{s+m+\ell=n} \mathrm{E}(A_n \mid s,m,\ell) \\ &\le C\cdot n + \frac{1}{\binom{n}{2}} \sum_{s+m+\ell=n} (c\ln n - D\cdot(\ln s + \ln m + \ln \ell)) \\ &= C\cdot n + c\ln n - \frac{3}{\binom{n}{2}} \sum_{s+m+\ell=n} D\cdot \ln s = C\cdot n + c\ln n - \frac{6}{n}\sum_{1\le s\le n} D\cdot \ln s\end{aligned}$$

We use that $\sum_{i=a}^{b} f(i) \ge \int_a^b f(x)\mathrm{d}x$ for monotone functions f defined on $[a,b]$ and obtain

$$\mathrm{E}(A_n) \le C\cdot n + c\ln n - \frac{6D}{n}\cdot(n\ln n - n + 1) = C\cdot n + c\ln n - 6D\cdot(\ln n - 1 + 1/n).$$

An easy calculation shows that from $D \ge \frac{c}{5}$ it follows that

$$c\ln n - 6D(\ln n - 1 + 1/n) \le -D\ln n,$$

which finishes the induction step.

Thus, the additional $O(\log n)$ terms sum up to $O(n)$ in the total average comparison count. Thus, the difference between the upper bound of $1.8n\ln n + O(n)$ derived in the proof of Theorem 4.2.1 and the exact cost is $O(n)$, and so the total average sorting cost of strategy $\mathcal{O}$ is $1.8n\ln n + O(n)$. □

Strategy $\mathcal{O}$ is the optimal strategy for dual-pivot quicksort. However, there exist other strategies whose comparison count for classification will differ by only an $o(n)$ term. We call these strategies *asymptotically optimal.*

We will now study the following "oracle" strategy $\mathcal{N}$: *If $s > \ell$ then always compare with p first, otherwise always compare with q first.*

Theorem 4.2.2
When using $\mathcal{N}$ in a dual pivot quicksort algorithm, we have $\mathrm{E}(C_n^{\mathcal{N}}) = 1.8n\ln n + o(n\ln n)$.

Proof. Lemma 3.2.3 says that the additional cost term for fixed pivots is

$$f_{s,\ell}^{\mathrm{q}}\cdot s + f_{s,\ell}^{\mathrm{p}}\cdot \ell + o(n). \tag{4.5}$$

If $\ell < s$, strategy $\mathcal{N}$ sets $f_{s,\ell}^{\mathrm{q}} = n-2$ and $f_{s,\ell}^{\mathrm{p}} = 0$; otherwise it sets $f_{s,\ell}^{\mathrm{q}} = 0$ and $f_{s,\ell}^{\mathrm{p}} = n-2$.

Using symmetry this means that the additional cost term of strategy $\mathcal{N}$ is

$$\frac{1}{\binom{n}{2}}\left(2\cdot\sum_{\substack{s+\ell\le n\\ \ell<s}}\ell+\sum_{\ell\le n/2}\ell\right)+o(n),$$

which means that

$$\mathrm{E}(P_n)=\frac{4}{3}n+\frac{1}{6}n+o(n)=1.5n+o(n).$$

Plugging this value into Theorem 3.1.2 gives $\mathrm{E}(C_n)=1.8n\ln n+o(n\ln n)$. □

4.2.2. Two Realistic Asymptotically Optimal Strategies

While strategy $\mathcal{O}$ looks into the yet unclassified part of the input and is interested if there are more small or more large elements in it, this decision could just be based on what has been seen so far.

We consider the following "counting" strategy $\mathcal{C}$: *The comparison at node v is with the smaller pivot first if $s_v > \ell_v$, otherwise it is with the larger pivot first.*

It is not hard to see that for some inputs the number of additional comparisons of strategy $\mathcal{O}$ and $\mathcal{C}$ can differ significantly. The next theorem shows that averaged over all possible inputs, however, there is only a small difference.

Theorem 4.2.3
Let $\mathrm{ACT}_{\mathcal{O}}$ and $\mathrm{ACT}_{\mathcal{C}}$ be the ACT for classifying n elements using strategy $\mathcal{O}$ and $\mathcal{C}$, respectively. Then $\mathrm{ACT}_{\mathcal{C}}=\mathrm{ACT}_{\mathcal{O}}+O(\log n)$. When using $\mathcal{C}$ in a dual-pivot quicksort algorithm, we get $\mathrm{E}(C_n^{\mathcal{C}})=1.8n\ln n+O(n)$.

Proof. Assume that strategy $\mathcal{O}$ inspects the elements in the order $a_{n-1},\dots,a_2$, while $\mathcal{C}$ uses the order $a_2,\dots,a_{n-1}$. If the strategies compare the element a_i to different pivots, then there are exactly as many small elements as there are large elements in $\{a_2,\dots,a_{i-1}\}$ or $\{a_2,\dots,a_i\}$, depending on whether i is even or odd, see Figure 4.1. Thus, the same calculation as in the proof of Theorem 4.2.1 shows that $\mathrm{ACT}_{\mathcal{C}}-\mathrm{ACT}_{\mathcal{O}}$ is $O(\log n)$, which sums up to a total additive contribution of $O(n)$ when using strategy $\mathcal{C}$ in a dual-pivot quicksort algorithm, see the proof of Theorem 4.2.1 for details. □

Thus, dual-pivot quicksort with strategy $\mathcal{C}$ has average cost at most $O(n)$ larger than dual-pivot quicksort using the (unrealistic) optimal strategy $\mathcal{O}$.

Now we consider a realistic variation of strategy $\mathcal{N}$. To decide whether there are more small or more large elements in the input, we take a sample of $n^{3/4}$ elements and make a guess based on this sample.

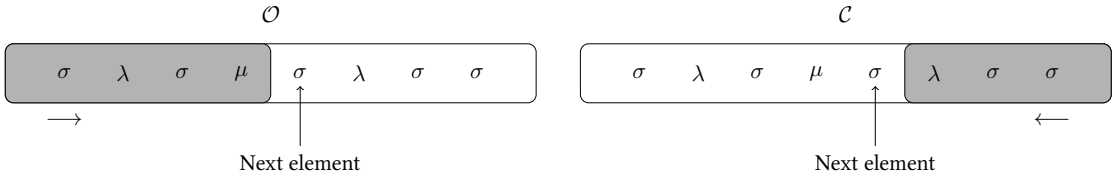

Figure 4.1.: Visualization of the decision process when inspecting an element using strategy $\mathcal{O}$ (left) and $\mathcal{C}$ (right). Applying strategy $\mathcal{O}$ from left to right uses that of the remaining elements three are small and one is large, so it decides that the element should be compared with p first. Applying strategy $\mathcal{C}$ from right to left uses that of the inspected elements two were small and only one was large, so it decides to compare the element with p first, too. Note that the strategies would differ if, e. g., the right-most element would be a medium element.

Specifically, consider the following "sampling" strategy $\mathcal{SP}$: *"Make the first $n^{3/4}$ comparisons against the smaller pivot first. Let s' denote the number of small elements, and let ℓ' denote the number of large elements seen in these first comparisons. If $s' > \ell'$, compare the remaining elements with p first, otherwise compare them with q first."*

Theorem 4.2.4
Let $\text{ACT}_{\mathcal{N}}$ and $\text{ACT}_{\mathcal{SP}}$ be the ACT for classifying n elements using strategy $\mathcal{N}$ and strategy $\mathcal{SP}$, respectively. Then $\text{ACT}_{\mathcal{SP}} = \text{ACT}_{\mathcal{N}} + o(n)$. When using $\mathcal{SP}$ in a dual-pivot quicksort algorithm, we get $\mathrm{E}(C_n^{\mathcal{SP}}) = 1.8n \ln n + o(n \ln n)$.

Proof. Fix the two pivots p and q, and thus s, m, and ℓ. Assume that $s < \ell$. (The other case follows by symmetry.)

Strategy $\mathcal{N}$ makes exactly s additional comparisons. We claim that strategy $\mathcal{SP}$ makes at most $s + o(n)$ additional comparisons. From Lemma 3.2.3, we know that the additional cost term of $\mathcal{SP}$ for fixed pivot choices is

$$\mathrm{E}\left(Z_n^{\mathcal{SP}} \mid s, \ell\right) = \frac{f_{s,\ell}^{\mathrm{q}} \cdot s + f_{s,\ell}^{\mathrm{p}} \cdot \ell}{n-2} + o(n).$$

We will now distinguish two cases:

Case 1: $s + 2n^{11/12} \geq \ell$. (The segment sizes s and ℓ are close to each other.)

Since $f^{\mathrm{p}}_{s,\ell} + f^{\mathrm{q}}_{s,\ell} = n - 2$, we may calculate:

$$\begin{aligned}\mathrm{E}(Z \mid s, \ell) &= \frac{f^{\mathrm{q}}_{s,\ell} \cdot s + f^{\mathrm{p}}_{s,\ell} \cdot \ell}{n-2} + o(n) \\ &\leq \frac{f^{\mathrm{q}}_{s,\ell} \cdot s + (s + 2n^{11/12}) \cdot f^{\mathrm{p}}_{s,\ell}}{n-2} + o(n) \\ &= s + o(n),\end{aligned}$$

So, the difference between strategy $\mathcal{N}$ and strategy $\mathcal{SP}$ is $o(n)$.

Case 2: $s + 2n^{11/12} < \ell$. (The segment sizes s and ℓ are far away from each other.)

By Lemma 3.2.2 we know that with very high probability $|s' - \mathrm{E}(s')| \leq n^{2/3}$ and $|\ell' - \mathrm{E}(\ell')| \leq n^{2/3}$. Now, consider a random input with s small and ℓ large elements such that $s + 2n^{11/12} < \ell$ and s' and ℓ' are concentrated around their expectation. We will now show that concentration of s' and ℓ' implies that $s' < \ell'$, and hence that $\mathcal{SP}$ (correctly) compares all elements to p first.

By assumption, we have that

$$\left| s' - n^{3/4} \cdot \frac{s}{n-2} \right| \leq n^{2/3} \text{ and } \left| \ell' - n^{3/4} \cdot \frac{\ell}{n-2} \right| \leq n^{2/3}.$$

So, we conclude that in this case $s' < s \cdot n^{-1/4} + n^{2/3}$ and $\ell' > \ell \cdot n^{-1/4} - n^{2/3}$. We calculate:

$$s' < \frac{s}{n^{1/4}} + n^{2/3} < \frac{\ell - 2n^{11/12}}{n^{1/4}} + n^{2/3} = \frac{\ell}{n^{1/4}} - 2n^{2/3} + n^{2/3} = \frac{\ell}{n^{1/4}} - n^{2/3} < \ell'.$$

Thus, whenever s' and ℓ' are close enough to their expectation, and s and ℓ far away from each other, strategy $\mathcal{SP}$ chooses the correct pivot for the first comparison. So, if s and ℓ are far away from each other, then the difference between the average classification cost of $\mathcal{N}$ and $\mathcal{SP}$ is $o(n)$.

We note that in the case that $\ell > s$, strategy $\mathcal{SP}$ chooses the wrong pivot for the first $n^{2/3}$ classifications. This contributes $o(n)$ to the average comparison count, which does not affect the dominating term of the average comparison count.

So, $\mathrm{ACT}_{\mathcal{SP}} = \mathrm{ACT}_{\mathcal{N}} + o(n)$, and applying Theorem 3.1.2 gives a total average comparison count of $1.8n \ln n + o(n \ln n)$. □

4.3. Discussion

In this section we analyzed some known classification strategies. We described two new classification strategies, which both achieve the minimum possible average comparison count up to lower order terms. The minimum average comparison count is $1.8n \ln n + O(n)$.

We will now experimentally evaluate the influence of the lower order terms to the average comparison count of the algorithms considered in this section. For this, we sorted random permutation of $\{1, \ldots, n\}$ using implementations of the classification strategies. The pseudocode

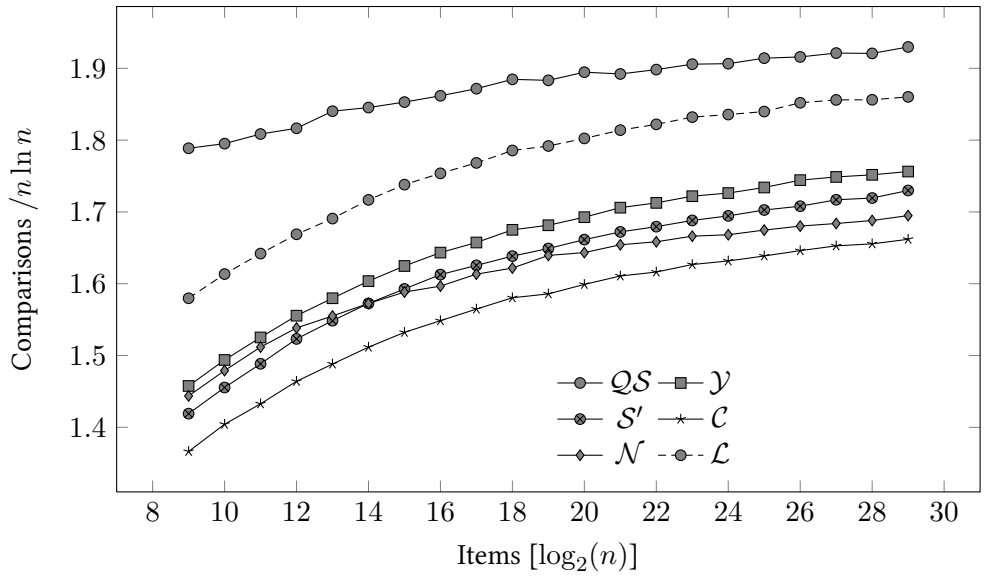

Figure 4.2.: Average comparison count (scaled by $n \ln n$) needed to sort a random input of up to $n = 2^{29}$ integers. We compare classical quicksort ($\mathcal{QS}$), Yaroslavskiy's algorithm ($\mathcal{Y}$), the optimal sampling algorithm ($\mathcal{N}$), the optimal counting algorithm ($\mathcal{C}$), the modified version of Sedgewick's algorithm ($\mathcal{S}'$), and the simple strategy *"always compare to the larger pivot first"* ($\mathcal{L}$). Each data point is the average over 400 trials.

of these algorithms can be found in Appendix A. Figure 4.2 shows the results of this experiment. We make two observations: (i) Lower order terms have a big influence on the average comparison count for "real-life" values of n. (ii) For n large enough, the relations between the different algorithms reflect the relation of the theoretical average comparison count nicely: the counting strategy $\mathcal{C}$ has the lowest comparison count, the sampling strategy $\mathcal{N}$ follows closely. (Note that while they share the same factor in the $n \ln n$ term, the difference due to lower order terms is clearly visible.) Subsequently, the modified version of Sedgewick's algorithm has a lower average comparison count than Yaroslavskiy's algorithm. Next, strategy $\mathcal{L}$ is slightly better for practical input sizes than classical quicksort. (This is also known from theory: on average classical quicksort makes $2n \ln n - 1.51n + O(\ln n)$ [Sed75] comparisons, while strategy $\mathcal{L}$ makes $2n \ln n - 2.75n + O(\ln n)$ comparisons. The latter result can be obtained by plugging in the exact average partition cost of $\mathcal{L}$, which is $5/3(n-2)+1$, into the exact solution of the recurrence (3.1), see [WN12, Section 3.1].)

5. Choosing Pivots From a Sample

In this section we consider the variation of dual-pivot quicksort where the pivots are chosen from a small sample. Intuitively, this guarantees better pivots in the sense that the partition sizes are more balanced. For classical quicksort, the median-of-k strategy is optimal w.r.t. minimizing the average comparison count [MR01], which means that the median in a sample of k elements is chosen as the pivot. The standard implementation of Yaroslavskiy's algorithm in Oracle's Java 7 uses an intuitive generalization of this strategy: it chooses the two tertiles in a sample of five elements as pivots.

We will compare classical quicksort and dual-pivot quicksort algorithms which use the two tertiles of the first five elements of the input, i. e., the second- and the fourth-largest element in the sample, as the two pivots. Moreover, we will see that the optimal pivot choices for dual-pivot quicksort are not the two tertiles of a sample of k elements, but rather the elements of rank $k/4$ and $k/2$.

We remark that while preparing this thesis, Nebel and Wild provided a much more detailed study of pivot sampling in Yaroslavskiy's algorithm [NW14].

5.1. Choosing the Two Tertiles in a Sample of Size 5 as Pivots

We sort the first five elements and take the second- and the fourth-largest elements as pivots. The probability that p and q, $p < q$, are chosen as pivots is exactly $(s \cdot m \cdot \ell)/\binom{n}{5}$. Following Hennequin [Hen91, pp. 52–53], for average partitioning cost $\mathrm{E}(P_n) = a \cdot n + O(1)$ we get

$$\mathrm{E}(C_n) = \frac{1}{H_6 - H_2} \cdot a \cdot n \ln n + O(n) = \frac{20}{19} \cdot a \cdot n \ln n + O(n), \tag{5.1}$$

where H_n denotes the n-th harmonic number.

When applying Lemma 3.2.3, we have average partitioning cost $a \cdot n + o(n)$. To show that the average comparison count becomes $20/19 \cdot a \cdot n \ln n + o(n \ln n)$ in this case, we would have to redo the proof of Theorem 3.1.2. Fortunately, this has already been done by Nebel and Wild in [NW14, Appendix E] for a much more general case of pivot sampling in dual-pivot quicksort.

We will now investigate the effect of pivot sampling on the average number of key comparisons in Yaroslavskiy's and the (optimal) sampling strategy $\mathcal{SP}$, respectively. The average number of

medium elements remains $(n-2)/3$. For strategy $\mathcal{Y}$, we calculate using Maple®

$$\mathrm{E}\left(P_n^{\mathcal{Y}}\right) = \frac{4}{3}n + \frac{1}{\binom{n}{5}} \sum_{s+\ell \le n-5} \frac{\ell \cdot (2s+m) \cdot s \cdot m \cdot \ell}{n-5} + o(n) = \frac{34}{21}n + o(n).$$

Applying (5.1), we get $\mathrm{E}\left(C_n^{\mathcal{Y}}\right) = 1.704n \ln n + o(n \ln n)$ key comparisons on average. (Note that Wild *et al.* [Wil+13] calculated this leading coefficient as well.) This is slightly better than "clever quicksort", which uses the median of a sample of three elements as pivot and achieves $1.714n \ln n + O(n)$ key comparisons on average [Hoa62]. For the sampling strategy $\mathcal{SP}$, we get

$$\mathrm{E}\left(P_n^{\mathcal{SP}}\right) = \frac{4}{3}n + \frac{2}{\binom{n}{5}} \sum_{\substack{s+\ell \le n-5 \\ s \le \ell}} s \cdot s \cdot m \cdot \ell + o(n) = \frac{37}{24}n + o(n).$$

Again using (5.1), we obtain $\mathrm{E}(C_n^{\mathcal{SP}}) = 1.623n \ln n + o(n \ln n)$, improving further on the leading coefficient compared to clever quicksort and Yaroslavskiy's algorithm.

5.2. Pivot Sampling in Classical Quicksort and Dual-Pivot Quicksort

In the previous subsection, we have shown that optimal dual-pivot quicksort using a sample of size 5 clearly beats clever quicksort which uses the median of three elements. We will now investigate how these two variants compare when the sample size grows.

The following proposition, which is a special case of [Hen91, Proposition III.9 and Proposition III.10], will help in this discussion.

Proposition 5.2.1
Let $a \cdot n + O(1)$ be the average partitioning cost of a quicksort algorithm $\mathcal{A}$ that chooses the pivot(s) from a sample of size k, for constants a and k. Then the following holds:

1. If $k+1$ is even and $\mathcal{A}$ is a classical quicksort variant that chooses the median of these k elements as pivot, then the average sorting cost is
$$\frac{1}{H_{k+1} - H_{(k+1)/2}} \cdot a \cdot n \ln n + O(n).$$
2. If $k+1$ is divisible by 3 and $\mathcal{A}$ is a dual-pivot quicksort variant that chooses the two tertiles of these k elements as pivots, then the average sorting cost is
$$\frac{1}{H_{k+1} - H_{(k+1)/3}} \cdot a \cdot n \ln n + O(n).$$

Sample Size	5	11	17	41
Median (QS)	$1.622n \ln n$	$1.531n \ln n$	$1.501n \ln n$	$1.468n \ln n$
Tertiles (DP QS)	$1.623n \ln n$	$1.545n \ln n$	$1.523n \ln n$	$1.504n \ln n$

Table 5.1.: Comparison of the leading term of the average cost of classical quicksort and dual-pivot quicksort for specific sample sizes. Note that for real-life input sizes, however, the linear term can make a big difference.

Note that for classical quicksort we have partitioning cost $n - 1$. Thus, the average sorting cost becomes $\frac{1}{H_{k+1}-H_{(k+1)/2}} n \ln n + O(n)$.

For dual-pivot algorithms, the probability that p and q, $p < q$, are the two tertiles in a sample of size k, where $k + 1$ is divisible by 3, is exactly

$$\frac{\binom{p-1}{(k-2)/3}\binom{q-p-1}{(k-2)/3}\binom{n-q}{(k-2)/3}}{\binom{n}{k}}.$$

Thus, the average partitioning cost $\mathrm{E}\left(P_{n,k}^{\mathcal{SP}}\right)$ of strategy $\mathcal{SP}$ using a sample of size k can be calculated as follows:

$$\mathrm{E}\left(P_{n,k}^{\mathcal{SP}}\right) = \frac{4}{3}n + \frac{2}{\binom{n}{k}} \sum_{s \le \ell} \binom{s}{(k-2)/3}\binom{m}{(k-2)/3}\binom{\ell}{(k-2)/3} \cdot s + o(n). \tag{5.2}$$

Unfortunately, we could not find a closed form of $\mathrm{E}\left(P_{n,k}^{\mathcal{SP}}\right)$. Some calculated values obtained via Maple® in which classical and dual-pivot quicksort with strategy $\mathcal{SP}$ use the same sample size can be found in Table 5.1. These values clearly indicate that starting from a sample of size 5, asymptotically, classical quicksort has a smaller average comparison count than dual-pivot quicksort. This raises the question whether dual-pivot quicksort is inferior to classical quicksort using the median-of-k strategy with regard to minimizing the average comparison count.

5.3. Optimal Segment Sizes for Dual-Pivot Quicksort

It is known from, e. g., [MR01] that for classical quicksort in which the pivot is chosen as the median of a fixed-sized sample, the leading term of the average comparison count converges with increasing sample size to the lower bound of $(1/\ln 2) \cdot n \ln n = 1.4426..n \ln n$. Nebel and Wild observed in [NW14] that this is not the case for Yaroslavskiy's algorithm, which makes at least $1.4931n \ln n + o(n \ln n)$ comparisons on average. In this section, we will show how to match the lower bound for comparison-based sorting algorithms with a dual-pivot approach.

We study the following setting, which was considered in [MR01; NW14] as well. We assume

that for a random input of n elements[1] we can choose (for free) two pivots w.r.t. a vector $\vec{\tau} = (\tau_1, \tau_2, \tau_3)$ such that the input contains exactly $\tau_1 n$ small elements, $\tau_2 n$ medium elements, and $\tau_3 n$ large elements. Furthermore, we consider the (simple) classification strategy $\mathcal{L}$: *"Always compare with the larger pivot first."*

The following lemma says that this strategy achieves the minimum possible average comparison count for comparison-based sorting algorithms, $1.4426..n \ln n$, when setting $\tau = (\frac{1}{4}, \frac{1}{4}, \frac{1}{2})$.

Lemma 5.3.1
Let $\vec{\tau} = (\tau_1, \tau_2, \tau_3)$ with $0 < \tau_i < 1$ and $\sum_i \tau_i = 1$, for $i \in \{1, 2, 3\}$. Assume that for each input size n we can choose two pivots such that there are exactly $\tau_1 \cdot n$ small, $\tau_2 \cdot n$ medium, and $\tau_3 \cdot n$ large elements. Then the comparison count of strategy $\mathcal{L}$ is

$$p^{\vec{\tau}}(n) \sim \frac{1 + \tau_1 + \tau_2}{-\sum_{1 \le i \le 3} \tau_i \ln \tau_i} n \ln n.$$

This value is minimized for $\vec{\tau}^* = (1/4, 1/4, 1/2)$ giving

$$p^{\vec{\tau}^*}(n) \sim \left(\frac{1}{\ln 2}\right) n \ln n = 1.4426..n \ln n.$$

Proof. On an input consisting of n elements, strategy $\mathcal{L}$ makes $n + (\tau_1 + \tau_2)n$ comparisons. Thus, the comparison count of strategy $\mathcal{L}$ follows the recurrence

$$p^{\vec{\tau}}(n) = n + (\tau_1 + \tau_2)n + p^{\vec{\tau}}(\tau_1 \cdot n) + p^{\vec{\tau}}(\tau_2 \cdot n) + p^{\vec{\tau}}(\tau_3 \cdot n).$$

Using the Discrete Master Theorem from [Rou01, Theorem 2.3, Case (2.1)], we obtain the following solution for this recurrence:

$$p^{\vec{\tau}}(n) \sim \frac{1 + \tau_1 + \tau_2}{-\sum_{i=1}^{3} \tau_i \ln \tau_i} n \ln n.$$

Using Maple®, one finds that $p^{\vec{\tau}}$ is minimized for $\vec{\tau}^* = (\frac{1}{4}, \frac{1}{4}, \frac{1}{2})$, giving $p^{\vec{\tau}^*}(n) \sim 1.4426..n \ln n$. □

The reason why strategy $\mathcal{L}$ with this particular choice of pivots achieves the lower bound is simple: it makes (almost) the same comparisons as does classical quicksort using the median of the input as pivot. On an input of length n, strategy $\mathcal{L}$ makes $3/2n$ key comparisons and then makes three recursive calls to inputs of length $n/4$, $n/4$, $n/2$. On an input of length n, classical quicksort using the median of the input as pivot makes n comparisons to split the input into two subarrays of length $n/2$. Now consider only the recursive call to the left subarray. After

[1] We disregard the two pivots in the following discussion.

$n/2$ comparisons, the input is split into two subarrays of size $n/4$ each. Now there remain two recursive calls to two subarrays of size $n/4$, and one recursive call to a subarray of size $n/2$ (the right subarray of the original input), like in strategy $\mathcal{L}$. Since classical quicksort using the median of the input clearly makes $n \log n$ key comparisons, this bound must also hold for strategy $\mathcal{L}$.

6. Generalization to Multi-Pivot Quicksort

In this section we generalize our theory with respect to the average comparison count of dual-pivot quicksort algorithms to the case that we use more than two pivots. Some of this content, specifically the generalization of the classification tree, the comparison tree as a tool for classification, and all statements of theorems and lemmas already appeared in the Master's thesis of Pascal Klaue [Kla14], which was prepared under the guidance of the author. The details of the calculations for 3-pivot quicksort in Section 6.3 can be found in [Kla14, Section 4.3], too. I am a very thankful that he allowed me to include these parts in this thesis. The general setup has been simplified a little and the proofs presented here differ significantly from the ones given in [Kla14]. (We use a general concentration argument similar to Lemma 3.2.2, while in [Kla14] specific concentration arguments were used.) Moreover, we solve the cost recurrence for k-pivot quicksort directly, and give a proof of Theorem 6.4.2 (which was only conjectured in [Kla14]). Also, Section 6.5 is new. The topics presented here are part of a future publication of the author, Martin Dietzfelbinger, and Pascal Klaue, which is in preparation.

6.1. General Setup

We assume that the input is a random permutation $(e_1, \dots, e_n)$ of $\{1, \dots, n\}$. Let $k \geq 1$ be an integer. The method "k-pivot quicksort" works as follows: If $n \leq k$ then sort the input directly. For $n > k$, sort the first k elements such that $e_1 < e_2 < \ldots < e_k$ and set $p_1 = e_1, \dots, p_k = e_k$. In the *partition step*, the remaining $n - k$ elements are split into $k + 1$ groups $\mathrm{A}_0, \dots, \mathrm{A}_k$, where an element x belongs to group A_h if $p_h < x < p_{h+1}$, see Figure 6.1. (For the ease of discussion, we set $p_0 = 0$ and $p_{k+1} = n + 1$.) The groups $\mathrm{A}_0, \dots, \mathrm{A}_k$ are then sorted recursively. We never compare two non-pivot elements against each other. This preserves the randomness in the groups $\mathrm{A}_0, \dots, \mathrm{A}_k$.

Solving the Recurrence for k-Pivot Quicksort. Let $k \geq 1$ be fixed. As for dual-pivot quicksort, we let C_n and P_n denote the random variables that count the comparisons made for sorting and partitioning, respectively.[1] The total cost of sorting inputs of length $n' \leq k$ in the recursion is $O(n)$. We shall ignore this linear term in the following, assuming the cost for the base cases is 0. The average comparison count of k-pivot quicksort clearly obeys the following recurrence.

[1] We omit the dependency on k for random variables in the notation.

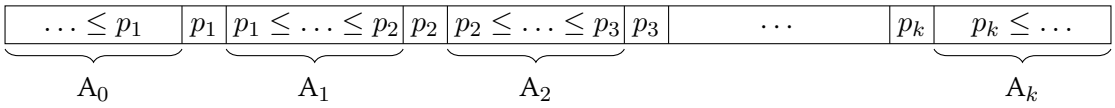

Figure 6.1.: Result of the partition step in k-pivot quicksort using pivots $p_1, \ldots, p_k$.

$$\mathrm{E}(C_n) = \mathrm{E}(P_n) + \frac{1}{\binom{n}{k}} \sum_{a_0+\ldots+a_k=n-k} (\mathrm{E}(C_{a_0}) + \ldots + \mathrm{E}(C_{a_k}))\,, \text{ with } E(C_n) = 0 \text{ for } n \le k.$$

We collect terms with a common factor $\mathrm{E}(C_\ell)$, for $0 \le \ell \le n-k$. Fix $\ell \in \{0, \ldots, n-k\}$. There are $k+1$ ways of choosing $j \in \{0, \ldots, k\}$ with $a_j = \ell$, and if j is fixed there are exactly $\binom{n-\ell-1}{k-1}$ ways to choose the other segment sizes $a_i, i \neq j$, such that $a_0 + \ldots + a_k = n-k$. (Note the equivalence between segment sizes and binary strings of length $n-\ell-1$ with exactly $k-1$ ones.) Thus, we conclude that

$$\mathrm{E}(C_n) = \mathrm{E}(P_n) + \frac{1}{\binom{n}{k}} \sum_{\ell=0}^{n-k} (k+1) \binom{n-\ell-1}{k-1} \mathrm{E}(C_\ell), \tag{6.1}$$

which was also observed in [Ili14]. (This generalizes the well known formula $\mathrm{E}(C_n) = n - 1 + 2/n \cdot \sum_{0 \le \ell \le n-1} \mathrm{E}(C_\ell)$ for classical quicksort, the formulas for $k = 2$ from, e. g., [WN12] or Section 3, and $k = 3$ from [Kus+14].)

The Continuous Master Theorem of Roura [Rou01] can again be applied to solve this recurrence. For partitioning cost $\mathrm{E}(P_n) = a \cdot n + O(1)$, for constant a, the solution of this recurrence can be found in [Hen91] and in [Ili14].

Theorem 6.1.1
Let $\mathcal{A}$ be a k-pivot quicksort algorithm which has for each subarray of length n partitioning cost $\mathrm{E}(P_n) = a \cdot n + o(n)$. Then

$$\mathrm{E}(C_n) = \frac{1}{\mathcal{H}_{k+1} - 1} an \ln n + o(n \ln n), \tag{6.2}$$

where $\mathcal{H}_{k+1} = \sum_{i=1}^{k+1} (1/i)$ is the $(k+1)$-st harmonic number.

Proof. Recall the statement of the Continuous Master Theorem, *cf.* Theorem 3.1.1. Recurrence (6.1) has weight

$$w_{n,j} = \frac{(k+1) \cdot k \cdot (n-j-1) \cdot \ldots \cdot (n-j-k+1)}{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}.$$

We define the shape function $w(z)$ as suggested in [Rou01] by

$$w(z) = \lim_{n\to\infty} n \cdot w_{n,zn} = (k+1)k(1-z)^{k-1}.$$

Using the Binomial theorem, we note that for all $z \in [0,1]$

$$\begin{aligned}|n \cdot w_{n,zn} - w(z)| &\leq k \cdot (k+1) \cdot \left| \frac{(n-zn)^{k-1}}{(n-k)^{k-1}} - (1-z)^{k-1} \right| \\ &\leq k \cdot (k+1) \cdot \left| (1-z)^{k-1} \cdot \left(1 + O\left(n^{-1}\right)\right) - (1-z)^{k-1} \right| = O(n^{-1}),\end{aligned}$$

and

$$\begin{aligned}|n \cdot w_{n,zn} - w(z)| &\leq k \cdot (k+1) \cdot \left| \frac{(n-zn-k)^{k-1}}{n^{k-1}} - (1-z)^{k-1} \right| \\ &\leq k \cdot (k+1) \cdot \left| \frac{(n-zn)^{k-1}}{n^{k-1}} + O\left(n^{-1}\right) - (1-z)^{k-1} \right| = O\left(n^{-1}\right).\end{aligned}$$

Now we have to check (3.2) to see whether the shape function is suitable. We calculate:

$$\begin{aligned}&\sum_{j=0}^{n-1} \left| w_{n,j} - \int_{j/n}^{(j+1)/n} w(z)\, \mathrm{d}z \right| \\ &= \sum_{j=0}^{n-1} \left| \int_{j/n}^{(j+1)/n} n \cdot w_{n,j} - w(z)\, \mathrm{d}z \right| \\ &\leq \sum_{j=0}^{n-1} \frac{1}{n} \max_{z\in[j/n,(j+1)/n]} |n \cdot w_{n,j} - w(z)| \\ &\leq \sum_{j=0}^{n-1} \frac{1}{n} \left(\max_{z\in[j/n,(j+1)/n]} |w(j/n) - w(z)| + O\left(n^{-1}\right) \right) \\ &\leq \sum_{j=0}^{n-1} \frac{1}{n} \left(\max_{|z-z'|\leq 1/n} \left|w(z) - w(z')\right| + O\left(n^{-1}\right) \right) \\ &\leq \sum_{j=0}^{n-1} \frac{k(k+1)}{n} \left(\max_{|z-z'|\leq 1/n} \left|(1-z)^{k-1} - (1-z-1/n)^{k-1}\right| + O\left(n^{-1}\right) \right) \\ &\leq \sum_{j=0}^{n-1} O\left(n^{-2}\right) = O\left(n^{-1}\right),\end{aligned}$$

where we again used the Binomial theorem in the last two lines.

Thus, w is a suitable shape function. By calculating

$$H := 1 - k(k+1) \int_0^1 z(1-z)^{k-1} \, \mathrm{d}z = 0,$$

we conclude that the second case of Theorem 3.1.1 applies for our recurrence. Consequently, we have to calculate

$$\hat{H} := -k(k+1) \int_0^1 z(1-z)^{k-1} \ln z \, \mathrm{d}z.$$

To estimate the integral, we use the following identity for Harmonic numbers due to Sofo [Sof12, Lemma 7]

$$\mathcal{H}_k = -k \int_0^1 z^{k-1} \ln z \, \mathrm{d}z.$$

By symmetry, we may calculate

$$\begin{aligned} -k(k+1)\int_0^1 z(1-z)^{k-1} \ln z \, \mathrm{d}z &= -k(k+1) \int_0^1 (1-z) z^{k-1} \ln(1-z) \, \mathrm{d}z \\ &= -k(k+1) \left(\int_0^1 z^{k-1} \ln(1-z) \, \mathrm{d}z - \int_0^1 z^k \ln(1-z) \, \mathrm{d}z \right) \\ &= -k(k+1) \left(\frac{\mathcal{H}_{k+1}}{k+1} - \frac{\mathcal{H}_k}{k} \right) = \mathcal{H}_{k+1} - 1. \end{aligned}$$

□

As in the case of dual-pivot quicksort, very small subarrays of size $n_0 \leq n^{1/\ln\ln n}$ occurring in the recursion require special care for some algorithms that are described in the next section. However, similar statements to the ones given in Section 3.1 ("Handling Small Subarrays") show that the total cost of sorting such subarrays in the recursion is bounded by $o(n \ln n)$ in the multi-pivot case as well.

The Classification Problem. As before, it suffices to study the *classification problem*: Given a random permutation $(e_1, \ldots, e_n)$ of $\{1, \ldots, n\}$, choose the pivots $p_1, \ldots, p_k$ and classify each of the remaining $n - k$ elements as belonging to the group $A_0, A_1, \ldots, A_{k-1}$, or A_k. In this setting, we have $a_i := |A_i| = p_{i+1} - p_i - 1$ for $i \in \{0, \ldots, k\}$.

Next, we will introduce our algorithmic model for solving the classification problem. Of course, this model will share a lot of similarities with the classification tree of dual-pivot quicksort. The main difference to dual-pivot quicksort, where an element was classified by either comparing it to p or to q first, is the classification of a single element.

Algorithmically, the classification of a single element x with respect to the pivots $p_1, \dots, p_k$ is done by using a *comparison tree* t. In a comparison tree the leaf nodes are labeled $\mathrm{A}_0, \dots, \mathrm{A}_k$, from left to right, the inner nodes are labeled $\mathrm{p}_1, \dots, \mathrm{p}_k$, in inorder. Figure 6.2 depicts a comparison tree for 5 pivots. A comparison tree with a particular pivot choice $p_1, \dots, p_k$ gives rise to a binary search tree. *Classifying* an element means searching for this element in the search tree. The group to which the element belongs is the label of the leaf reached in that way.

A *classification strategy* is formally described as a *classification tree* as follows. A classification tree is a $(k+1)$-way tree with a root and $n-k$ levels of inner nodes as well as one leaf level. Each inner node v has two labels: an index $i(v) \in \{k+1, \dots, n\}$, and a comparison tree $t(v)$. The element $e_{i(v)}$ is classified using the comparison tree $t(v)$. The $k+1$ edges out of a node are labeled $0, \dots, k$, resp., representing the outcome of the classification as belonging to group $\mathrm{A}_0, \dots, \mathrm{A}_k$, respectively. On each of the $(k+1)^{n-k}$ paths each index from $\{k+1, \dots, n\}$ occurs exactly once. An input $(e_1, \dots, e_n)$ determines a path in the classification tree in the obvious way: sort the pivots, then use the classification tree to classify $e_{k+1}, \dots, e_n$. The groups to which the elements in the input belong can then be read off from the nodes and edges along the path from the root to a leaf in the classification tree.

To fix some more notation, for each node v, and for $h \in \{0, \dots, k\}$, we let a_h^v be the number of edges labeled "h" on the path from the root to v. Furthermore, let $C_{h,i}$ denote the random variable which counts the number of elements classified as belonging to group A_h, for $h \in \{0, \dots k\}$, in the first i levels, for $i \in \{0, \dots, n-k\}$, i. e., $C_{h,i} = a_h^v$ when v is the node on level i of the classification tree reached for an input. Analogously to the dual-pivot case, in many proofs we will need that $C_{h,i}$ is not far away from its expectation $a_h/(n-i-k)$ for fixed pivot choices. As in Section 3, one can use the *method of averaged bounded differences* to show concentration despite dependencies between tests.

Lemma 6.1.2
Let the pivots $p_1, \dots, p_k$ be fixed. Let $C_{h,i}$ be defined as above. Then for each h with $h \in \{0, \dots, k\}$ and for each i with $1 \le i \le n-k$ we have that

$$\Pr\left(|C_{h,i} - \mathrm{E}(C_{h,i})| > n^{2/3}\right) \le 2\exp\left(-n^{1/3}/2\right).$$

Proof. The proof is analogous to the proof of Lemma 3.2.2 in the dual-pivot quicksort case. □

6.2. The Average Comparison Count for Partitioning

In this section, we will obtain a formula for the average comparison count of an arbitrary classification tree. We make the following observations for all classification strategies: We need $k \log k = O(1)$ comparisons to sort $e_1, \dots, e_k$, i. e., to determine the k pivots $p_1, \dots, p_k$ in

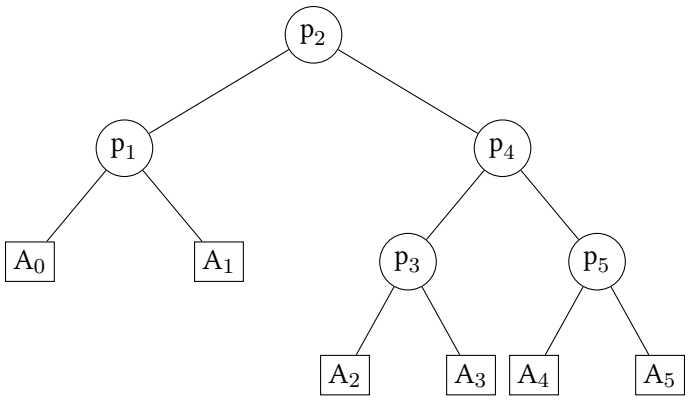

Figure 6.2.: A comparison tree for 5 pivots.

order. If an element x belongs to group A_i, it must be compared to p_i and p_{i+1}. (Of course, no real comparison takes place against p_0 and p_{k+1}.) On average, this leads to $\frac{2k}{k+1}n + O(1)$ comparisons—regardless of the actual classification strategy.

For the following paragraphs, we fix a classification strategy, i. e., a classification tree T. Let v be an arbitrary inner node of T.

If $e_{i(v)}$ belongs to group A_h, exactly $\mathrm{depth}_{t(v)}(\mathrm{A}_h)$ comparisons are made to classify the element. We let C_v^T denote the number of comparisons that take place in node v during classification. Clearly, $P_n^T = \sum_{v \in T} C_v^T$. For the average classification cost $\mathrm{E}(P_n^T)$ we get:

$$\mathrm{E}(P_n^T) = \frac{1}{\binom{n}{k}} \sum_{1 \le p_1 < p_2 < \ldots < p_k \le n} \mathrm{E}(P_n^T \mid p_1, \ldots, p_k).$$

We define $p^v_{p_1,\ldots,p_k}$ to be the probability that node v is reached if the pivots are $p_1, \ldots, p_k$. We may write:

$$\mathrm{E}(P_n^T \mid p_1, \ldots, p_k) = \sum_{v \in T} \mathrm{E}(C_v^T \mid p_1, \ldots, p_k) = \sum_{v \in T} p^v_{p_1,\ldots,p_k} \cdot \mathrm{E}(C_v^T \mid p_1, \ldots, p_k,\ v \text{ reached}). \tag{6.3}$$

For a comparison tree t and group sizes $a'_0, \ldots, a'_k$, we define the *cost* of t on these group sizes as the number of comparisons it makes for classifying an input with these group sizes, i. e.,

$$\mathrm{cost}^t(a'_0, \ldots, a'_k) := \sum_{0 \le i \le k} \mathrm{depth}_t(\mathrm{A}_i) \cdot a'_i.$$

Furthermore, we define its average cost $c^t_{\text{avg}}(a'_0, \ldots, a'_k)$ as follows:

$$c^t_{\text{avg}}(a'_0, \ldots, a'_k) := \frac{\text{cost}^t(a'_0, \ldots, a'_k)}{\sum_{0 \le i \le k} a'_i}. \tag{6.4}$$

Under the assumption that node v is reached and that the pivots are $p_1, \ldots, p_k$, the probability that the element $e_{i(v)}$ belongs to group A_h is exactly $(a_h - a^v_h)/(n - k - \text{level}(v))$, for each $h \in \{0, \ldots, k\}$, with a^v_h defined as in Section 6.1. Summing over all groups, we get

$$\text{E}(C^T_v \mid p_1, \ldots, p_k,\ v \text{ reached}) = c^{t(v)}_{\text{avg}}(a_0 - a^v_0, \ldots, a_k - a^v_k).$$

Plugging this into (6.3) gives

$$\text{E}(P^T_n \mid p_1, \ldots, p_k) = \sum_{v \in T} p^v_{p_1,\ldots,p_k} \cdot c^{t(v)}_{\text{avg}}(a_0 - a^v_0, \ldots, a_k - a^v_k). \tag{6.5}$$

Remark 6.2.1. From (6.5) it is clear that the order in which elements are tested has no influence on the average classification cost of a classification strategy. So, we may always assume that the element tests are made in some fixed order, e. g., $e_{k+1}, \ldots, e_n$.

Let $\mathcal{T}_k$ be the set of all possible comparison trees. For each $t \in \mathcal{T}_k$, we define the random variable F^t that counts the number of times t is used during classification. For given $p_1, \ldots, p_k$, and for each $t \in \mathcal{T}_k$, we let

$$f^t_{p_1,\ldots,p_k} := \text{E}(F^t \mid p_1, \ldots, p_k) = \sum_{\substack{v \in T \\ t(v) = t}} p^v_{p_1,\ldots,p_k} \tag{6.6}$$

be the average number of times comparison tree t is used in T under the condition that the pivots are $p_1, \ldots, p_k$.

Now, if it was decided in each step by independent random experiments with the correct expectation $a_h/(n-k)$, for $0 \le h \le k$, whether an element belongs to group A_h or not, it would be clear that for each $t \in \mathcal{T}_k$ the contribution of t to the average classification cost is $f^t_{p_1,\ldots,p_k} \cdot c^t_{\text{avg}}(a_0, \ldots, a_k)$. We can prove that this intuition is true for all classification trees, excepting that one gets an additional $o(n)$ term due to the elements tested not being independent.

Lemma 6.2.2
Let T be a classification tree. Then

$$\text{E}(P^T_n) = \frac{1}{\binom{n}{k}} \sum_{1 \le p_1 < p_2 < \ldots < p_k \le n} \quad \sum_{t \in \mathcal{T}_k} f^t_{p_1,\ldots,p_k} \cdot c^t_{\text{avg}}(a_0, \ldots, a_k) + o(n).$$

Proof. Fix a set of pivots $p_1, \dots, p_k$. Using the definition of $f^t_{p_1,\dots,p_k}$ from (6.6), we can re-write (6.5) as follows:

$$\begin{aligned}
\mathrm{E}(P_n^T \mid p_1, \dots, p_k) &= \sum_{v \in T} p^v_{p_1,\dots,p_k} \cdot c^{t(v)}_{\text{avg}}(a_0 - a_0^v, \dots, a_k - a_k^v) \\
&= \sum_{t \in \mathcal{T}_k} f^t_{p_1,\dots,p_k} \cdot c^t_{\text{avg}}(a_0, \dots, a_k) - \\
&\quad \sum_{v \in T} p^v_{p_1,\dots,p_k} \left(c^{t(v)}_{\text{avg}}(a_0, \dots, a_k) - c^{t(v)}_{\text{avg}}(a_0 - a_0^v, \dots, a_k - a_k^v) \right). \qquad (6.7)
\end{aligned}$$

For a node v in the classification tree, we say that v is *good* if

$$\left| c^{t(v)}_{\text{avg}}(a_0, \dots, a_k) - c^{t(v)}_{\text{avg}}(a_0 - a_0^v, \dots, a_k - a_k^v) \right| \le \frac{k^2}{n^{1/12}}.$$

Otherwise, v is called *bad*. By considering good and bad nodes in (6.7) separately, we obtain

$$\begin{aligned}
\mathrm{E}(P_n^T \mid p_1, \dots, p_k) &\le \sum_{t \in \mathcal{T}_k} f^t_{p_1,\dots,p_k} \cdot c^t_{\text{avg}}(a_0, \dots, a_k) + \sum_{\substack{v \in T \\ v \text{ is good}}} p^v_{p_1,\dots,p_k} \cdot \frac{k^2}{n^{1/12}} + \\
&\quad \sum_{\substack{v \in T \\ v \text{ is bad}}} p^v_{p_1,\dots,p_k} \cdot \left(c^{t(v)}_{\text{avg}}(a_0, \dots, a_k) - c^{t(v)}_{\text{avg}}(a_0 - a_0^v, \dots, a_k - a_k^v) \right) \\
&\le \sum_{t \in \mathcal{T}_k} f^t_{p_1,\dots,p_k} \cdot c^t_{\text{avg}}(a_0, \dots, a_k) + k \cdot \sum_{\substack{v \in T \\ v \text{ is bad}}} p^v_{p_1,\dots,p_k} + o(n) \\
&= \sum_{t \in \mathcal{T}_k} f^t_{p_1,\dots,p_k} \cdot c^t_{\text{avg}}(a_0, \dots, a_k) + \\
&\quad k \cdot \sum_{i=1}^{n-k} \Pr(\text{a bad node is reached on level } i) + o(n). \qquad (6.8)
\end{aligned}$$

It remains to bound the second summand of (6.8). We observe that

$$\begin{aligned}
&\left| c^{t(v)}_{\text{avg}}(a_0, \dots, a_k) - c^{t(v)}_{\text{avg}}(a_0 - a_0^v, \dots, a_k - a_k^v) \right| \\
&\qquad \le (k-1) \cdot \sum_{h=0}^{k} \left| \frac{a_h}{n-k} - \frac{a_h - a_h^v}{n - k - \text{level}(v)} \right| \\
&\qquad \le (k-1) \cdot (k+1) \cdot \max_{0 \le h \le k} \left\{ \left| \frac{a_h}{n-k} - \frac{a_h - a_h^v}{n - k - \text{level}(v)} \right| \right\}.
\end{aligned}$$

Thus, by definition, whenever v is a bad node, there exists $h \in \{0, \dots, k\}$ such that

$$\left| \frac{a_h}{n-k} - \frac{a_h - a_h^v}{n-k-\text{level}(v)} \right| > \frac{1}{n^{1/12}}.$$

Recall from Section 6.1 the definition of the random variable $C_{h,i}$ and its connection to the values a_h^v on level i of the classification tree. For fixed $i \in \{1, \dots, n-k\}$, assume that the random variables $C_{h,i}, h \in \{0, \dots, k\}$, are tightly concentrated around their expectation. (From Lemma 6.1.2 we know that this is true with very high probability). For each $h \in \{0, \dots, k\}$, and each level $i \in \{1, \dots, n-k\}$ we calculate

$$\left| \frac{a_h}{n-k} - \frac{a_h - C_{h,i}}{n-k-i} \right| \leq \left| \frac{a_h}{n-k} - \frac{a_h(1-i/(n-k))}{n-k-i} \right| + \left| \frac{n^{2/3}}{n-k-i} \right| = \frac{n^{2/3}}{n-k-i}.$$

That means that for each of the first $i \leq n - n^{3/4}$ levels with very high probability we are in a *good node* on level i, because the deviation from the ideal case that the element test on level i reveals an "A_h"-element with probability $a_h/(n-k)$ is smaller than $n^{2/3}/(n-k-i) \leq n^{2/3}/n^{3/4} = 1/n^{1/12}$. Thus, for the first $n - n^{3/4}$ levels the contribution of the sums of the probabilities of bad nodes in (6.8) is $o(n)$. For the last $n^{3/4}$ levels of the tree, we use that the contribution of the probabilities that we reach a bad node on level i is at most 1 for a fixed level.

This shows that the second summand in (6.8) is $o(n)$. The lemma now follows from averaging over all possible pivot choices. A lower bound follows in an analogous way. □

6.3. Example: 3-pivot Quicksort

Here we study some variants of 3-pivot quicksort algorithms in the light of Lemma 6.2.2. This paradigm got recent attention by the work of Kushagra *et al.* [Kus+14], who provided evidence that in practice a 3-pivot quicksort algorithm might be faster than Yaroslavskiy's dual-pivot quicksort algorithm.

In 3-pivot quicksort, we might choose from five different comparison trees. These trees, together with their comparison cost, are depicted in Figure 6.3. We will study the average comparison count of three different strategies in an artificial setting: We assume, as in the analysis, that our input is a permutation of $\{1, \dots, n\}$. So, after choosing the pivots the algorithm knows the exact group sizes in advance. Generalizing this strategy is a topic of the subsequent section.

All considered strategies will follow the same idea: After choosing the pivots, they will check which comparison tree has the smallest average cost for the group sizes found in the input; this tree is used for all classifications. The difference will be in the set of comparison trees we allow the algorithm to choose from. In the next section we will explain why deviating from such a strategy, i. e., using different trees during the classification for fixed group sizes, does not help for decreasing the average comparison count up to lower order terms, as already noticed for dual-pivot quicksort with the optimal strategy $\mathcal{N}$.

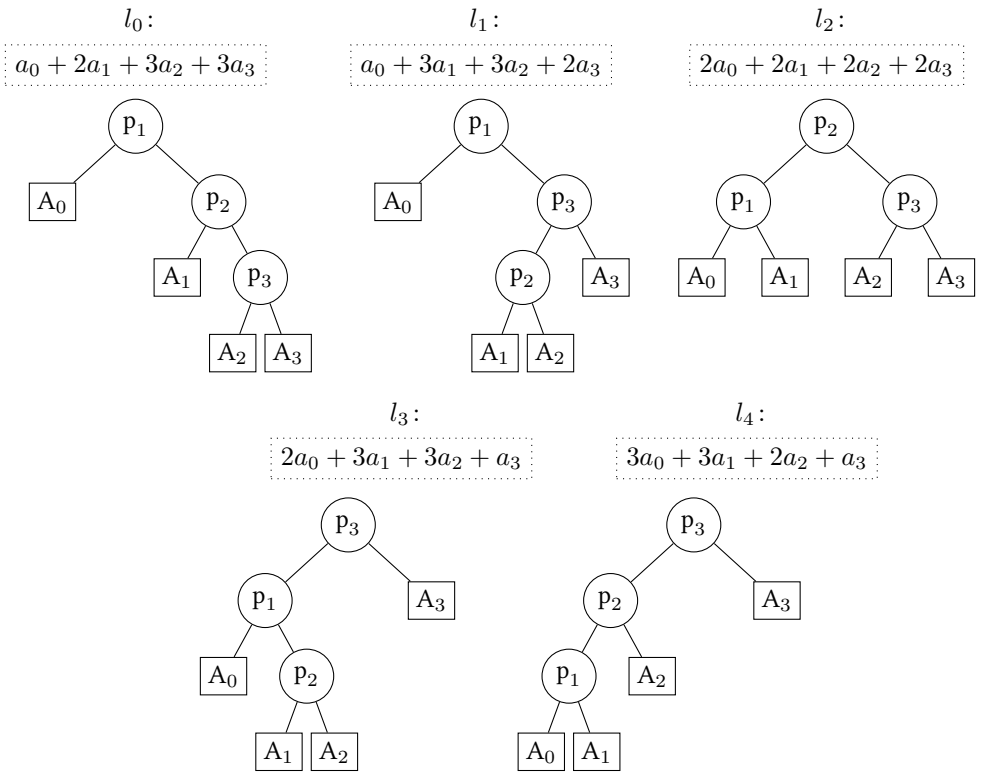

Figure 6.3.: The different comparison trees for 3-pivot quicksort with their comparison cost (dotted boxes, only displaying the numerator).

The Symmetric Strategy. In the algorithm of [Kus+14], the symmetric comparison tree l_2 is used in the classification of each element. Using Lemma 6.2.2, we get[2]

$$\begin{aligned}\mathrm{E}(P_n) &= \frac{1}{\binom{n}{3}} \sum_{a_0+a_1+a_2+a_3=n-3} (2a_0 + 2a_1 + 2a_2 + 2a_3) + o(n) \\ &= 2n + o(n).\end{aligned}$$

Using Theorem 6.1.1, we conclude that

$$\mathrm{E}(C_n) = 24/13 n \ln n + o(n \ln n) \approx 1.846 n \ln n + o(n \ln n),$$

[2]Of course, $\mathrm{E}(P_n) = 2(n-3)$, since each classification makes exactly two comparisons.

as known from [Kus+14]. This improves over classical quicksort ($2n \ln n + O(n)$ comparisons on average), but is worse than optimal dual-pivot quicksort ($1.8n \ln n + O(n)$ comparisons on average, see Section 4) or median-of-3 quicksort ($1.714n \ln n + O(n)$ comparisons on average, see Section 5).

Using Three Trees. Here we restrict our algorithm to choose only among the comparison trees $\{l_1, l_2, l_3\}$. The computation of a cost-minimal comparison tree is then simple: Suppose that the segment sizes are $a_0, \dots, a_3$. If $a_0 > a_3$ and $a_0 > a_1 + a_2$ then comparison tree l_1 has minimum cost. If $a_3 \geq a_0$ and $a_3 > a_1 + a_2$ then comparison tree l_3 has minimum cost. Otherwise l_2 has minimum cost.

Using Lemma 6.2.2, the average partition cost with respect to this set of comparison trees can be calculated (using Maple®) as follows[3]:

$$\begin{aligned} \mathrm{E}(P_n) &= \frac{1}{\binom{n}{3}} \sum_{a_0+\dots+a_3=n-3} \min \left\{ \begin{smallmatrix} a_0+2a_1+3a_2+3a_3, 2a_0+2a_1+2a_2+2a_3, \\ 2a_0+3a_1+3a_2+1a_3 \end{smallmatrix} \right\} + o(n) \\ &= \frac{17}{9} n + o(n). \end{aligned}$$

This yields the following average comparison cost:

$$\mathrm{E}(C_n) = \frac{68}{39} n \ln n + o(n \ln n) \approx 1.744 n \ln n + o(n \ln n).$$

Using All Five Trees. Now we let our strategies choose among all five trees. Using Lemma 6.2.2 and the average cost for all trees from Figure 6.3, we calculate (using Maple®)

$$\begin{aligned} \mathrm{E}(P_n) &= \frac{1}{\binom{n}{3}} \sum_{a_0+\dots+a_3=n-3} \min \left\{ \begin{smallmatrix} a_0+2a_1+3a_2+3a_3, a_0+3a_1+3a_2+2a_3, \\ 2a_0+2a_1+2a_2+2a_3, 2a_0+3a_1+3a_2+a_3, \\ 3a_0+3a_1+2a_2+a_3 \end{smallmatrix} \right\} + o(n) \\ &= \frac{133}{72} n + o(n). \end{aligned} \tag{6.9}$$

This yields the following average comparison cost:

$$\mathrm{E}(C_n) = \frac{133}{78} n \ln n + o(n \ln n) \approx 1.705 n \ln n + o(n \ln n),$$

which is—as will be explained in the next section—the lowest possible average comparison count one can achieve by picking three pivots directly from the input. So, using three pivots gives a

[3] There was a lot of tweaking and manual work necessary before Maple® was able to find a closed form of the sum. Details of these calculations can be found in [Kla14].

slightly lower average comparison count than quicksort using the median of three elements as the pivot and makes about $0.1n \ln n$ fewer comparisons as comparison-optimal dual-pivot quicksort.

6.4. (Asymptotically) Optimal Classification Strategies

In this section, we will discuss the obvious generalizations for the optimal strategies $\mathcal{O}, \mathcal{C}, \mathcal{N}$, and $\mathcal{SP}$ for dual-pivot quicksort to k-pivot quicksort. In difference to Section 4, we first present the "improper" classification strategy ($\mathcal{O}$ and $\mathcal{N}$, respectively) that uses different classification trees depending on the pivot choices and then directly show its connection with the implementation of that strategy ($\mathcal{C}$ and $\mathcal{SP}$, respectively).

Since all these strategies need to compute cost-minimal comparison trees, this section starts with a short discussion of algorithms for this problem. Then, we discuss the four different strategies.

6.4.1. Choosing an Optimal Comparison Tree

For optimal k-pivot quicksort algorithms it is of course necessary to devise an algorithm that can compute an optimal comparison tree for group sizes $a_0, \dots, a_k$, i. e., a comparison tree that minimizes (6.4). It is well known that the number of binary search trees with k inner nodes equals the k-th Catalan number, which is approximately $4^k / \left((k+1)\sqrt{\pi k}\right)$. Choosing an optimal tree is a standard application of dynamic programming, and is known from textbooks as "choosing an optimum binary search tree", see, e. g., [Knu73]. The algorithm runs in time and space $O(k^2)$.

6.4.2. The Optimal Classification Strategy and its Algorithmic Variant

Here, we consider the following strategy $\mathcal{O}_k$:[4] *Given $a_0, \dots, a_k$, the comparison tree $t(v)$ is one that minimizes cost$^t(a_0 - a_0^v, \dots, a_k - a_k^v)$ over all comparison trees t.*

Although being unrealistic, since the exact group sizes $a_0, \dots, a_k$ are in general unknown to the algorithm, strategy $\mathcal{O}_k$ is the optimal classification strategy, i. e., it minimizes the average comparison count.

Theorem 6.4.1
Strategy $\mathcal{O}_k$ is optimal for each k.

Proof. Strategy $\mathcal{O}_k$ chooses for each node v in the classification tree the comparison tree that minimizes the average cost in (6.5). So, it minimizes each term of the sum, and thus minimizes the whole sum in (6.5). □

[4] For all strategies we just say which comparison tree is used in a given node of the classification tree, since the test order is arbitrary (see Remark 6.2.1).

We remark here that in difference to strategy $\mathcal{O}_2$, *cf.* Theorem 4.2.1, we could not find an argument that yields the average comparison count of strategy $\mathcal{O}_k$ for $k \geq 3$. This is an important open question.

As in the dual-pivot case there exist other strategies whose average comparison count differs by at most $o(n)$ from the average comparison count of $\mathcal{O}_k$. Again, we call such strategies *asymptotically optimal*. Strategy $\mathcal{C}_k$ is an algorithmic variant of $\mathcal{O}_k$. It works as follows: *The comparison tree $t(v)$ is one that minimizes cost$^t(a_0^v, \ldots, a_k^v)$ over all comparison trees t.*

Theorem 6.4.2
Strategy $\mathcal{C}_k$ is asymptotically optimal for each k.

Proof. By Remark 6.2.1, assume that strategy $\mathcal{O}_k$ classifies elements in the order $e_{k+1}, \ldots, e_n$, while strategy $\mathcal{C}_k$ classifies them in reversed order, i. e., $e_n, \ldots, e_{k+1}$. Then the comparison tree that is used by $\mathcal{C}_k$ for element e_{k+i} is the same as the one that $\mathcal{O}_k$ uses for element e_{k+i+1}, for $i \in \{1, \ldots, n-k-1\}$. Let C_i^T denote the number of comparisons used to classify e_{k+i} with classification tree T.

Fix the pivots $p_1, \ldots, p_k$, let T denote the classification tree of strategy $\mathcal{O}_k$, and let T' denote the classification tree of strategy $\mathcal{C}_k$. Fix some integer $i \in \{1, \ldots, n-k-1\}$. Fix an arbitrary sequence $(a'_0, \ldots, a'_k) \in \mathbb{N}^{k+1}$, for $a'_h \leq a_h, h \in \{0, \ldots, k\}$, with $a'_0 + \ldots + a'_k = i - 1$ and $|a'_h - (i-1) \cdot a_h/(n-k)| \leq n^{2/3}$. Assume that the elements $e_{k+1}, \ldots, e_{k+i-1}$ have group sizes $a'_0, \ldots, a'_k$, and let t be a comparison tree with minimal cost w.r.t. $(a_0 - a'_0, \ldots, a_k - a'_k)$. For fixed $i \in \{2, \ldots, n-k-1\}$, we calculate:

$$
\begin{aligned}
&\left|\mathrm{E}\left(C_i^T \mid a_0, \ldots, a_k\right) - \mathrm{E}\left(C_{i-1}^{T'} \mid a_0, \ldots, a_k\right)\right| \\
&= \left|c^t_{\text{avg}}(a_0 - a'_0, \ldots, a_k - a'_k) - c^t_{\text{avg}}(a'_0, \ldots, a'_k)\right| \\
&= \left|\frac{\sum_{h=0}^k \mathrm{depth}(\mathrm{A}_h) \cdot (a_h - a'_h)}{n-k-i} - \frac{\sum_{h=0}^k \mathrm{depth}(\mathrm{A}_h) \cdot a'_h}{i}\right| \\
&\leq \left|\frac{\sum_{h=0}^k \mathrm{depth}(\mathrm{A}_h)(a_h - \frac{a_h \cdot i}{n-k})}{n-k-i} - \frac{\sum_{h=0}^k \mathrm{depth}(\mathrm{A}_h)\left(i \cdot \frac{a_h}{n-k}\right)}{i}\right| + \frac{k^2 \cdot n^{2/3}}{n-i-k} + \frac{k^2 \cdot n^{2/3}}{i} \\
&= \frac{k^2 \cdot n^{2/3}}{n-i-k} + \frac{k^2 \cdot n^{2/3}}{i}.
\end{aligned}
$$

Since the concentration argument of Lemma 6.1.2 holds with very high probability, the difference between the average comparison count of element e_{k+i} (for $\mathcal{O}_k$) and e_{k+i-1} (for $\mathcal{C}_k$) is at most

$$\frac{k^2 \cdot n^{2/3}}{n-i-k} + \frac{k^2 \cdot n^{2/3}}{i} + o(1).$$

Thus, the difference of the average comparison count over all elements $e_{k+i}, \dots, e_{k+j}$, $i \geq n^{3/4}, j \leq n - n^{3/4}$, is at most $o(n)$. For elements outside of this range, the difference in the average comparison count is at most $2k \cdot n^{3/4}$. So, the total difference of the comparison count between strategy $\mathcal{O}_k$ and strategy $\mathcal{C}_k$ is at most $o(n)$. □

This shows that the optimal strategy $\mathcal{O}_k$ can be approximated by an actual algorithm that makes an error of up to $o(n)$, which sums up to an error term of $o(n \ln n)$ over the whole recursion. We have seen in the dual-pivot case that the difference between $\mathcal{O}_2$ and $\mathcal{C}_2$ is $O(\log n)$. It remains an open question to prove tighter bounds than $o(n)$ for the difference of the average comparison count of $\mathcal{O}_k$ and $\mathcal{C}_k$ for $k \geq 3$.

6.4.3. An Oblivious Strategy and its Algorithmic Variant

Now we turn to strategy $\mathcal{N}_k$: *Given $a_0, \dots, a_k$, the comparison tree $t(v)$ used at node v is one that minimizes cost$^t(a_0, \dots, a_k)$ over all comparison trees t.*

Strategy $\mathcal{N}_k$ uses a fixed comparison tree for all classifications for given group sizes, but has to know these sizes in advance.

Theorem 6.4.3
Strategy $\mathcal{N}_k$ is asymptotically optimal for each k.

Proof. According to Lemma 6.2.2 the average comparison count is determined up to lower order terms by the parameters $f^t_{p_1,\dots,p_k}$, for each $t \in \mathcal{T}_k$. For each $p_1, \dots, p_k$, strategy $\mathcal{N}_k$ chooses the comparison tree which minimizes the average cost. According to Lemma 6.2.2, this is optimal up to an $o(n)$ term. □

We will now describe how to implement strategy $\mathcal{N}_k$ by using sampling. Strategy $\mathcal{SP}_k$ works as follows: Let $t_0 \in \mathcal{T}_k$ be an arbitrary comparison tree. After the pivots are chosen, inspect the first $n^{3/4}$ elements and classify them using t_0. Let $a'_0, \dots, a'_k$ denote the number of elements that belonged to $\mathrm{A}_0, \dots, \mathrm{A}_k$, respectively. Let t be an optimal comparison tree for $a'_0, \dots, a'_k$. Then classify each of the remaining elements by using t.

Theorem 6.4.4
Strategy $\mathcal{SP}_k$ is asymptotically optimal for each k.

Proof. Fix the k pivots $p_1, \dots, p_k$ and thus $a_0, \dots, a_k$. Let t^* be a comparison tree with minimal cost w.r.t. $a_0, \dots, a_k$.

According to Lemma 6.2.2, the average comparison count $\mathrm{E}(C_n^{\mathcal{SP}_k} \mid p_1, \dots, p_k)$ can be calculated as follows:

$$\mathrm{E}(C_n^{\mathcal{SP}_k} \mid p_1, \dots, p_k) = \sum_{t \in \mathcal{T}_k} f^t_{p_1,\dots,p_k} \cdot c^t_{\text{avg}}(a_0, \dots, a_k) + o(n).$$

Let $a'_0, \ldots, a'_k$ be the group sizes after inspecting $n^{3/4}$ elements. Let t be a comparison tree with minimal cost w.r.t. $a'_0, \ldots, a'_k$. We call t *good* if

$$c^t_{\text{avg}}(a_0, \ldots, a_k) - c^{t^*}_{\text{avg}}(a_0, \ldots, a_k) \leq \frac{2k}{n^{1/12}}, \text{ or equivalently}$$
$$\text{cost}^t(a_0, \ldots, a_k) - \text{cost}^{t^*}(a_0, \ldots, a_k) \leq 2kn^{11/12}, \tag{6.10}$$

otherwise we call t *bad*. We define good_t as the event that the sample yields a good comparison tree. We calculate:

$$\begin{aligned}
\mathrm{E}(C_n^{\mathcal{SP}_k} \mid p_1, \ldots, p_k) &= \sum_{\substack{t \in \mathcal{T}_k \\ t \text{ good}}} f^t_{p_1,\ldots,p_k} \cdot c^t_{\text{avg}}(a_0, \ldots, a_k) + \sum_{\substack{t \in \mathcal{T}_k \\ t \text{ bad}}} f^t_{p_1,\ldots,p_k} \cdot c^t_{\text{avg}}(a_0, \ldots, a_k) + o(n) \\
&\leq n \cdot c^{t^*}_{\text{avg}}(a_0, \ldots, a_k) + \sum_{\substack{t \in \mathcal{T}_k \\ t \text{ bad}}} f^t_{p_1,\ldots,p_k} \cdot c^t_{\text{avg}}(a_0, \ldots, a_k) + o(n) \\
&\leq n \cdot c^{t^*}_{\text{avg}}(a_0, \ldots, a_k) + k \cdot \sum_{\substack{t \in \mathcal{T}_k \\ t \text{ bad}}} f^t_{p_1,\ldots,p_k} + o(n).
\end{aligned} \tag{6.11}$$

Now we derive an upper bound for the second summand of (6.11). After the first $n^{3/4}$ classifications the algorithm will either use a good comparison tree or a bad comparison tree for the remaining classifications. $\Pr\left(\overline{\text{good}_t} \mid p_1, \ldots, p_k\right)$ is the ratio of nodes on each level from $n^{3/4}$ to $n - k$ of the classification tree of nodes labeled with bad trees (in the sense of (6.10)). Summing over all levels, the second summand of (6.11) is thus at most $k \cdot n \cdot \Pr\left(\overline{\text{good}_t} \mid p_1, \ldots, p_k\right)$.

Lemma 6.4.5
Conditioned on $p_1, \ldots, p_k$, good_t occurs with very high probability.

Proof. For each $i \in \{0, \ldots, k\}$, let a'_i be the random variable that counts the number of elements from the sample that belong to group A_i. According to Lemma 6.1.2, with very high probability we have that $|a'_i - \mathrm{E}(a'_i)| \leq n^{2/3}$, for each i with $0 \leq i \leq k$. By the union bound, with very high probability there is no a'_i that deviates by more than $n^{2/3}$ from its expectation $n^{-1/4} \cdot a_i$. We will now show that if this happens then the event good_t occurs. We obtain the following upper bound for an arbitrary comparison tree $t' \in \mathcal{T}_k$:

$$\begin{aligned}
\text{cost}^{t'}\left(a'_0, \ldots, a'_k\right) &= \sum_{0 \leq i \leq k} \text{depth}_{t'}(\mathrm{A}_i) \cdot a'_i \\
&\leq \sum_{0 \leq i \leq k} \text{depth}_{t'}(\mathrm{A}_i) \cdot n^{2/3} + n^{-1/4} \cdot \text{cost}^{t'}(a_0, \ldots, a_k) \\
&\leq k^2 n^{2/3} + n^{-1/4} \cdot \text{cost}^{t'}(a_0, \ldots, a_k).
\end{aligned}$$

Similarly, we get a corresponding lower bound. Thus, for each comparison tree $t' \in \mathcal{T}_k$ it holds that

$$\frac{\operatorname{cost}^{t'}(a_0,\dots,a_k)}{n^{1/4}} - k^2 n^{2/3} \le \operatorname{cost}^{t'}(a'_0,\dots,a'_k) \le \frac{\operatorname{cost}^{t'}(a_0,\dots,a_k)}{n^{1/4}} + k^2 n^{2/3},$$

and we get the following bound:

$$\begin{aligned}\operatorname{cost}^{t}(a_0,\dots,a_k) &- \operatorname{cost}^{t^*}(a_0,\dots,a_k)\\ &\le n^{1/4}\big(\operatorname{cost}^{t}(a'_0,\dots,a'_k) - \operatorname{cost}^{t^*}(a'_0,\dots,a'_k)\big) + 2n^{1/4}\cdot k^2 \cdot n^{2/3}\\ &\le 2k^2 \cdot n^{11/12}.\end{aligned}$$

(The last inequality follows because t has minimal cost w.r.t. $a'_0,\dots,a'_k$.) Thus, t is good. □

Using this lemma, the average comparison count of $\mathcal{SP}_k$ is at most a summand of $o(n)$ larger than the average comparison count of $\mathcal{N}_k$. Hence, $\mathcal{SP}_k$ is asymptotically optimal as well. □

Remark 6.4.6. Since the number of comparison trees in $\mathcal{T}_k$ is exponentially large in k, one might want to restrict the set of used comparison trees to some subset $\mathcal{T}'_k \subseteq \mathcal{T}_k$. The strategies presented here are optimal w.r.t. this subset of possible comparison trees as well.

6.5. Guesses About the Optimal Average Comparison Count of k-Pivot Quicksort

In this section we use the theory developed so far to consider the optimal average comparison count of k-pivot quicksort. We compare the result to the well known median-of-k strategy of classical quicksort.

By Lemma 6.2.2 and Theorem 6.4.3, the minimal average comparison cost for k-pivot quicksort, up to lower order terms, is

$$\frac{1}{\binom{n}{k}} \sum_{a_0+\dots+a_k=n-k} \min\left\{\operatorname{cost}^{t}(a_0,\dots,a_k) \mid t \in \mathcal{T}_k\right\} + o(n). \tag{6.12}$$

Then applying Theorem 6.1.1 gives the minimal average comparison count for k-pivot quicksort.

Unfortunately, we were not able to solve (6.12) for $k \ge 4$. (Already the solution for $k = 3$ as stated in Section 6.3 required a lot of manual tweaking before using Maple®.) This remains an open question. We resorted to experiments. As we have seen at the end of Section 4, estimating the total average comparison count by sorting inputs does not allow to estimate the leading term of the average comparison count correctly, because lower order terms have a big influence on the average comparison count for real-life input lengths. We used the following approach instead: For each $n \in \{5\cdot 10^4, 10^5, 5\cdot 10^5, 10^6, \dots, 5\cdot 10^7\}$, we generated 10 000 random permutations of

k	opt. k-pivot	median-of-k
2	$1.8n \ln n$	–
3	$1.705n \ln n$	$1.714n \ln n$
4	$1.65n \ln n$	–
5	$1.61n \ln n$	$1.622n \ln n$
6	$1.59n \ln n$	–
7	$1.577n \ln n$	$1.576n \ln n$
8	$1.564n \ln n$	–
9	$1.555n \ln n$	$1.549n \ln n$

Table 6.1.: Optimal average comparison count for k-pivot quicksort for $k \in \{2, \dots, 9\}$. For $k \geq 4$ these numbers are based on experiments). For odd k, we also include the average comparison count of quicksort with the median-of-k strategy. (The numbers for the median-of-k variant can be found in [Emd70] or [Hen91].)

$\{1, \dots, n\}$ and ran strategy $\mathcal{O}_k$ for each input, i. e., we only classified the input with the optimal strategy. The figures were constant beyond $n = 5{\cdot}10^5$, so we restrict our evaluation to $n = 5{\cdot}10^7$. For the average partitioning cost measured in these experiments, we then applied (6.2) to derive the leading factor of the total average comparison count. Table 6.1 shows the measurements we obtained for $k \in \{2, \dots, 9\}$ and $n = 5 \cdot 10^7$. Note that the results for $k \in \{2, 3\}$ are almost identical to the exact theoretical bounds. Additionally, this table shows the theoretical results known for classical quicksort using the median-of-k strategy, see Section 5. Interestingly, from Table 6.1 we see that based on our experimental data for k-pivot quicksort the median-of-k strategy has a slightly lower average comparison count than the (rather complicated) optimal partitioning methods for k-pivot quicksort for $k \geq 7$.

We have already seen that for classical quicksort using the median-of-k strategy, the leading term of the average sorting cost matches the lower bound of $\approx 1.4426..n \ln n + O(n)$ for k large enough [MR01]. This is also true for optimal k-pivot quicksort. For a proof, observe that optimal k-pivot quicksort is not worse than k-pivot quicksort that uses a fixed tree from $\mathcal{T}_k$ in each classification. Now, suppose $k = 2^\kappa - 1$, for an integer $\kappa \geq 1$. Then there exists exactly one tree in $\mathcal{T}_k$ in which all leaves are on level κ ("the symmetric tree"). Classifying the input with this tree makes exactly $\kappa \cdot (n - k)$ comparisons. According to Theorem 6.1.1, this strategy turned into a k-pivot quicksort algorithm has average sorting cost

$$\frac{\kappa n \ln n}{\mathcal{H}_{2^\kappa} - 1} + o(n \ln n). \tag{6.13}$$

When k goes to infinity, the average sorting cost is hence $1/(\ln 2) n \ln n + o(n \ln n)$. So, the average comparison count of optimal k-quicksort also converges to the lower bound for comparison-based sorting algorithms.

6.6. Discussion

In this section we considered the generalization of our theory for dual-pivot quicksort to the case that we use more than two pivots. We showed how to calculate the average comparison count for an arbitrary k-pivot quicksort algorithm. We generalized the natural optimal dual-pivot quicksort algorithms to k-pivot quicksort algorithms and proved their optimality with respect to minimizing the average comparison count. While we exemplified our theory at the case of three pivots, the formulas are so complicated that for $k \geq 4$ we had to resort to experiments. The results of these experiments suggested that comparison-optimal k-pivot quicksort is not better than classical quicksort using the median-of-k strategy, even for small k. Since this section ends our study on the average comparison count of k-pivot quicksort algorithms, we reflect on some open questions:

1. It would be interesting to see how one calculates the solution for (6.12) to obtain the optimal average comparison count of k-pivot quicksort for $k \geq 4$.

2. One should also study the average comparison count of strategy $\mathcal{O}_k$, for $k \geq 3$, in terms of a direct argument comparable to the proof of Theorem 4.2.1 for $\mathcal{O}_2$. Although for $k = 3$ we know from the optimality of strategy $\mathcal{N}_3$ in connection with (6.9) that strategy $\mathcal{O}_3$ makes $133/72n + o(n)$ comparisons on average, we do not know how to obtain this bound directly.

3. We conjecture that the difference in the average comparison count of $\mathcal{O}_k$ and $\mathcal{C}_k$ is significantly smaller than $o(n)$. Experiments suggest that the difference is $O\left(k^2 \log n\right)$.

4. The comparison of k-pivot quicksort and median-of-k quicksort from Table 6.1 might be unfair. We compared these variants because each of these algorithms looks at k elements of the input. Actually, the cost for sorting the pivots influences the linear term of the average comparison count, so one should rather look for k and k' such that optimal k-pivot quicksort makes the same effort as classical quicksort using the median-of-k' strategy with respect to lower order terms. To prove such results, the solution of the recurrence for generalized quicksort (see Theorem 6.1.1) must involve exact lower order terms. Currently, such solutions are only known for partitioning cost $a \cdot n + O(1)$ for constant a from [Hen91; Ili14].

We close our theoretical study of optimal k-pivot quicksort w.r.t. the average comparison count with one remark about the practical impact of optimal k-pivot quicksort algorithms. When the optimal comparison tree is computed by the dynamic programming algorithm mentioned in [Knu73] for optimal binary search trees, neither strategy $\mathcal{C}_k$ nor $\mathcal{SP}_k$ can compete in empirical running times with classical quicksort. For k small enough, we can find out what comparisons the dynamic programming approach makes to compute the optimal comparison tree. These decisions can be "hard-coded" into the k-pivot quicksort algorithm. For $k = 3$, the results of the experiments from [Kla14] clearly show that optimal strategies cannot compete with classical

quicksort, even when bypassing the computation of a cost-minimal comparison tree. We will consider the case $k = 2$ in detail in Section 8. The experiments suggest that strategy $\mathcal{SP}_2$ is faster than classical quicksort.

This immediately raises the question if there are other theoretical cost measures more suited to explain running time behavior. This will be the topic of the next section.

7. The Cost of Rearranging Elements

In the previous sections we focused on the average number of comparisons needed to sort a given input in terms of solving the classification problem. We have described very natural comparison-optimal k-pivot quicksort algorithms. However, experiments suggested that the improvements in the average comparison count can be achieved in much simpler ways, e. g., by using the median from a small sample as pivot in classical quicksort.

Kushagra *et al.* described in [Kus+14] a fast three-pivot algorithm and gave reason to believe that the improvements of multi-pivot quicksort algorithms with respect to running times are due to their better cache behavior. They also reported from experiments with a seven-pivot algorithm, which ran more slowly than their three-pivot algorithm. The goal of this section is to find out how their arguments generalize to quicksort algorithms that use more than three pivots. In connection with the running time experiments from Section 8, this allows us to make more accurate predictions than [Kus+14] about the influence of cache behavior to running time. One result of this study will be that it is not surprising that their seven-pivot approach is slower, because it has worse cache behavior than three- or five-pivot quicksort algorithms using a specific partitioning strategy.

At the beginning of this section we will reflect upon the importance of different cost measures with respect to the running time of a sorting algorithm. Then we will specify the problem setting, introduce the basic primitive which allows "moving elements around" and discuss related work. Next, we will describe three different algorithms. These algorithms will be evaluated with respect to the number of assignments they make and the number of memory accesses they incur. The latter cost measure will allow us to speculate about the cache behavior of these algorithms.

7.1. Why Look at Other Cost Measures Than Comparisons

Counting the average number of assignments a sorting algorithm makes on a given input is the another classical cost measure for sorting algorithms. From a running time perspective, it seems unintuitive that comparisons are the crucial factor, especially when key comparisons are cheap, e. g., when comparing 32-bit integers. Counting assignments might be more important, because an assignment usually involves access to a memory cell, which can be very slow due to significant speed differences between the CPU and the main memory. The situation is of course not that simple. From an empirical point of view, a mixture of many different components makes an algorithm fast (or unavoidably slow). For example, while the comparison of two elements is usually cheap, mispredicting the destination that a branch takes, i. e., the outcome of the comparison, may incur a significant penalty to running time, because the CPU wasted work on

executing instructions on the wrongly predicted branch. These so-called *branch mispredictions* are an important bottleneck to the performance of modern CPU's [HP12]. On the other hand, the *cache behavior* of an algorithm is also very important to its running time, because an access to main memory in modern computers can be slower than executing a few hundred simple CPU instructions. A cache tries to speed up access to main memory.

We first give a short theoretical introduction to caches. Here we adopt the cache-related notation of Mehlhorn and Sanders [MS03]. A *cache* consists of m *cache blocks* or *cache lines*. Each cache block has a certain size B. The *cache size* is $M = m \cdot B$. (This can be measured in bytes or the number of items of a certain data type.) Data transport from higher levels, e. g., other caches or main memory, is done in *memory blocks*. Typically, the cache is divided into $s = m/a$ *cache sets*, where a is called the *associativity* of the cache. A memory block with address $x \cdot B$ can only be stored in the cache set x mod s. For reasons of speed, price, and energy consumption, actual caches usually have an associativity of at most 16.

Every memory access is first looked up in the cache. If the cache contains the content of the memory cell then a *cache hit* occurs and the data can be used. Otherwise a *cache miss* is incurred and the data has to be retrieved from memory (or a higher-level cache). Then it will be stored in the cache. Storing a memory segment usually means that another segment must be evicted, and there exist different strategies to handle this situation. Nowadays, many CPU's use a variant of the "least recently used" (LRU) strategy, which evicts the cache block in the cache set whose last access lies farthest away in the past.

The cache structure of modern CPU's is hierarchical. For example, the Intel i7 that we used in our experiments has three data caches: There is a very small L1 cache (32KB of data) and a slightly larger L2 cache (256KB of data) very close to the processor. Each CPU core has its own L1 and L2 cache. They are both 8-way associative. Shared among cores is a rather big L3 cache that can hold 8MB of data and is 16-way associative. Caches greatly influence running time. While a lookup in main memory costs many CPU cycles ($\approx$ 140 cycles on the Intel i7 used in our experiments), a cache access is very cheap and costs about 4, 11, and 25 cycles for a hit in L1, L2, and L3 cache, respectively [Lev09]. Also, modern CPU's use *prefetching* to load memory segments into cache before they are accessed. Usually, there exist different prefetchers for different caches, and there exist different strategies to prefetch data, e. g., "load two adjacent cache lines", or "load memory segments based on predictions by monitoring data flow".

From a theoretical point, much research has been conducted to study algorithms with respect to their cache behavior, see, e. g., the survey paper of Rahman [Rah02]. (We recommend this paper as an excellent introduction to the topic of caches.) For such a study one first had to refine the standard model of computation, because in the classical RAM model [SS63] the machine operates on machine words with random access costing unit time. This model cannot be used to study cache effects. Consequently, Aggarwal and Vitter proposed the external memory model (EM-model) [AV88], in which the machine consists of a fast memory ("cache") of size M and an infinitely large, slow memory ("disk"). Data can only be accessed from fast cache and is exchanged between cache and disk in blocks of size B. The complexity of an algorithm in this model is usually measured by the number of cache faults it incurs. An algorithm in this model can

use M and B and must work for all (suitable) values of M and B. A *cache-oblivious* algorithm in the external memory model does not use M and B in its program code [Fri+12]. Hence, these algorithms show good cache behavior for arbitrary memory sizes.

In [LL99], LaMarca and Ladner gave a theoretical analysis of the cache behavior of sorting algorithms. They compared quicksort, mergesort, heapsort and radix sort and showed that cache misses can be analyzed rigorously. In the style of their paper, we will study three natural partitioning strategies for k-pivot quicksort.

The first strategy extends the "crossing-pointer technique" of Hoare [Hoa62] for classical quicksort, which was also the basis of Yaroslavskiy's algorithm [WN12] and the 3-pivot algorithm of Kushagra *et al.* [Kus+14]. The basic idea of this strategy is that one pointer scans the array from left to right; another pointer scans the array from right to left. Misplaced elements are exchanged "on the way" with the help of pointers that point to the starting cells of group segments. Our results, with regard to the cache behavior of this partitioning strategy, show that variants using 3 or 5 pivots have the best cache behavior. No benefit with respect to cache behavior can be achieved by using more than 5 pivots. This allows us to study the influence of cache behavior to running time in much more detail than previous work did. For example, Kushagra *et al.* [Kus+14] analyzed the cache behavior of classical quicksort, Yaroslavskiy's dual-pivot quicksort, and their own three-pivot algorithm. They drew the conclusion that cache behavior is the important factor to running time. Our results indicate that this hypothesis should be used with caution, since the 5-pivot quicksort algorithm will turn out to have even better cache behavior than the algorithm of Kushagra *et al.*, but it will be even slower than classical quicksort.

The second and third strategy work in a two-pass fashion and are inspired by the radix sort implementation of McIlroy *et al.* [MBM93] and the sample sort implementation of Sanders and Winkel [SW04]. Both strategies classify the input in the first pass to obtain the segment sizes of the input elements. One strategy uses these group sizes to copy elements to a correct position by allocating a second array. (Thus, it is not *in-place*.) The other strategy uses the segment sizes and permutes the input to obtain the actual partition. We will show that both of these strategies have good cache behavior when many pivots (e. g., 127 pivots) are used. However, we shall see that it is necessary to store the element classification in the first pass to make this algorithm competitive with respect to running time. So, we get an interesting space-time tradeoff.

While the importance of hardware data caches is folklore today, there are other cache like structures which are not that well known. According to the survey paper of Rahman [Rah02], minimizing misses in the so-called *translation-lookaside buffer* (TLB) can be as important (or more important) to the performance of programs. This buffer accelerates the translation of virtual addresses (used by every process in modern operation systems) to physical addresses in memory. Whenever the physical address corresponding to a virtual address in a process cannot be obtained from this buffer, the *hardware page walker* works with the translation table of the operation system to obtain the mapping. The crucial limitation is the number of entries it can hold. On our Intel i7 the TLB has two levels, consisting of 64 and 512 entries, respectively, for each core [Lev09]. In addition, there is a TLB consisting of 32 entries for large pages. Note that TLB misses and data cache misses can occur independently of each other, since entries in the data cache

are tagged by their physical address in main memory. The importance of the TLB in sorting algorithms has been noted in other papers, e. g., by Agarwal in [Aga96] and Jiménez-González *et al.* in [JNL02]. However, a theoretical model to address the cost of virtual address translation has only been introduced recently by Jurkiewicz and Mehlhorn [JM13]. Their paper is motivated by some very surprising experimental findings. For example, they showed that a random scan of an array with n elements has running time behavior like $O(n \log n)$, where standard measures would predict time $O(n)$. As it will turn out, TLB misses will also play a crucial role to the performance of multi-pivot quicksort algorithms.

Besides cache behavior, *branch mispredictions* can be another crucial factor for running time. A study of the empirical behavior of sorting algorithms with regard to branch mispredictions is due to Biggar *et al.* [Big+08]. Modern CPU's are pipelined and use a dynamic branch predictor to predict the outcome of a conditional branch. A mispredicted branch always means wasted work because the wrong instructions have been executed in the pipeline. On some hardware architectures, the pipeline must be flushed after a branch misprediction, which usually involves a penalty proportional to the depth of the pipeline. (This is not the case on the Intel i7 we used in our experiments.) For general purpose programs, these predictors work rather well, see, e. g., the paper of Biggar *et al.* [Big+08] and the references therein. However, branch prediction is hard for comparison-based sorting algorithms: Slightly extending the standard lower bound argument based on decision trees for comparison-based sorting algorithms, Brodal and Moruz showed in [BM05] that a comparison-based sorting algorithm which makes $O(dn \log n)$ comparisons, for a constant $d > 1$, makes $\Omega(n \log n / \log d)$ branch mispredictions. (For $d = 1$ it must make $\Omega(n \log n)$ branch mispredictions.) Making these calculations precise, in algorithms which make close to $n \log n$ comparisons, each comparison has a close to 50% chance of being true, see, e. g., [SW04].[1] When the number of comparisons is further away from $n \log n$, comparisons are biased. For example, Biggar *et al.* [Big+08] proved that in classical quicksort with random pivot choice, key comparisons can be successfully predicted in about 71% of the cases. A very recent, detailed analysis of classical quicksort and Yaroslavskiy's algorithm is due to Martínez *et al.* [MNW15]. They showed that branch mispredictions can be analyzed accurately. However, they conclude that this cost measure cannot explain the actual difference in running time between Yaroslavskiy's algorithm and classical quicksort observed in practice.

Branch mispredictions can yield odd effects for specific hardware choices: Kaligosi and Sanders [KS06] report from experiments on a Pentium 4 Prescott generation which has such a long pipeline (that has to be flushed upon a misprediction) that branch mispredictions become a dominating factor in the running time. In that case, choosing a skewed pivot may actually improve running time! For example, in [KS06] the fastest quicksort variant uses the element of rank $n/10$ as pivot. We will disregard branch mispredictions in this section. In the experimental evalua-

[1] This can also be used to explain many insights we found in Section 5. A dual pivot approach that uses the tertiles of a sample yields "skewed" pivots, i. e., biased comparisons. Hence it cannot make close to $n \log n$ comparisons on average. Furthermore, it explains why strategy $\mathcal{L}$ using the elements of rank $n/4$ and $n/2$ as pivots makes close to $n \log n$ comparisons. Also, Yaroslavskiy's algorithm cannot achieve this lower bound because it uses both pivots for comparisons and hence must have biased comparisons.

tion in Section 8 we will report on the behavior of our algorithms with respect to branch mispredictions and mention some known programming techniques to lower the number of branch mispredictions.

7.2. Problem Setting, Basic Algorithms and Related Work

For the analysis, we again assume that the input is a random permutation of the set $\{1, \ldots, n\}$ which resides in an array $A[1..n]$. Fix an integer $k \geq 1$. The first k elements are chosen as the pivots. Our goal is to obtain a partition of the input, as depicted in Figure 6.1 on Page 46. Here, determining whether the element $A[i]$ belongs to group $\mathrm{A}_0, \mathrm{A}_1, \ldots,$ or A_k is for free, and we are interested in the average number of element movements and the average number of memory accesses needed to obtain the partition. (The latter cost measure will be defined precisely in Section 7.5.)

In terms of moving elements around, one traditionally uses the "swap"-operation which exchanges two elements. In the case that one uses two or more pivots, we we will see that it is beneficial to generalize this operation. We define the operation `rotate`$(\mathtt{i}_1, \ldots, \mathtt{i}_\ell)$ as follows:

$$\text{tmp} \leftarrow A[i_1]; A[i_1] \leftarrow A[i_2]; A[i_2] \leftarrow A[i_3]; \ldots; A[i_{\ell-1}] \leftarrow A[i_\ell]; A[i_\ell] \leftarrow \text{tmp}.$$

Intuitively, `rotate` performs a cyclic shift of the elements by one position. (A `swap`$(A[i_1], A[i_2])$ is a `rotate`(i_1, i_2).) A `rotate`$(i_1, \ldots, i_\ell)$ operation makes exactly $\ell + 1$ assignments.

Assuming the groups of all elements are known, we can think of the problem in the following way: Given n elements from the set $\{0, \ldots, k\}$, for k being a constant, rearrange the elements into ascending order. For $k = 2$, this problem is known under the name "Dutch national flag problem", proposed by Dijkstra [Dij76]. (Given n elements where each element is either "red", "white", or "blue", rearrange the elements such that they resemble the national flag of the Netherlands.) The algorithm proposed by Dijkstra has been used to deal with the problem of equal elements in standard quicksort and is known as "3-way partitioning" [SB]. As noticed by Wild *et al.* in [WNN13], an improved algorithm for the *dutch national flag* problem due to Meyer [Mey78] is the partitioning strategy in Yaroslavskiy's algorithm. For the analysis of algorithms solving this problem see [McM78]. For $k > 2$, we only know the work of McIlroy *et al.* [MBM93], who devised an algorithm they named "American flag sort", which will be the topic of Section 7.3.1.

7.3. Algorithms

We will discuss three different algorithms. We shall disregard the pivots in the description of the algorithms. We assume that they reside in the first k cells of the array. In a final step the k pivots have to be moved into the correct positions between group segments. This is possible with at most k rotate operations.

7.3.1. Partitioning After Classification

Here we assume that the elements of the input have been classified in a first pass. Partitioning the input with respect to the $k+1$ different groups is then solved via the following approach that is an adaption of Algorithm 4.1 from [MBM93].

Each $k \geq 1$ gives rise to an algorithm *Permute*$_k$. It works in the following way. Suppose the group sizes are $a_0, \ldots, a_k$. For each $h \in \{0, \ldots, k\}$ let $o_h = k + 1 + \sum_{0 \leq i \leq h-1} a_i$. Let $o_{k+1} = n + 1$. Then the segment in the array which contains the elements of group A_h in the partition is $A[o_h..o_{h+1}-1]$. For each group A_h, $h \in \{0, \ldots, k\}$, the algorithm uses two variables. The variable c_h ("count") contains the number of elements in group A_h that have not been seen so far. (Of course, initially $\mathsf{c}_h = a_h$.) The variable o_h ("offset") contains the largest index where the algorithm has made sure that $A[o_h..\mathsf{o}_h - 1]$ only contains A_h-elements. Initially, $\mathsf{o}_h = o_h$. Basically, the algorithm scans the array from left to right until it finds a misplaced element at $A[j]$ with $o_h \leq j \leq o_{h+1} - 1$. Let this element be x and suppose x belongs to group $h' \in \{0, \ldots, k\}$. Now repeat the following until an element is written into $A[j]$: The algorithm scans the array from $A[\mathsf{o}_{h'}]$ to the right until it finds a misplaced element y at $A[j']$. (Note that $j' \leq o_{h'+1} - 1$.) Assume y belongs to group h''. Write x into $A[j']$. If $h'' = h$, then write y into $A[j]$. Otherwise, set $h' = h''$ and $x = y$ and continue the loop. This is iterated until the input is partitioned. The pseudocode of the algorithm is shown in Algorithm 1. An example of this algorithm is given in Figure 7.1.

We also consider a variant of Algorithm 1 we call "Copy$_k$". This algorithm was the basic partitioning algorithm in the "super scalar sample sort algorithm" of Sanders and Winkel [SW04]. It uses the same offset values as Algorithm 1. Instead of an in-place permutation it allocates a new array and produces the partition by sweeping over the input array, copying elements to a final position in the new array using these offsets. So, this algorithm does not work in-place. The pseudocode of this algorithm is given as Algorithm 2.

7.3.2. Partitioning During Classification

We now describe a family of algorithms that produce a partition in a single pass. Each $k \geq 1$ gives rise to an algorithm *Exchange*$_k$. The basic idea is similar to classical quicksort: One pointer scans the array from left to right; another pointer scans the array from right to left, exchanging misplaced elements on the way. Algorithm 3 uses $k - 1$ additional pointers to store the start of groups $\mathrm{A}_1, \ldots, \mathrm{A}_{k-1}$. Figure 7.3 shows the idea of the algorithm for $k = 6$; Figures 7.4–7.6 show the different rotations being made by Algorithm 3 in lines 8, 13, and 17. This algorithm is used for $k = 3$ in the implementation of the 3-pivot algorithm of Kushagra *et al.* [Kus+14]. For $k = 2$ it can be used to improve the basic implementation of Yaroslavskiy's algorithm from [WN12], see [WNN13].

We now study these algorithms with respect to the average number of assignments and the average number of memory accesses they make.

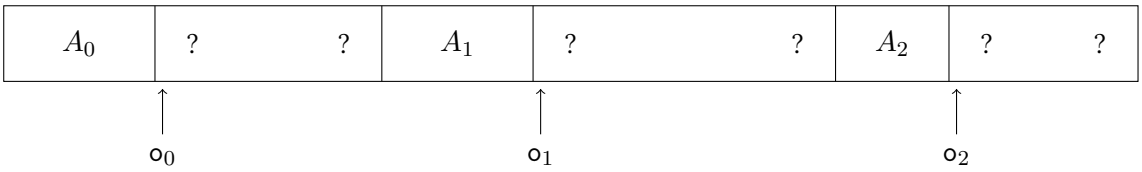

Figure 7.1.: General memory layout of Algorithm 1 for $k = 2$.

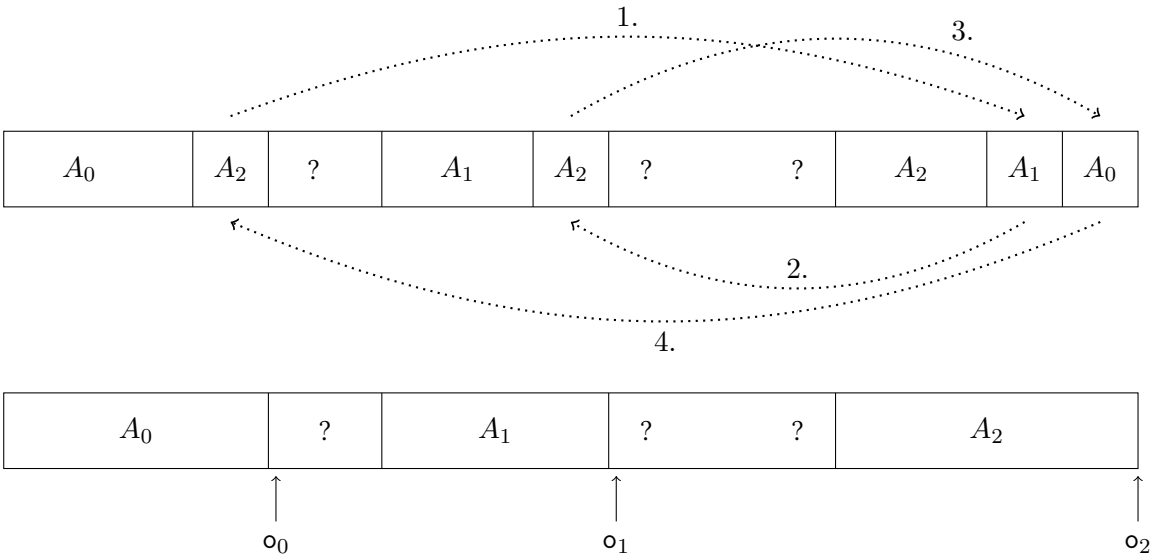

Figure 7.2.: Top: Example for the cyclic rotations occurring in one round of Algorithm 1 starting from the example given in Figure 7.1. First, the algorithm finds an A_2-element, which is then moved into the A_2-segment (1.), replacing an A_1-element which is moved into the A_1-segment (2.). It replaces an A_2-element that is moved to replace the next misplaced element in the A_2-segment, an A_0 element (3.). This element is then moved to the A_0-segment (4.), overwriting the misplaced A_2-element, which ends the round. Bottom: Memory layout and offset indices after moving the elements from the example.

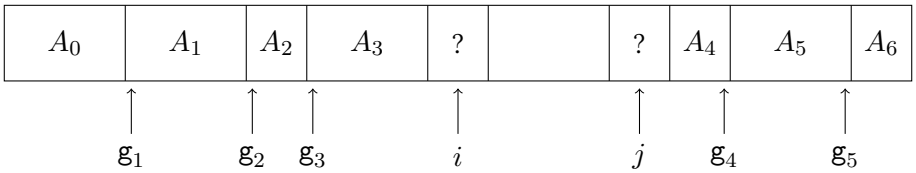

Figure 7.3.: General memory layout of Algorithm 3 for $k = 6$. Two pointers i and j are used to scan the array from left-to-right and right-to-left, respectively. Pointers $g_1, \ldots, g_{k-1}$ are used to point to the start of segments.

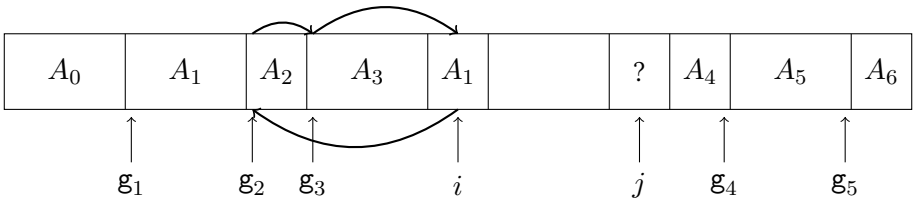

Figure 7.4.: The `rotate` operation in Line 8 of Algorithm 3. An element that belongs to group A_1 is moved into its respective segment. Pointers i, g_2, g_3 are increased by 1 afterwards.

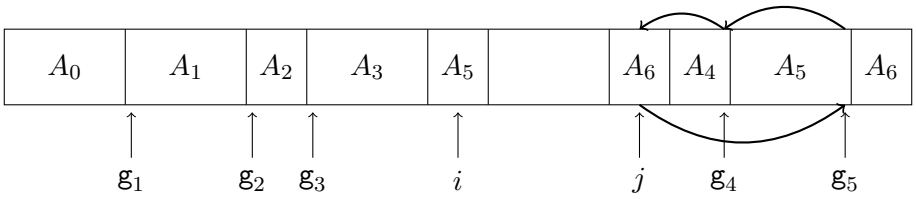

Figure 7.5.: The `rotate` operation in Line 13 of Algorithm 3. An element that belongs to group A_6 is moved into its respective segment. Pointers j, g_4, g_5 are decreased by 1 afterwards.

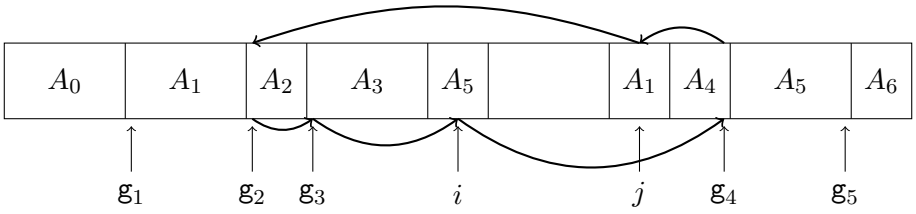

Figure 7.6.: Example for the rotate operation in Line 17 of Algorithm 3. The element found at position i is moved into its specific segment. Subsequently, the element found at position j is moved into its specific segment.

Algorithm 1 Permute elements to produce a partition

procedure *Permute*$_k(A[1..n])$
Requires: Segment sizes are $a_0, \ldots, a_k$.

```
1:  ∀h ∈ {0,...,k} : c_h ← a_h; o_h = k + 1 + Σ_{i=0}^{h-1} a_i;
2:  for h from 0 to k − 1 do
3:      while c_h > 0 do
4:          while A[o_h] belongs to group A_h do                ▷ Find misplaced element
5:              o_h ← o_h + 1; c_h ← c_h − 1;
6:          if c_h = 0 then
7:              break;
8:          home ← o_h;
9:          from ← home;
10:         x ← A[from];
11:         while true do                                       ▷ Move elements cyclicly
12:             A_g ← Group of x;
13:             while A[o_g] belongs to group A_g do            ▷ Skip non-misplaced elements
14:                 o_g ← o_g + 1; c_g ← c_g − 1;
15:             to ← o_g; o_g ← o_g + 1; c_g ← c_g − 1;
16:             from ← to;
17:             if home ≠ from then
18:                 r ← A[to]; A[to] ← x; x ← r;
19:             else
20:                 A[from] ← x;
21:                 break;
```

7.4. Assignments

Since we are interested in cost measures for predicting empirical running time, we only count assignments *involving an array access*. For example, in Line 18 of Algorithm 1 we count two assignments. Here we assume that variables which are needed frequently are in registers of the CPU. An assignment between two registers is much cheaper.

We start by fixing notation. Let AS_n and PAS_n be the total number of assignments needed for sorting and partitioning, resp., a given input of length n. For the average number of assignments, we get the recurrence

$$\mathrm{E}(\mathrm{AS}_n) = \mathrm{E}(\mathrm{PAS}_n) + \frac{1}{\binom{n}{k}} \sum_{a_0+\ldots+a_k=n-k} (\mathrm{E}(\mathrm{AS}_{a_0}) + \ldots + \mathrm{E}(\mathrm{AS}_{a_k})).$$

This recurrence has the same form as (6.1), so we may apply (6.2) and focus on a single partition

Algorithm 2 Copy elements to produce a partition

procedure *Copy*$_k(A[1..n])$
Requires: Segment sizes are $a_0, \ldots, a_k$.
1: $\forall h \in \{0, \ldots, k\} : \mathtt{o}_h = k + 1 + \sum_{i=0}^{h-1} a_i$;
2: allocate a new array $B[1..n]$;
3: **for** $\mathtt{i}$ **from** $k + 1$ **to** n **do**
4: $\quad \mathrm{A}_p \leftarrow$ group of $A[\mathtt{i}]$;
5: $\quad B[\mathtt{o}[p]] \leftarrow A[\mathtt{i}]$;
6: $\quad \mathtt{o}[p] \leftarrow \mathtt{o}[p] + 1$;
7: Copy the content of B to A;

step. We consider Algorithm 1. In each round, Algorithm 1 makes one assignment in Line 10 and two assignments involving array access for each iteration that reaches Line 18. At the end of the loop it makes one assignment in Line 20. We charge two assignments to each element that is moved to its final location in Line 18. (Recall that we only account for assignments involving array cells.) Furthermore, the assignment in Line 10 is charged extra to the assignment in Line 20. Thus, each misplaced element incurs exactly two assignments. Furthermore, we charge for an input of n elements cost $n - k$ for the classification step that precedes partitioning.

By a simple calculation it follows that on average there are $k(n - k)/(k + 2)$ misplaced elements. Hence,

$$\mathrm{E}\left(\mathrm{PAS}_n\right) = \frac{3k + 4}{k + 2} \cdot (n - k), \tag{7.1}$$

and by applying (6.2) we conclude that

$$\mathrm{E}\left(\mathrm{AS}_n\right) = \frac{3k + 4}{(k + 2) \cdot (\mathcal{H}_{k+1} - 1)} \cdot n \ln n + o(n \ln n). \tag{7.2}$$

In particular, the average partitioning cost will converge to $3(n - k)$ for large k.

The analysis of Algorithm 2 is even simpler. It makes exactly $n - k$ assignments to produce a partition (Line 5 in Algorithm 2). In addition, we charge $n - k$ assignments for copying the input back (Line 7 in Algorithm 2), and $n - k$ assignments for the classification step, which we charge to Line 4 in Algorithm 2. So, it makes $3(n - k)$ assignments in one partitioning step, and the average number of assignments for sorting is

$$3n \ln n/(\mathcal{H}_{k+1} - 1) + o(n \ln n). \tag{7.3}$$

Counting assignments in Algorithm 3 is a little bit harder. Fix the pivots and hence the group sizes $a_0, \ldots, a_k$. Let $k' = \lceil (k - 1)/2 \rceil$, and let $H := a_0 + \ldots + a_{k'}$. We charge the number of assignments in Algorithm 3 to the two pointers i and j separately in the following way: Let p

Algorithm 3 Move elements by rotations to produce a partition

procedure *Exchange*$_k(A[1..n])$

1: $\mathtt{i} \leftarrow k+1; \mathtt{j} \leftarrow n;$
2: $\mathtt{k}' \leftarrow \lceil \frac{k-1}{2} \rceil;$
3: $\mathtt{g}_1, \ldots, \mathtt{g}_{\mathtt{k}'} \leftarrow \mathtt{i};$
4: $\mathtt{g}_{\mathtt{k}'+1}, \ldots, \mathtt{g}_{k-1} \leftarrow \mathtt{j};$
5: $\mathtt{p}, \mathtt{q} \leftarrow -1;$ ▷ p and q hold the group indices of the elements indexed by i and j.
6: **while** $\mathtt{i} < \mathtt{j}$ **do**
7: **while** $A[\mathtt{i}]$ belongs to group $A_{\mathtt{p}}$ with $\mathtt{p} \leq \mathtt{k}'$ **do**
8: **if** $p < \mathtt{k}'$ **then**
9: rotate(i,$\mathtt{g}_{\mathtt{k}'}, \ldots, \mathtt{g}_{\mathtt{p}+1}$);
10: $\mathtt{g}_{\mathtt{p}+1}\mathtt{++}; \ldots; \mathtt{g}_{\mathtt{k}'}\mathtt{++};$
11: i++;
12: **while** $A[\mathtt{j}]$ belongs to group $A_{\mathtt{q}}$ with $\mathtt{q} > \mathtt{k}'$ **do**
13: **if** $\mathtt{q} > \mathtt{k}' + 1$ **then**
14: rotate(j,$\mathtt{g}_{\mathtt{k}'+1}, \ldots, \mathtt{g}_{\mathtt{q}-1}$);
15: $\mathtt{g}_{\mathtt{q}-1}\mathtt{--}; \ldots; \mathtt{g}_{\mathtt{k}'+1}\mathtt{--};$
16: j--;
17: **if** $\mathtt{i} < \mathtt{j}$ **then**
18: rotate(i, $\mathtt{g}_{\mathtt{k}'}, \ldots, \mathtt{g}_{\mathtt{q}+1}$, j, $\mathtt{g}_{\mathtt{k}'+1}, \ldots, \mathtt{g}_{\mathtt{p}-1}$);
19: i++; $\mathtt{g}_{\mathtt{q}+1}\mathtt{++}; \ldots; \mathtt{g}_{\mathtt{k}'}\mathtt{++};$
20: j--; $\mathtt{g}_{\mathtt{k}'+1}\mathtt{--}; \ldots; \mathtt{g}_{\mathtt{p}-1}\mathtt{--};$

and q hold the content of the variables p and q, respectively, when the pointer i is at position i, and the pointer j is at position j. The rotate operation in Line 8 is charged to the pointer i. One rotation of an element that belongs to group $A_p, p < k'$, makes $2 + k' - p$ assignments. Similarly, the rotate operation in Line 13 is charged to the pointer j and has cost $1 + q - k'$. The rotate operation in Line 17 makes exactly $2 + p - q$ assignments and is charged as follows: We charge $p - k' - 1$ assignments to pointer i, and $3 + k' - q$ assignments to pointer j. Consequently, we define

$$\text{cost}_i(p) = \begin{cases} 2 + k' - p, & \text{if } p < k', \\ p - k' - 1, & \text{if } k' + 1 < p, \\ 0, & \text{otherwise,} \end{cases}$$

and

$$\text{cost}_j(q) = \begin{cases} 3 + k' - q, & \text{if } q < k', \\ 1 + q - k', & \text{if } k' + 1 < q, \\ 0, & \text{otherwise.} \end{cases}$$

Let $\text{A}_{i,p}$ and $\text{A}_{j,q}$ be the number of elements inspected by i and j that belong to group A_p and A_q, respectively. The exact assignment count for partitioning an input of n elements is then

$$\text{PA}_n = \sum_{p=0}^{k} \text{A}_{i,p} \cdot \text{cost}_i(p) + \sum_{q=0}^{k} \text{A}_{j,q} \cdot \text{cost}_j(q). \tag{7.4}$$

For a random input (excepting the pivot choices) the average number of assignments can be calculated as follows: Pointer i is increased by one whenever an element is found that belongs to group $\text{A}_p, p \leq k'$. Thus, it inspects exactly H array cells. Analogously, pointer j inspects $n-k-H$ array cells.[2] So, the pointer i incurs assignments for the array segment $A[k+1..k+H+1]$, while the pointer j makes assignments for $A[k+H+2..n]$. Now, consider $\text{E}(\text{A}_{i,p} \mid a_0, \ldots, a_k)$ for a fixed integer p, with $0 \leq p \leq k$. There are exactly a_p elements of group A_p in the input. The positions of these elements are drawn from the $n-k$ possible positions by sampling without replacement. Hence, we know that

$$\text{E}(\text{A}_{i,p} \mid a_0, \ldots, a_k) = H \cdot \frac{a_p}{n-k}.$$

Analogously, for a fixed integer q, with $0 \leq q \leq k$, we have that

$$\text{E}(\text{A}_{j,q} \mid a_0, \ldots, a_k) = (n-k-H) \cdot \frac{a_q}{n-k}.$$

Plugging this into (7.4) and rearranging terms yields, for odd k, the formula

$$\begin{aligned} \text{E}(\text{PAS}_n \mid a_0, \ldots, a_k) = &\left(\frac{H}{n-k}\right) \cdot \left(\sum_{j=1}^{k'} \left((2+j) \cdot a_{k'-j} + j \cdot a_{k'+1+j}\right)\right) \\ &+ \left(\frac{n-k-H}{n-k}\right) \cdot \left(3a_{k'} + \sum_{j=1}^{k'} \left((3+j) \cdot a_{k'-j} + (2+j) \cdot a_{k'+1+j}\right)\right). \end{aligned} \tag{7.5}$$

[2] They may cross over by one element. We disregard this case; it costs no additional assignments.

For even k, we have that:

$$\mathrm{E}(\mathrm{PAS}_n \mid a_0, \ldots, a_k) = \left(\frac{H}{n-k}\right) \cdot \left(\sum_{j=1}^{k'} \left((2+j) \cdot a_{k'-j} + (j-1) \cdot a_{k'+j}\right)\right)$$
$$+ \left(\frac{n-k-H}{n-k}\right) \left(3a_{k'} + (3+k')a_0 + \sum_{j=1}^{k'-1} \left((3+j)a_{k'-j} + (1+j)a_{k'+1+j}\right)\right). \quad (7.6)$$

Using Maple® we calculated the average number of assignments of Algorithm 1–Algorithm 3 for $k \in \{1, \ldots, 9\}$. Since Algorithm 1 and Algorithm 2 benefit from larger values of k, we also calculated the average assignment count for using them with $15, 31, 63$, and 127 pivots. Table 7.1 shows the results of these calculations. The average assignment count of Algorithm 1 slowly increases for k getting larger. The average assignment count for sorting decreases, first rapidly, then more slowly. For $k \geq 31$, Algorithm 1 makes fewer assignments than classical quicksort, which makes $n \ln n$ assignments. Algorithm 2 makes exactly $3(n-k)$ assignments on each input. The average assignment count of this algorithm decreases for growing k, and is for $k \geq 31$ practically the same as the average assignment count of Algorithm 1. For Algorithm 3, the average number of assignments rapidly increases from classical quicksort to quicksort variants with at least two pivots. Afterwards, it slowly increases. Interestingly, Algorithm 3 is slightly better for three pivots than for two pivots.[3]

In summary, Algorithm 1 and Algorithm 2 make many assignments for small values of k. For $k \geq 31$, they achieve a lower average assignment count than classical quicksort. Algorithm 3 does not benefit from using more than one pivot. We will now compare these calculations with measurements we got from experiments.

Empirical Verification. Figure 7.7 shows the measurements we got with regard to the average number of assignments for implementations of the algorithms described before. In the experiment, we sorted 600 random permutations of $\{1, \ldots, n\}$ for each $n = 2^i$ with $9 \leq i \leq 27$.

For Algorithm 3, we see that the measurements agree with our theoretical study (*cf.* Table 7.1). There is a big gap between the average assignment count for one pivot ("Exchange$_1$") and the variants using more than one pivot. Also, the 3-pivot algorithm makes fewer assignments (on average) than the 2-pivot algorithm. In order, it follows the 5-pivot, 7-pivot (omitted in the plots), and 9-pivot algorithm. Their average assignment count is very close to our calculations. For Algorithm 1, we see that lower order terms have a big influence on the actual measurements. In all measurements, the average assignment count is slightly lower than what we expect from the leading term disregarding lower order terms. For large k the influence of lower order terms seems to decrease. We see that Algorithm 1 makes fewer assignments than classical quicksort for k large enough. Our experiments for Algorithm 2 showed that—as expected—there is almost

[3] This has been observed in the experiments in [Kus+14], too.

k	E(PAS$_n$) (Algorithm 1)	E(AS$_n$) (Algorithm 1)	E(PAS$_n$) (Algorithm 2)	E(AS$_n$) (Algorithm 2)	E(PAS$_n$) (Algorithm 3)	E(AS$_n$) (Algorithm 3)
1	$2.33n$	$4.66n \ln n$	$3n$	$6n \ln n$	$0.5n$	$n \ln n$
2	$2.5n$	$3n \ln n$	$3n$	$3.6n \ln n$	$1.33n$	$1.6n \ln n$
3	$2.6n$	$2.4n \ln n$	$3n$	$2.77n \ln n$	$1.70n$	$1.57n \ln n$
4	$2.66n$	$2.08n \ln n$	$3n$	$2.34n \ln n$	$2.13n$	$1.66n \ln n$
5	$2.71n$	$1.87n \ln n$	$3n$	$2.07n \ln n$	$2.40n$	$1.66n \ln n$
6	$2.75n$	$1.73n \ln n$	$3n$	$1.88n \ln n$	$2.75n$	$1.73n \ln n$
7	$2.77n$	$1.62n \ln n$	$3n$	$1.75n \ln n$	$3n$	$1.75n \ln n$
8	$2.8n$	$1.53n \ln n$	$3n$	$1.64n \ln n$	$3.31n$	$1.81n \ln n$
9	$2.82n$	$1.46n \ln n$	$3n$	$1.56n \ln n$	$3.55n$	$1.84n \ln n$
15	$2.88n$	$1.21n \ln n$	$3n$	$1.26n \ln n$	–	–
31	$2.94n$	$0.96n \ln n$	$3n$	$0.98n \ln n$	–	–
63	$2.97n$	$0.79n \ln n$	$3n$	$0.80n \ln n$	–	–
127	$2.98n$	$0.67n \ln n$	$3n$	$0.68n \ln n$	–	–

Table 7.1.: Average number of assignments for partitioning (E(PAS$_n$)) and average number of assignments for sorting (E(AS$_n$)) an array of length n disregarding lower order terms using Algorithm 1, Algorithm 2, and Algorithm 3. We did not calculate the average comparison count for Algorithm 3 for $k \in \{15, 31, 63, 127\}$. For comparison, note that classical quicksort ("Exchange$_1$") makes $n \ln n$ assignments involving array accesses on average.

no difference to Algorithm 1 for large values of k. Consequently, variants of Algorithm 2 are omitted from the plots.

Generalization of Algorithm 3. The figures from Table 7.1 show that the average assignment count of Algorithm 3 rapidly increases from one pivot to variants using more than one pivot. We make the following observations about a generalization of Algorithm 3. For this algorithm, the parameter k' (cf. Algorithm 3, Line 2) can be set to an arbitrary value from $\{0, \dots, k-1\}$ and the formulae (7.5) and (7.6) still hold. For $k = 2$, due to symmetry, setting $k' = 0$ or $k' = 1$ yields the same values. As in the sampling strategy $\mathcal{SP}$ for comparison-optimal multi-pivot quicksort algorithms, we can make use of unbalanced inputs, i. e., inputs where there are much more/less small elements than large elements, as follows. Suppose the group sizes are a_0, a_1, and a_2. If $a_0 > a_2$, the algorithm should use $k' = 0$, because it makes fewer assignments for elements that belong to group A$_0$. If $a_2 \geq a_0$, analogous arguments show that the variant with $k' = 1$ should be used. If the partitioning process would correctly choose the parameter k' depending on the relation of a_0 and a_2, the average assignment count for partitioning decreases from $1.333n + O(1)$ to $1.15n + O(1)$ in Table 7.1. For an actual implementation, one would look at the first $n^{3/4}$ elements and decide whether to choose $k' = 0$ or $k' = 1$ according to ratio of small/large element in the sample.

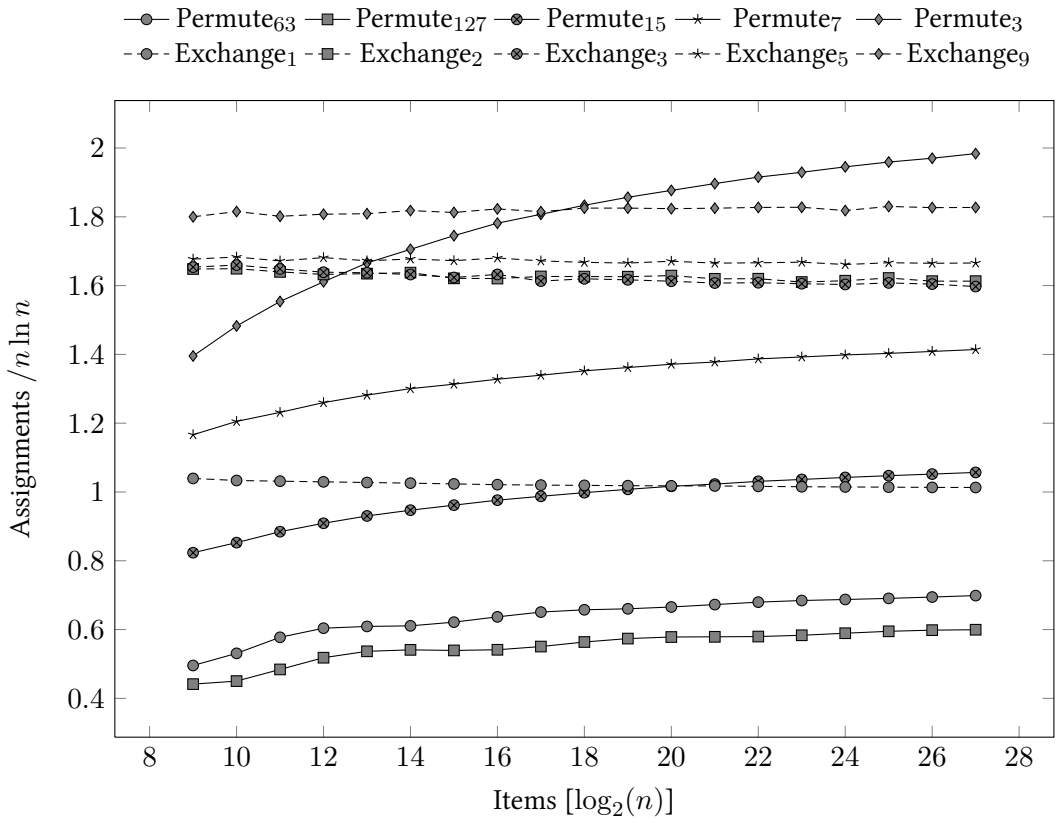

Figure 7.7.: The average number of assignments for sorting an input consisting of n elements using Algorithm 1 ("Permute$_k$") and Algorithm 3 ("Exchange$_k$") for certain values of k. Each data point is the average over 600 trials.

We did not look into generalizing this approach to $k \geq 3$. We remark that the analysis should be simpler than calculating the average comparison count for comparison-optimal k-pivot quicksort, since there are fewer cost terms. (For k pivots, $k' \in \{0, ..., k-1\}$. So for each group size we have to calculate the minimum over k cost terms, whereas for comparison-optimal k-pivot quicksort this minimum was taken over exponentially many cost terms.)

In summary, only Algorithm 1 and Algorithm 2 are better than classical quicksort with respect to counting assignments, but only for a large number of pivots. In particular, Algorithm 3 does not benefit from using more than one pivot, which was also observed for the case $k = 2$ by Wild and Nebel [WN12] and $k = 3$ by Kushagra *et al.* [Kus+14]. In the next section, we will consider a different cost measure that shows that partitioning using Algorithm 3 requires less effort when using more than one pivot.

7.5. Memory Accesses and Cache Misses

We will now study how many array accesses the pointers of the partitioning algorithms Algorithm 1–3 require to sort an input. As we will show this cost measure can be analyzed rigorously. From an empirical point of view the cost measure gives a lower bound on the number of clock cycles the CPU spends just waiting for memory. It can also be used to predict the cache behavior of Algorithm 1–3. We will see that it gives good estimates for the cache misses in L1 cache which we observed in our experiments.

A memory access occurs whenever the CPU reads the content of a memory address or writes to it. (We do not distinguish between read and write access, which is common when the architecture uses *write-back* caches [Rah02].) Let the random variable MA_n count the number of memory accesses that a k-pivot quicksort algorithm requires until an input of length n is sorted. Let PMA_n denote the number of memory accesses that the algorithm requires in the first partitioning step. In general, we get the recurrence:

$$\mathrm{E}(\mathrm{MA}_n) = \mathrm{E}(\mathrm{PMA}_n) + \frac{1}{\binom{n}{k}} \sum_{a_0+\ldots+a_k=n-k} \left(\mathrm{E}(\mathrm{MA}_{a_0}) + \ldots + \mathrm{E}(\mathrm{MA}_{a_k})\right).$$

Again, this recurrence has the form of (6.1), so we may apply (6.2). Thus, from now on we focus on a single partitioning step.

We charge memory accesses for Algorithm 1 as follows: During the classification phase each array cell is accessed exactly once. In the partitioning phase each element that is already at a final position is accessed exactly once. All other elements are accessed twice, once for reading the element, once for writing it into another table cell. Comparing with the analysis of the assignments that Algorithm 1 makes, there is exactly one array access per assignment, so we get (see Section 7.4)

$$\mathrm{E}(\mathrm{PMA}_n) = \frac{3k+4}{k+2}(n-k). \tag{7.7}$$

For Algorithm 2, we charge $n-k$ memory accesses for the first classification step. Then, we charge $2(n-k)$ array accesses for the partitioning step. Finally, we charge $2(n-k)$ array accesses for copying the input back. So we have

$$\mathrm{E}(\mathrm{PMA}_n) = 5(n-k). \tag{7.8}$$

Again, the analysis of Algorithm 3 is more difficult. For this algorithm, the number of memory accesses is the sum over all $i \in \{1,\ldots,k-1\}$ of the number of array cells visited by pointer $\mathtt{g}_i$ plus the number of array cells visited by pointers `i` and `j`. Let the pivots and thus $a_0,\ldots,a_k$ be fixed. The pointers $\mathtt{i},\mathtt{j}$ scan the whole array, and thus inspect $n-k$ array cells. When Algorithm 3 terminates, $\mathtt{g}_1$ points to $A[k+a_0+1]$, having visited exactly a_0 array cells. An

k	$E(PMA_n)$ (Algorithm 1)	$E(MA_n)$ (Algorithm 1)	$E(PMA_n)$ (Algorithm 2)	$E(MA_n)$ (Algorithm 2)	$E(PMA_n)$ (Algorithm 3)	$E(MA_n)$ (Algorithm 3)
1	$2.33n$	$4.66n \ln n$	$5n$	$10n \ln n$	$1n$	$2n \ln n$
2	$2.5n$	$3n \ln n$	$5n$	$6n \ln n$	$1.33n$	$1.6n \ln n$
3	$2.6n$	$2.4n \ln n$	$5n$	$4.62n \ln n$	$1.5n$	$1.385n \ln n$
4	$2.66n$	$2.08n \ln n$	$5n$	$3.89n \ln n$	$1.8n$	$1.402n \ln n$
5	$2.71n$	$1.87n \ln n$	$5n$	$3.45n \ln n$	$2n$	$1.379n \ln n$
6	$2.75n$	$1.73n \ln n$	$5n$	$3.14n \ln n$	$2.29n$	$1.435n \ln n$
7	$2.77n$	$1.62n \ln n$	$5n$	$2.91n \ln n$	$2.5n$	$1.455n \ln n$
8	$2.8n$	$1.53n \ln n$	$5n$	$2.73n \ln n$	$2.77n$	$1.519n \ln n$
9	$2.82n$	$1.46n \ln n$	$5n$	$2.59n \ln n$	$3n$	$1.555n \ln n$
15	$2.88n$	$1.21n \ln n$	$5n$	$2.1n \ln n$	$4.5n$	$1.89n \ln n$
31	$2.94n$	$0.96n \ln n$	$5n$	$1.63n \ln n$	$8.5n$	$2.78n \ln n$
63	$2.97n$	$0.79n \ln n$	$5n$	$1.34n \ln n$	$16.5n$	$4.41n \ln n$
127	$2.98n$	$0.67n \ln n$	$5n$	$1.18n \ln n$	$32.5n$	$7.33n \ln n$

Table 7.2.: Average number of memory accesses for partitioning ($E(PMA_n)$) and average number of memory accesses for sorting an array ($E(MA_n)$) of length n disregarding lower order terms. Note that classical quicksort makes $2n \ln n$ memory accesses on average.

analogous statement can be made for the pointers $g_2, \ldots, g_{k-1}$. On average, we have $(n-k)/(k+1)$ elements of each group $A_0, \ldots, A_k$, so g_1 and g_{k-1} each visit $(n-k)/(k+1)$ array cells on average, g_2 and g_{k-2} each visit $2(n-k)/(k+1)$ array cells, and so on.

For the average number of memory accesses during a partitioning step we consequently get

$$E(PMA_n) = \begin{cases} 2 \cdot \sum_{i=1}^{\lceil k/2 \rceil} \frac{i \cdot (n-k)}{k+1}, & \text{for odd } k, \\ 2 \cdot \sum_{i=1}^{k/2} \frac{i \cdot (n-k)}{k+1} + \frac{k/2+1}{k+1} \cdot (n-k), & \text{for even } k, \end{cases}$$

and a simple calculation shows

$$E(PMA_n) = \begin{cases} \frac{k+3}{4} \cdot (n-k), & \text{for odd } k, \\ \left(\frac{k+3}{4} + \frac{3}{k+1} \right) \cdot (n-k), & \text{for even } k. \end{cases} \tag{7.9}$$

We calculated the average number of memory accesses for $k \in \{1, \ldots, 9, 15, 31, 63, 127\}$ using (7.7), (7.8), (7.9), and (6.2). Table 7.2 shows the results of these calculations. As before, with respect to the average assignment count, Algorithm 1 benefits from large k. For $k \in \{1, 2\}$ it has very high cost. For $k \geq 5$, it outperforms classical quicksort. Starting from $k \geq 15$ (for values of k considered in the calculations) it improves over Algorithm 3. Algorithm 2 also benefits from k getting larger. For large k, it makes about $5/3$ times as many memory accesses as Algorithm 1. More interestingly, and in big difference to counting assignments, Algorithm 3 actually improves

over classical quicksort when using more than one pivot. A 3-pivot quicksort algorithm, using this partitioning algorithm, has lower cost than classical and dual-pivot quicksort. Interestingly, the average number of memory accesses is minimized by the 5-pivot partitioning algorithm. The difference to the 3-pivot algorithm is, however, only small. Using more than 5 pivots increases the average number of memory accesses. Since each memory access, even when it can be served from L1 cache, is much more expensive than other operations like simple subtraction or addition on registers, this shows that are big differences in the time the CPU has to wait for memory between multi-pivot quicksort algorithms.

We now face the question what the considerations we have made so far could mean for the cache behavior. Intuitively, fewer memory accesses should yield better cache behavior, when memory accesses are done "scan-like" as in the algorithms considered here. The argument used in [LL99] and [Kus+14] is as follows: When each of the m cache memory blocks holds exactly B keys, then a scan of n' array cells (that have never been accessed before) incurs $\lceil n'/B \rceil$ cache misses. In theoretical models that allow control over the cache replacement strategy, this can easily be proven to be true for the algorithms considered here. However, suppose that k is large and we use Algorithm 1. With respect to the fact that a memory block can only be placed into a small number of different cache lines, it seems hard to believe that such a simple argument should hold. For example, suppose "many" elements have been moved in the loop in Line 11 of Algorithm 1. Then the access to $A[\texttt{from}]$ on Line 20 might incur a second cache miss, since the last access to this memory cell lies far away in the past. Moreover, when k is "large" it might also happen that particular segments are stored in cache lines and get evicted before they are read from/written to again. We observe that in Algorithm 1 the situation is very much like having $k+1$ sequences of total length n' that are scanned concurrently. The decision which pointer is to be moved next is based on the classification of an element. This problem (with an adversary that picks the pointer to be advanced next at random) has been studied by Mehlhorn and Sanders in [MS03]. Assuming that the starting addresses of these sequences are random they showed that the cache misses incured by such a scanning task are bounded by $O(n'/B)$ as long as $k = O(m/B^{1/a})$, where a is the associativity of the cache. For example, the L1 cache of the Intel i7 CPU used in our experiments can store $m = 512$ cache lines of size $B = 64$ byte and is 8-way associative. In this case, $m/B^{1/a}$ is about 300, so we may assume that as long as k is small enough, our assumption for the relation between memory accesses and cache misses only is a constant factor higher than our estimate. Of course in our setting we care about these constants. Consequently, we are going to compare our estimate to measurements from experiments.

Empirical Verification. We implemented multi-pivot quicksort algorithms using Algorithm 1, Algorithm 2 and Algorithm 3, resp., for partitioning. For Algorithm 1 and Algorithm 2, "Permute$_k$" and "Copy$_k$" denote the variants that classify each element twice: once during the classification phase and once during the partitioning phase. As we shall see later, it will be beneficial to consider these strategies with the modification that element groups are stored in the classification phase. Consequently, "Permute$'_k$" and "Copy$'_k$" are the variants that store the element groups

Algorithm	Ex_1	Ex_2	Ex_5	Ex_9	Perm_1	Perm_7	Perm_{31}	Perm_{127}
avg. L1 misses / n	0.125	0.163	0.25	0.378	0.25	0.25	0.25	0.28

Table 7.3.: Cache misses incured by Algorithm 1 ("Perm_k") and Algorithm 3 ("Ex_k") in a single partitioning step. All values are averaged over 600 trials.

in an auxiliary array and use these classifications as an oracle in the second phase. (Note that this introduces a linear memory overhead, which is often considered undesirable.) We measured cache misses for each algorithm on inputs of size 2^i with $9 \leq i \leq 27$, for certain values of k. These measurements were obtained with the help of the "performance application programming interface" (PAPI), which is available at `http://icl.cs.utk.edu/papi/`. Independently, we measured the number of misses in the translation-lookaside buffer (TLB) using the linux tool "*perf*".

First, we check whether the assumption that partitioning an input of n elements using Algorithm 1 or Algorithm 3 incurs $\lceil \mathrm{E}\left(\mathrm{PMA}_n\right)/B \rceil$ cache misses or not. (Recall that B is the number of elements in one cache line and $\mathrm{E}\left(\mathrm{PMA}_n\right)$ is the average number of memory accesses during partitioning.) In the experiment, we partitioned 600 inputs consisting of $n = 2^{27}$ items using Algorithm 1, for $1, 7, 31$, and 127 pivots, and Algorithm 3, for $1, 2, 5$, and 9 pivots. The measurements with respect to L1 cache misses are shown in Table 7.3. Theoretically, Algorithm 3 should incur $0.125n$, $0.166n$, $0.25n$, and $0.375n$ L1 cache misses for $k \in \{1, 2, 5, 9\}$, respectively. The results from Table 7.3 show that the empirical measurements are very close to these values. On the other hand, the measurements for Algorithm 1 are much lower than what we would expect by calculating $\mathrm{E}\left(\mathrm{PMA}_n\right)/B$, *cf.* Table 7.2. This is easily explained: We have charged two array accesses for a misplaced element. However, the second array access should always be cached, see Line 18 in Algorithm 1. Keeping this in mind, we should assume that the algorithm requires $2n$ memory accesses, which reflects the measurements, except for the case of 127 pivots. There we have to deal with the problem mentioned on the previous page: Cache blocks are evicted before they are accessed again. However, even for such a large number of pivots our prediction is accurate.

Table 7.4 shows the exact measurements regarding L1 cache misses for sorting 600 random inputs consisting of $n = 2^{27}$ elements and relates them to each other. We first consider our results with respect to Algorithm 3 ("Exchange_k"). The figures indicate that the relation with respect to the measured number of L1 cache misses of the different algorithms *exactly* reflect their relation with respect to the average number of memory accesses. However, while the average number of cache misses correctly reflects the relative relations, the measured values (scaled by $n \ln n$) are lower than we would expect by simply dividing $\mathrm{E}(\mathrm{MA}_n)$ by the block size B. We suspect this is due to (i) the influence of lower order terms and (ii) array segments considered in the recursion already being present in cache. For variants of Algorithm 1, the relation with respect to memory accesses predicts cache behavior correctly, as well. However, the exact ratio with respect to memory accesses does not translate to the ratio between cache misses. (It should be noted that

Algo	E(MA_n)	L1 Cache Misses
Exchange_1	$2n \ln n$ (+ 45.0%)	$0.14n \ln n$ (+ 48.9%)
Exchange_2	$1.6n \ln n$ (+ 16.0%)	$0.11n \ln n$ (+ 16.9%)
Exchange_3	$1.385n \ln n$ (+ 0.4%)	$0.096n \ln n$ (+ 1.3%)
Exchange_5	$1.379n \ln n$ (−)	$0.095n \ln n$ (−)
Exchange_7	$1.455n \ln n$ (+ 5.5%)	$0.1n \ln n$ (+ 5.3%)
Exchange_9	$1.555n \ln n$ (+ 12.8%)	$0.106n \ln n$ (+ 12.2%)
Permute_1	$4.66n \ln n$ (+237.9%)	$0.29n \ln n$ (+177.5%)
Permute_3	$2.4n \ln n$ (+ 74.0%)	$0.17n \ln n$ (+ 67.1%)
Permute_7	$1.62n \ln n$ (+ 17.5%)	$0.098n \ln n$ (+ 3.2%)
Permute_{15}	$1.21n \ln n$ (− 14.0%)	$0.07n \ln n$ (− 36.6%)
Permute_{31}	$0.96n \ln n$ (− 43.6%)	$0.06n \ln n$ (− 66.9%)
Permute_{63}	$0.79n \ln n$ (− 74.6%)	$0.05n \ln n$ (− 90.9%)
Permute_{127}	$0.67n \ln n$ (−105.8%)	$0.05n \ln n$ (− 99.3%)
$\text{Permute}'_{127}$	$1.12n \ln n$ (− 23.1%)	$0.067n \ln n$ (− 49.5%)
Copy'_{127}	$1.575n \ln n$ (+ 14.2%)	$0.11n \ln n$ (+ 18.9%)

Table 7.4.: Average number of L1 cache misses compared to the average number of memory accesses. Measurements have been obtained for $n = 2^{27}$. Cache misses are scaled by $n \ln n$. In parentheses, we show the ratio to the best algorithmic variant of Algorithm 3 w.r.t. memory/cache behavior ($k = 5$), calculated from the non-truncated experimental data.

the ordering with respect to memory accesses equals—only with the exception of Permute_7—the ordering with respect to L1 cache misses.) We also tested the variant of the Permute_k algorithm that stores the groups of elements in the first pass. Of course, $\text{Permute}'_{127}$ makes more cache misses than Permute_{127}. By using one byte per element group—which is sufficient for at most 255 pivots—, it incurs about 33% more L1 cache misses than Permute_{127}. Finally, we consider one specific variant of Algorithm 2 ("Copy_k") that also stores element groups. For the variant "Copy'_{127}", results are not surprising. It incurs about twice as many cache misses as Permute_{127} with an additional overhead due to the auxiliary array. In summary, memory accesses are a suitable cost measure to predict the L1 cache behavior of Algorithm 1—3. (For Algorithm 1 and Algorithm 2 some manual tweaking was necessary.) However, this is not true with regard to L2 and L3 cache behavior of these algorithms. In our experiments, the best algorithm with respect to L2 cache behavior was Permute_{15}. The worst algorithm due to L2 cache behavior was Copy'_{127}, incuring more than 6 times more L2 cache misses than $\text{Permute}'_{15}$. This picture was strengthend even more for L3 cache behavior. There, Copy'_{127} made about 10 times as many cache misses as Permute_{15}, which in turn was only slightly worse than Exchange_7, the best algorithm with respect to L3 cache behavior. Detailed experimental data can be found in Table B.2 in Appendix B.

To find out how large values of k make partitioning hard, we now consider the behavior of the algorithms regarding load misses in the translation-lookaside buffer (TLB misses). (Recall that the TLB is the cache that speeds up the translating between virtual and physical address

Algorithm	avg. TLB load misses
Exchange_1	$0.0407n \ln n$(0.0%)
Exchange_9	$0.0412n \ln n$(1.2%)
$\text{Permute}'_7$	$0.0421n \ln n$(3.5%)
Permute_{15}	$0.0416n \ln n$(2.2%)
$\text{Permute}'_{15}$	$0.0498n \ln n$(22.4%)
Permute_{127}	$0.0716n \ln n$(76.0%)
$\text{Permute}'_{127}$	$0.1203n \ln n$(195.7%)
Permute_{512}	$0.0873n \ln n$(114.5%)
$\text{Permute}'_{512}$	$0.1401n \ln n$(244.4%)
Copy'_{127}	$0.041n \ln n$(0.8%)

Table 7.5.: Average number of TLB misses for random inputs with 2^{27} items over 100 trials. Load misses are scaled by $n \ln n$. The number in parentheses shows the relative difference to algorithm Exchange_1.

space.) Figure 7.8 shows the measurements we got for some selected algorithms. (The results for all algorithms can be found in Table B.3 in Appendix B.) For arrays with no more than 2^{22} items, no misses in the TLB occur. This drastically changes for larger inputs. Each algorithm suffers from a growing number of TLB misses. The algorithms based on Exchange_k, and the algorithms Copy'_{127}, $\text{Permute}'_7$, and Permute_{15} suffer the fewest TLB misses. For larger k, algorithms based on Permute_k or $\text{Permute}'_k$ incur much more TLB misses. For example, Permute_{127} suffers 1.76 times more TLB misses than Exchange_1. $\text{Permute}'_{512}$ shows the worst behavior with respect to TLB misses, incuring 3.44 times more TLB misses than Exchange_1. This shows that algorithms based on "Permute_k" suffer in performance for large k because of TLB misses. More detailed results with respect to TLB misses are shown in Table 7.5.

In summary this section described general partitioning strategies for multi-pivot quicksort. We considered the average number of assignments and the average number of memory accesses. We have shown that studying memory accesses allows us to predict empirical differences in cache misses between these algorithms. For a small number of pivots, none of these strategies make fewer assignments than classical quicksort. With respect to memory accesses and cache misses, Algorithm 3 ("Exchange_k") can improve on classical quicksort and shows very good memory behavior for three or five pivots. For a large number of pivots, Algorithm 1 ("Permute_k") improves over classical quicksort and over Algorithm 3 in general. However, for a large number of pivots it incurs many TLB misses. Algorithm 2 ("Copy_k") uses a simpler partitioning strategy that avoids problems regarding the TLB even for a large number of pivots, but has worse cache behavior than Algorithm 1.

In the next section, we will consider running time measurements of multi-pivot quicksort algorithms and try to explain our findings by linking them to the theoretical cost measures considered here.

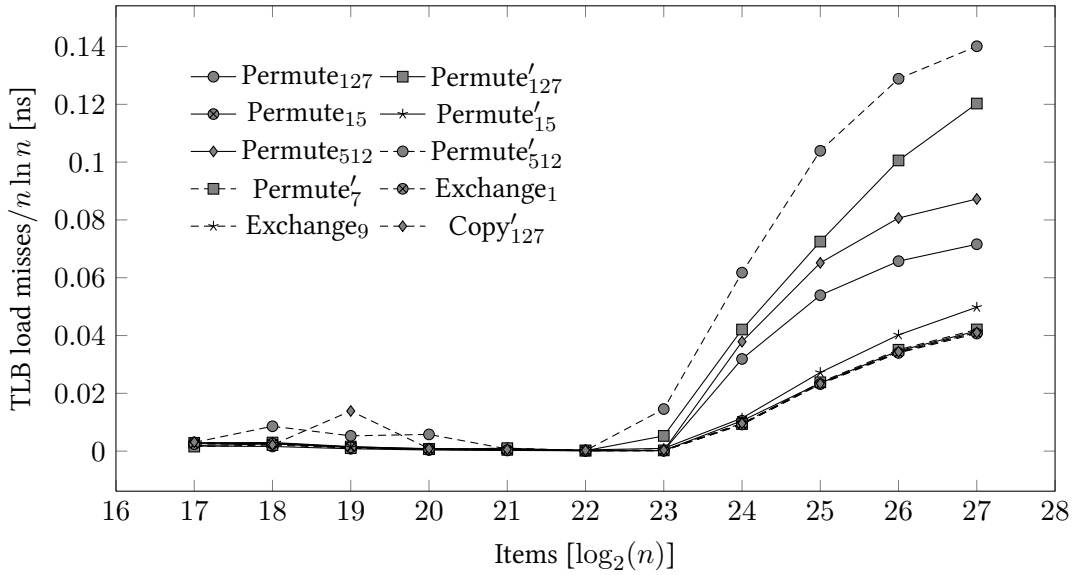

Figure 7.8.: TLB misses for Algorithms 1–3. Each data point is averaged over 500 trials, TLB load misses are scaled by $n \ln n$.

8. Running Time Experiments

We have implemented the methods presented in this thesis in C++. Details about the machine used in our experiments can be found in Section 1. For compiling C++ code, we used *gcc* in version 4.8. We did no manual tweaking to the produced assembler code. We used the compiler flags *-O3* and *-funroll-loops*. The option *-funroll-loops* tells the compiler to "optimize" loop statements, e. g., by unrolling the loop body for loops which consist only of a few iterations. (In general this might slow an algorithm down.) In all settings, we used *-march=native*, which means that the compiler tries to optimize for the specific CPU architecture we use during compilation. We remark that these optimization flags have a big impact on observed running times. While there is only a small difference between the settings *-O2* and *-O3* in our setup, some algorithms benefit significantly from unrolling loops. The influence will be described later in more detail. However, we stress that our results do not allow final statements on the running time behavior of quicksort variants. (Since such a small compiler flag has such an impact on running time.) The source code of our algorithms can be found at `http://eiche.theoinf.tu-ilmenau.de/maumueller-diss/`. The experimental framework to measure running time and generate inputs is based on source code written by Timo Bingmann.

Since we consider many different algorithms, we structure this section as follows: First, we consider the dual-pivot quicksort strategies from Section 4. Next, we consider k-pivot quicksort algorithms based on the partitioning algorithm "Exchange$_k$" (Algorithm 3). Subsequently, we will compare k-pivot quicksort algorithms based on algorithm "Permute$_k$" (Algorithm 1) and "Copy$_k$" (Algorithm 2). At the end, we will summarize our findings with respect to the running time of multi-pivot quicksort algorithms.

In each experiment, we sort random permutations of $\{1, \ldots, n\}$. Usually, we test input sizes $n = 2^i$, for $17 \leq i \leq 27$, and average our results over 500 trials. We consider the *significance* of running time differences by letting each algorithm sort the same 500 inputs containing 2^{27} items. This allows us to compare running times in more detail, for example, by saying that algorithm A was faster than algorithm B for at least 95% of the inputs.

Detailed experimental data has been moved to the appendix to keep this section readable. Appendix B contains exact measurements for all algorithms considered here.

8.1. Running Times of Dual-Pivot Quicksort Algorithms

For better readability, the algorithms considered in this section are presented in Table 8.1. Pseudocode for the dual-pivot methods is provided in Appendix A. In the following, we use a calli-

Abbreviation	Full Name	Strategy	Pseudocode
$\mathcal{Y}$	Yaroslavskiy's Algorithm	Section 4.1	Algorithm 6 (Page 207)
$\mathcal{L}$	Larger Pivot First	Section 4.1	Algorithm 7 (Page 208)
$\mathcal{SP}$	Sample Algorithm	Section 4.2	Algorithm 10 (Page 212)
$\mathcal{C}$	Counting Algorithm	Section 4.2	Algorithm 11 (Page 213)

Table 8.1.: Overview of the dual-pivot quicksort algorithms considered in the experiments.

graphic letter both for the classification strategy and the actual dual-pivot quicksort algorithm.

The running time results we obtained are shown in Figure 8.1. We see that Yaroslavskiy's algorithm and the simple strategy $\mathcal{L}$ ("*Always compare to the larger pivot first*") are the fastest algorithms. The comparison-optimal sampling algorithm $\mathcal{SP}$ cannot compete with these two algorithms with respect to running time. On average it is about 5.1% slower than algorithm $\mathcal{Y}$. The slowest algorithm is the counting algorithm $\mathcal{C}$; on average it is about 14.3% slower than $\mathcal{Y}$. We see that only the running time of strategy $\mathcal{SP}$ seems to be affected by the input size. This is due to the fact that it sorts inputs that contain at most 1024 items with Yaroslavskiy's algorithm, which makes it faster for small inputs. (For such small inputs, the sampling step adds too much overhead.) We note that our implementations of dual-pivot quicksort algorithms did not benefit from loop unrolling.

Now, we consider the significance of differences in running time. In Table 8.2 we consider the number of cases which support the hypothesis that an algorithm is a given percentage faster than another algorithm. The table shows that the difference in running time is about 1% smaller than the average running time suggested if we consider only "significant" running time differences, i. e., differences that were observed for at least 95% of the inputs. In particular, we conclude that there is no significant difference in running time between $\mathcal{L}$ and $\mathcal{Y}$. This result surprises, for algorithm $\mathcal{Y}$ makes fewer comparisons ($1.9n \ln n$ vs. $2n \ln n$) than algorithm $\mathcal{L}$. Furthermore, both algorithms require the same number of assignments and have similar cache behavior. From Table B.4 (in Appendix B) we conclude that $\mathcal{L}$ executes about 10% fewer instructions than $\mathcal{Y}$. This is mainly caused by avoiding the test whether or not the two pointers that move towards each other have crossed more often in $\mathcal{L}$. (See Line 2 in Algorithm 6 on Page 207 and the same line in Algorithm 7.) Since these instructions are fairly simple and predictable, the difference in instruction count does not translate into significantly better running time.

8.2. Running Times of k-Pivot Quicksort Algorithms based on "Exchange$_k$"

We now consider the running times of k-pivot quicksort algorithms implementing the partitioning strategy "Exchange$_k$" (Algorithm 3), for $k \in \{1, 2, 3, 5, 7, 9\}$. We use classical quicksort for $k = 1$, we use strategy $\mathcal{L}$ for $k = 2$. For $k = 3$, we use the recently discovered algorithm of

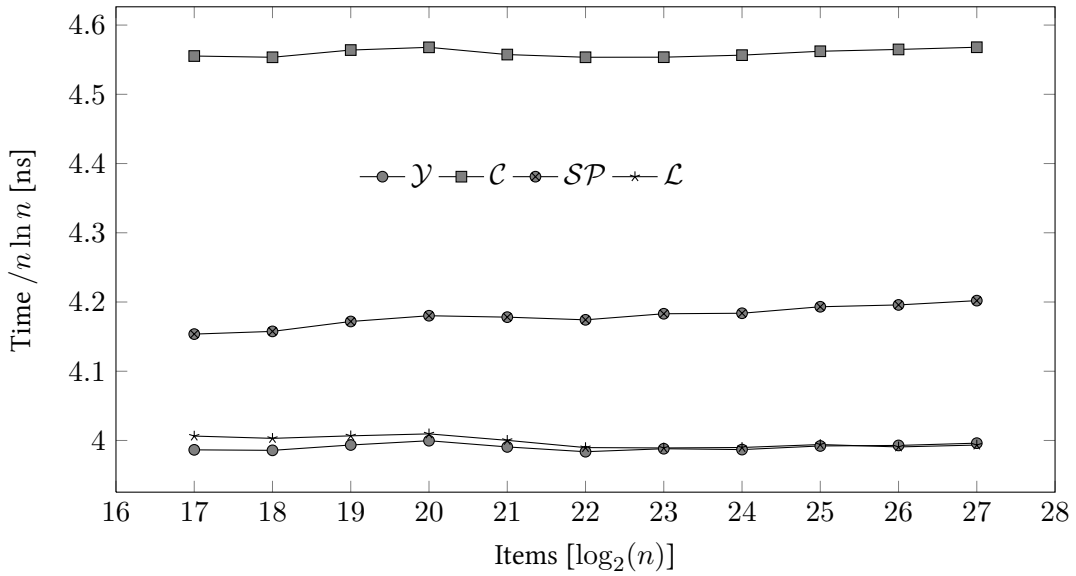

Figure 8.1.: Running time experiments for dual-pivot quicksort algorithms. Each data point is the average over 500 trials. Times are scaled by $n \ln n$.

Kushagra *et al.* [Kus+14], which combines Algorithm 3 with the symmetrical comparison tree l_2 from Figure 6.3. For 5, 7, and 9 pivots, we use Algorithm 3 with the comparison trees depicted in Figure 8.2. We remark that the source code becomes quite complicated for algorithms based on Exchange$_k$ for large k. For example, the implementation of "Exchange$_9$" with the comparison tree from Figure 8.2 has about 400 lines of C++ code.

The results of our experiments can be seen in Figure 8.3. With respect to the average running time, we see that the 3-pivot algorithm of Kushagra *et al.* and the dual-pivot algorithm $\mathcal{L}$ are the fastest algorithms. All other algorithms are significantly slower. Among the remaining algorithms, classical quicksort is slightly faster than 5-pivot quicksort. The 7-pivot algorithm and the 9-pivot algorithm are slowest. With respect to significant differences in running time, i. e., running times observed for at least 95% of the test inputs, we cannot spot a difference between the 2- and 3-pivot algorithm. Classical quicksort, the 5-pivot algorithm, the 7-pivot algorithm, and the 9-pivot algorithm are 6.5%, 7.7%, 13.7%, and 16.5% slower than the 3-pivot algorithm, respectively. The scaled times are almost constant for all algorithms.

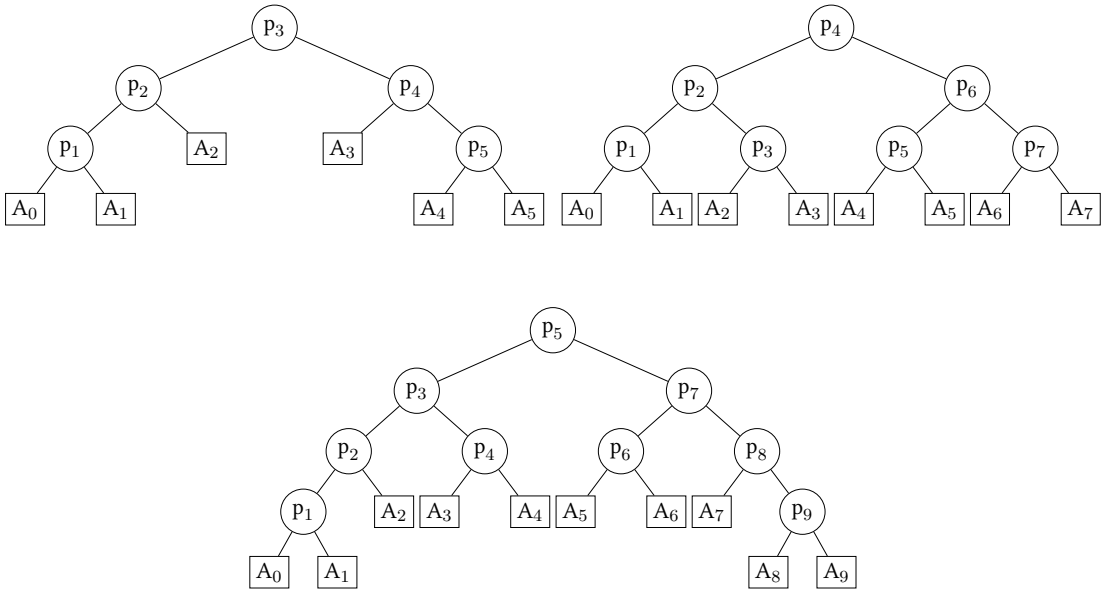

Figure 8.2.: The comparison trees used for the 5-,7-, and 9-pivot algorithms.

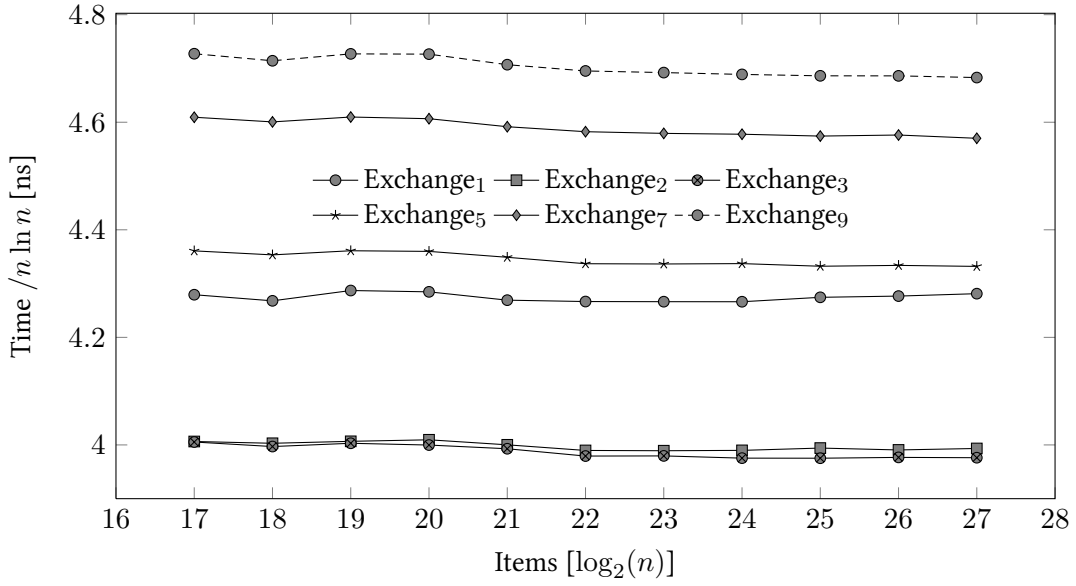

Figure 8.3.: Running time experiments for k-pivot quicksort algorithms based on the "Exchange$_k$" partitioning strategy. Each data point is the average over 500 trials. Times are scaled by $n \ln n$.

	$\mathcal{Y}$	$\mathcal{L}$	$\mathcal{SP}$	$\mathcal{C}$
$\mathcal{Y}$	–	–/–/0.5%	4.5%/5.1%/5.9%	13.4%/14.3%/15.6%
$\mathcal{L}$	–/0.1%/0.7%	–	4.2%/5.1%/6.8%	13.3%/14.3%/15.9%
$\mathcal{SP}$	–	–	–	7.8%/8.6%/9.9%
$\mathcal{C}$	–	–	–	–

Table 8.2.: Comparison of the actual running times of the algorithms on 500 different inputs of size 2^{27}. A table cell in a row labeled "A" and a column labeled "B" contains a string "$x\%/y\%/z\%$" and is read as follows: "In about 95%, 50%, and 5% of the cases algorithm A was more than x, y, and z percent faster than algorithm B, respectively."

8.3. Running Times of k-Pivot Quicksort Algorithms based on "Permute$_k$" and "Copy$_k$"

Here we consider the running times of k-pivot quicksort algorithms implementing the partitioning strategies "Permute$_k$" (Algorithm 1) and "Copy$_k$" (Algorithm 2), respectively. We first remark that both algorithms are only competitive when element groups are stored during the classification phase. When classifying each element twice, the running times of all these algorithms are a factor of at least 1.7 higher than the running time of classical quicksort. One byte per element suffices to store the outcome of the classification for fewer than 256 pivots. When sorting 64-bit integers as in our experiments, the memory overhead is thus roughly 12.5%. In the further discussion, we assume element groups to be stored. We refer to the algorithms by Permute$'_k$ and Copy$'_k$.

In our experiments, variants of Permute$'_k$ and Copy$'_k$ which use fewer than seven pivots were much slower than the algorithms based on Exchange$_k$. Consequently, we omit the results here and report on the results we obtained for $k \in \{7, 15, 31, 127, 255, 511\}$. The group of an element is determined using the obvious symmetrical comparison tree for $2^\kappa - 1$ pivots, for $\kappa \geq 1$, in which all leaves are on the same level. For the implementation of this strategy, we used a nice trick due to Sanders and Winkel [SW04] communicated to us with source code by Timo Bingmann. We store the symmetrical classification tree implicitly in an array as it is known from binary heaps, i. e., the left and right child of a node stored at position j in the array is at positions $2j$ and $2j + 1$ in the array, respectively. For the classification of an element, we use a standard binary search in this implicit representation. Suppose for a specific element this binary search ended at position j with $j > k$. The group the element belongs to is then $j - k$. This classification strategy does incur only few branch mispredictions on modern CPU's, because the decision whether to continue at array position $2j$ or $2j + 1$ after comparing an element with the pivot at array position j can be implemented by a *predicated move*. (This was done automatically by the compiler in our experiments.) We used the source code of Timo Bingmann for the implementation of the classification strategy. We remark that algorithms based on "Permute$_k$" and "Copy$_k$" strongly benefit from loop unrolling.

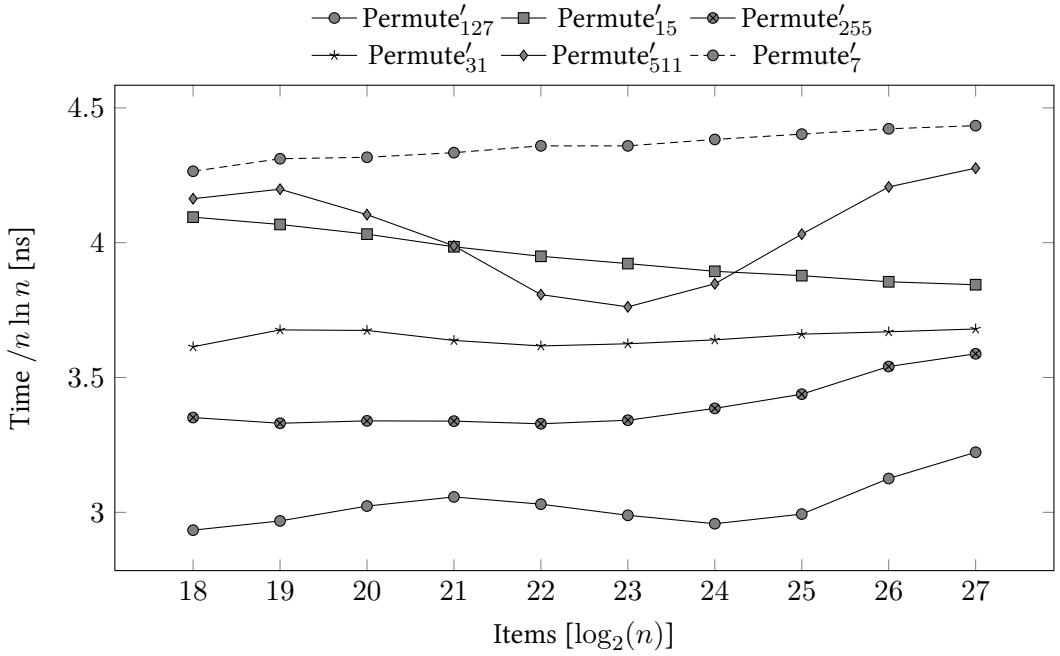

Figure 8.4.: Running time experiments for k-pivot quicksort algorithms based on the "Permute$_k$" partitioning algorithm. Each data point is the average over 500 trials. Times are scaled by $n \ln n$.

Figure 8.4 shows the results of our experiments with respect to algorithms based on Permute$'_k$. We see that the variant using 127 pivots provides the best running times. For $n = 2^{27}$, it is about 10.5%, 12.5%, 17.5%, 31%, and 34% faster, for at least 95% of the inputs, than the variants using 255, 31, 15, 511, and 7 pivots, respectively. Furthermore, the 15-pivot variant becomes faster for larger n. On the other hand, the 31-pivot variant becomes slightly slower. The 127- and 255-pivot algorithms and especially the 511-pivot algorithm become slower for larger inputs. We suspect that this is due to misses in the TLB, as studied in the previous section. From that section we also known that TLB misses do not have a strong impact for algorithms based on Copy$'_k$. Our experiments show that for this variant, using 127 pivots is also the best choice.

Last of all, we compare Copy$'_{127}$, which is the super scalar sample sort algorithm of Sanders and Winkel [SW04], with the fastest algorithm based on the Exchange$_k$ strategy ("Exchange$_3$") and the fastest algorithm based on the Permute$_k$ strategy ("Permute$'_{127}$"). For reference to library algorithms, we also show the results we got with respect to the well-engineered quicksort variant in the C++ standard library (`std::sort`). (This algorithm is a variant of introsort, a quicksort variant with worst-case $O(n \log n)$ running time [Mus97].)

The result of this experiment is shown in Figure 8.5. The super scalar sample sort algorithm of Sanders and Winkel is fastest in our setup. For $n = 2^{27}$ it is on average about 17.3% faster

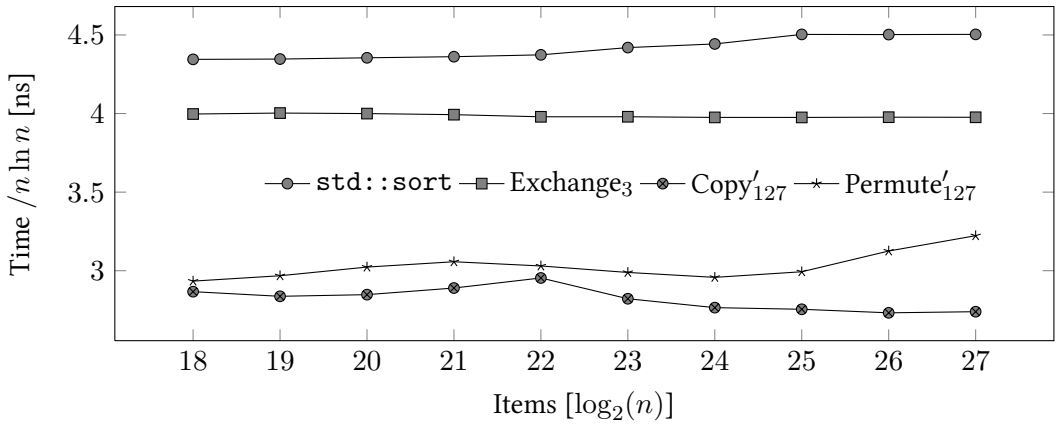

Figure 8.5.: Final Running time experiments for k-pivot quicksort algorithms based in C++. Each data point is the average over 500 trials. Times are scaled by $n \ln n$.

than Permute$'_{127}$, which needs roughly half the space. (Only classification results are stored.) Since our implementation of Copy$'_{127}$ and Permute$'_{127}$ only differ in the partitioning phase, this strongly supports the hypothesis that the running time of Permute$'_{127}$ is strongly influenced by TLB misses. Exchange$_3$ needs no additional space and is about 23.4% slower than Permute$'_{127}$. Both Copy$'_{127}$ and Permute$'_{127}$ benefit from using Exchange$_3$ for handling small subarrays of size at most 1024. `std::sort` is the slowest algorithm, being about 13.3% slower than Exchange$_3$.

In summary, the answer to the question *"Which is the fastest quicksort variant?"* strongly depends on the amount of additional space one is willing to allocate. Only considering variants that work *in-place* (except for the recursion stack), the three-pivot algorithm of Kushagra *et al.* [Kus+14] seems to be the best choice. In our setup, there is almost no difference in running time to the dual-pivot algorithms $\mathcal{Y}$ and $\mathcal{L}$. If we allow an additional overhead of one byte per item, running times greatly improve by using Permute$'_{127}$. However, we suspect that the behavior of this strategy with regard to TLB misses could make it slower than algorithms based on Exchange$_k$ on some architectures. Furthermore, it is important that the CPU supports the *predicated move* instruction to save branch mispredictions. Finally, if space is no limiting factor, then Copy$'_{127}$, i. e., the super scalar sample sort algorithm of Sanders and Winkel from [SW04] is the method of choice. With such a large space overhead other sorting methods, e. g., radix sort variants, should also be considered when sorting integers. ([Big+08] report that on their setup a radix sort variant was faster than classical quicksort.) We did not test these methods due to time constraints.

In the last section, we try to link our experimental findings to our theoretical cost measures.

8.4. Do Theoretical Cost Measures Help Predicting Running Time?

The running time experiments showed that multi-pivot quicksort makes it possible to achieve a better running time than classical quicksort, as observed in Java with the introduction of Yaroslavskiy's algorithm as the standard sorting algorithm in Java 7, and in the paper [Kus+14]. We now evaluate how the theoretical performance of an algorithm coincidences with its running time in practice.

Combining the theoretical cost measures "comparisons", "assignments", and "memory accesses", algorithms using the "Permute$_k$" partitioning strategy should—for k large enough—outperform algorithms based on the "Exchange$_k$" partitioning strategy. We have observed this in our experiments. However, it was necessary to store the element groups. For a large number of pivots, the TLB adds a significant overhead to the running time of algorithms based on Permute$_k$.

With respect to differences in running times of the "Exchange$_k$"-based algorithms, the theoretical cost measures "memory accesses" and "assignments" show disadvantages of this partitioning strategy for larger k. This agrees with our measurements from Figure 8.3. So, these cost measures make us believe that the "Exchange$_k$" partitioning strategy is only fast for a small number of pivots. However, for small values of k these cost measures cannot explain specific observations, e. g., (i) why "Exchange$_9$" is significantly slower than classical quicksort ("Exchange$_1$"), and (ii) why there is no significant difference between the 2- and 3-pivot quicksort algorithm.

In our tests, we also counted the average number of instructions and the average number of branch mispredictions, see Table B.4 on Page 219 for details. We believe that a theoretical study on the average number of instructions in the style of Wild *et al.* [WNN13] would have been beneficial to explain our findings. From our measurements, "Exchange$_3$" executes fewest instructions, closely followed by $\mathcal{L}$. Also, Permute$'_{127}$ and Copy$'_{127}$ executes the fewest instructions of the tested algorithms based on the strategies Permute$'_k$ and Copy$'_k$, which nicely reflects the empirical running time behavior.

With respect to branch mispredictions, we see that implementing the binary search in the symmetrical classification tree by predicated moves decreases the average number of branch mispredictions. (Variants based on the Exchange$_k$ strategy incur almost four times as much branch mispredictions as algorithms based on Permute$_k$ and Copy$_k$.) From differences in branch mispredictions, one might also find reasons for Exchange$_1$ being faster than Exchange$_9$. (In our experiments, Exchange$_1$ makes $0.1n \ln n$ fewer branch mispredictions than Exchange$_9$, while having almost the same average instruction count.)

Linking our measurements of cache misses and TLB misses with the often known penalties for these events, we can also speculate about the number of CPU cycles the algorithm has to wait for memory. Dividing these numbers by the total number of CPU cycles needed to sort the input gives us an idea of how much of the sorting time is mandatory, i. e., cannot be avoided. The exact description of the methodology and detailed results of this approach can be found in Appendix B. Our basic results are as follows. Copy$'_{127}$ shows the highest ratio of CPU cycles necessary for memory accesses divided by the total number of CPU cycles needed for sorting. In this algorithm, about 92% of the CPU cycles needed for sorting the input are necessary for

memory accesses anyway. (This is achieved by "decoupling" the classifications, because each element can be classified independently in the first pass.) In comparison, only 47% of the CPU cycles have to be used just for memory accesses by Exchange$_3$. For classical quicksort, about 63% of the CPU cycles are mandatory for memory accesses.

9. Conclusion and Open Questions

In the first part of this thesis, we studied quicksort algorithms that use more than one pivot. Motivated by the recently discovered dual-pivot algorithm of Yaroslavskiy [Yar09] and the three-pivot algorithm of Kushagra *et al.* [Kus+14], we provided a detailed analysis of multi-pivot quicksort algorithms w.r.t. three different cost measures: comparisons, assignments, and memory accesses.

We have described natural strategies that achieve the minimal possible average comparison count for k-pivot quicksort. These strategies either count the group sizes observed so far or use a small sampling step to decide how to classify the next element. More generally, we showed how to calculate the average comparison count of a multi-pivot quicksort algorithm. The calculation turned out to be difficult and we were only able to estimate the minimal average comparison count of multi-pivot quicksort for the case of using at most three pivots. For more than three pivots, we resorted to experiments to obtain rough approximations of the minimum average comparison count. This led us conjecture that optimal k-pivot quicksort is inferior to the standard median-of-k approach for classical quicksort. Already for four pivots, optimal classification strategies were too complex to yield improvements in empirical running time.

Next, we studied the cost of the actual partitioning step with respect to the average assignment count and the average number of memory accesses. We described three general partitioning algorithms. The first two algorithms partitioned the input in two passes, classifying the input in the first pass and producing the actual partition in the second pass. One of these strategies obtained this partition with an in-place permutation, the other strategy allocated a new array. These strategies turned out to make fewer assignments, memory accesses and L1 cache misses than classical quicksort when used with many pivots. Our experiments showed that it is necessary to store the element classifications after the first pass to make these algorithms competitive. Then, both algorithms were much faster than classical quicksort in practice, on the cost of an additional memory overhead. We also studied a partitioning strategy that produced the partition in a single pass, generalizing the partitioning strategy of classical quicksort, Yaroslavskiy's algorithm and the three-pivot algorithm of Kushagra *et al.* [Kus+14]. This strategy showed very good cache behavior when used with three or five pivots. In experiments, the variants using two and three pivots were the fastest algorithms, but were slower than the two-pass algorithms. We saw that memory accesses predicted the running time differences in many cases very well.

In addition to the open questions from Section 6.6, we pose the following directions for future work:

1. Usually, pivots are chosen from a sample to balance the size of subproblems, as shown for

dual-pivot quicksort in Section 5. It would be interesting to see how the theoretical cost measures change when using pivot sampling in a multi-pivot quicksort algorithm.

2. Memory accesses could not explain the L2 and L3 cache behavior of our algorithms. It would be interesting to see how this can be analyzed.

3. Our partitioning algorithms were not optimal with respect to the average number of assignments they required to rearrange the input. (It is a simple exercise to describe inputs in which Algorithm 1 requires too many assignments.) It would be interesting to describe "assignment-optimal" multi-pivot quicksort algorithms. Here it seems like one should look for permute sequence (in the sense of Algorithm 1) that are as long as possible.

4. Our running time experiments were conducted on random permutations of $\{1, \dots, n\}$. For practical purposes, other input distributions are also important, e. g., inputs with equal keys or inputs with some kind of "presortedness".

Part II | Hashing

10. Introduction

Hashing is a central technique in the design of (randomized) algorithms and data structures. It finds application in such diverse areas as hash tables, load balancing, data mining, and machine learning. The basic idea of hashing is to map elements from a (usually very large) universe U to some smaller range R. To simplify the analysis of a hashing-based algorithm or data structure, one traditionally assumes that a hash function is "fully random", i. e., hash values are distributed uniformly and independently in the range R. Additionally, their use is "free of charge", i. e., a hash function consumes no space and its evaluation takes unit time. Unfortunately, such functions are not efficient, since their representation takes $|U| \log |R|$ bits. Consequently, many scientific papers were devoted to the construction of explicit hash functions, which are not fully random, but usually just "good enough" for running a specific application. This part of the thesis pursues exactly this goal.

While a considerable amount of CPU time is spent in storing and reading data from hash tables—many programming languages implement associative arrays as a standard data structure and recommended their use in most cases—, little effort is put into the calculation of hash values. Often, hash values are deterministic and collisions are easy to find. According to Pagh [Pag14], in Oracle's *Java 7* the hash value $h(x)$ of a string $x = a_1 \dots a_n$ follows the recursion $h(a_1 \dots a_n) = \text{ord}(a_n) + 31 \cdot h(a_1 \dots a_{n-1})$, with $h(\varepsilon) = 0$, in signed 32-bit arithmetic. Sets of elements that all have the same hash value are easy to find: First, observe that the strings "Aa" and "BB" collide. With this knowledge and the recursive formula from above, one can see that all strings $(\text{Aa}|\text{BB})^n$, for $n \geq 1$, collide as well. Since hash tables are often used by web servers to parse packets, attackers were able to render servers unusable with little traffic, see, e. g., [CW03]. As of today, at least three major programming languages adopted stronger hash functions (e. g., *Murmur3* [App] or *SipHash* [AB12]). These hash functions are nowadays *salted* with a random seed when starting the program to make it harder to find collisions among keys. In this thesis, we do not further discuss such hash functions; we only consider them in the experimental evaluation in the last section.

Hash functions considered in theory use randomization to avoid *worst-case* inputs, i. e., the use of the hash function will have provable guarantees on every possible input, which is a fundamental difference to deterministic hashing. The aim is to find practical, i. e., *fast* hash functions with provable theoretical guarantees. While researchers started to work on this task almost 35 years ago, many important open problems have only been solved recently. Selected results will be discussed in the next paragraphs.

Traditionally, explicit hash function constructions build upon the work of Carter and Wegman [CW79]. They proposed a technique called *universal hashing*. In universal hashing, a hash

function is picked at random from a set $\mathcal{H} \subseteq \{h \mid h\colon U \to R\}$. (We call such a set $\mathcal{H}$ a *hash family* or *hash class*.) The influential notions in universal hashing are "universality" and "independence", introduced by Carter and Wegman in [CW79]. The rigorous mathematical definition of these concepts will be provided in Section 11. Informally, $\mathcal{H}$ is called *universal* if choosing a hash function $h \in \mathcal{H}$ at random guarantees that the probability that for two distinct keys $x, y \in U$ we have $h(x) = h(y)$ is close to what we get in the fully random case. *Universality* of a hash family suffices for applications such as *chained hashing* where the expected number of colliding elements is central in the analysis. For a fixed integer k, we call $\mathcal{H}$ *k-independent*, if for a randomly chosen $h \in \mathcal{H}$ the hash values of each set of at most k distinct keys are uniform and independent. The canonical representation of a k-independent hash family is the family of all degree $k-1$ polynomials over some prime field. For the representation of such a polynomial, we just store its k coefficients (k words). The evaluation is possible in time $O(k)$. A large body of work has been devoted to the applicability of k-independent hash families. One of the most surprising results, due to Pagh, Pagh, and Ruciz [PPR09], is that 5-wise independence suffices for running linear probing—the most often used hash table implementation—, where "suffices" will always mean that the guarantees are close to what we would get when using fully random hash functions. Interestingly, this degree of independence is also necessary, for Pătraşcu and Thorup [PT10] constructed an artificial 4-wise independent hash family which does not allow running linear probing robustly. Another example where a constant degree of independence is sufficient is frequency estimation. In [AMS99], Alon, Matias, and Szegedy showed that 4-wise independence suffices for F_2-estimation. For such applications, both storing and evaluating the polynomial is possible in constant space and time. For many other applications, such as cuckoo hashing [PR04] and ε-minwise independent hashing [Ind01], we know that a logarithmic degree of independence suffices (in the size of the key set for the former, in $1/\varepsilon$ for the latter). In that case, polynomials use logarithmic space and evaluation time. If one aims for constant evaluation time, there exist the construction of Siegel [Sie04]—although Siegel states that his construction has constant albeit impractical evaluation time—and, more recently, the simple yet powerful construction of Thorup [Tho13].

Finding a proof that a certain degree of independence allows running a specific application has the advantage that one can choose freely from the pool of available hash families that achieve the necessary degree of independence. If a faster hash family becomes known in future research, one can just switch to this hash function. For example, this has happened with the introduction of Thorup and Zhang's fast tabulation-hashing class [TZ04; TZ12]. On the other hand, lower bounds on a certain degree of independence often use artificial constructions and do not rule out the possibility that "weak hash functions" (based on their universality or degree of independence) actually suffice for running a specific application. Notable exceptions are the analysis of Dietzfelbinger and Schellbach [DS09a; DS09b], who showed that cuckoo hashing cannot be run with the so-called class of linear hash functions and the class of multiplicative hash functions in certain situation, and Pătraşcu and Thorup [PT10], who demonstrated that linear probing is not robust when using the multiplicative class of hash functions.

Only in the last decade, the analysis of specific explicit hash families has been a fruitful re-

search area. Dietzfelbinger and Woelfel [DW03] showed in 2003 that a hash family introduced by Dietzfelbinger and Meyer auf der Heide in [DM90] allows running cuckoo hashing. In 2006, Woelfel [Woe06a] demonstrated that the same hash class could be used for running the `GoLeft` allocation algorithm of Voecking [Vöc03] in the area of load balancing. In 2011, Pătraşcu and Thorup [PT11] (full version [PT12]) analyzed a simple tabulation class of hash functions known at least since Zobrist's use of it in the 1970-ies [Zob70]. They proved that it has sufficient randomness properties in many applications, including static cuckoo hashing, linear probing, and ε-minwise independent hashing, despite of the fact that it is only 3-wise independent. In tabulation hashing, each key is a tuple $(x_1, \ldots, x_c)$ which is mapped to the hash value $f_1(x_1) \oplus \cdots \oplus f_c(x_c)$ by c uniform random hash functions $f_1, \ldots, f_c$, each with a domain of cardinality $U^{1/c}$. Two years later, the same authors introduced "twisted tabulation hashing" [PT13], which gives even stronger randomness properties in many applications. Recently, Dahlgaard, Knudsen, Rotenberg, and Thorup extended the use of simple tabulation to load balancing [Dah+14], showing that simple tabulation suffices for sequential load balancing with two choices. Furthermore, Dahlgaard and Thorup proved that twisted tabulation is ε-minwise independent [DT14]. While these hash functions provide constant evaluation time, their description length is polynomial in the size of the key set. With respect to description length, Celis, Reingold, Segev, and Wieder [Cel+13] presented a new hash class which is more concerned about space complexity. In 2014, Reingold, Rothblum, and Wieder [RRW14] showed that this class of hash functions has strong enough randomness properties for running a slightly modified version of cuckoo hashing and sequential allocation with two hash functions ("the power of two choices"). While it has non-constant evaluation time, its description length is notably smaller than what one gets using the standard polynomial approach ($O(\log n \log\log n)$ vs. $O(\log^2 n)$ bits).

Several techniques to circumvent or justify the uniform hashing assumption have been proposed. The most general one is to "simulate" uniform hashing. Suppose we want to construct a hash function that takes on fully random values from R. The idea is to generate a family H of hash functions at random such that with high probability H is "uniform" on $S \subseteq U$, which means that a random hash function $h \in H$ restricted to the domain S is a true random function. Such a simulation was presented by Pagh and Pagh in [PP08], by Dietzfelbinger and Woelfel in [DW03], and by Dietzfelbinger and Rink in [DR09]. In this thesis, we will provide a simple alternative construction which builds upon the work of [PP08]. However, such simulations require at least a linear (in $|S| \cdot \log |R|$) number of bits of additional space, which is often undesirable. Another perspective on uniform hashing is to assume that the key set $S = \{x_1, \ldots, x_n\} \subseteq U$ is "sufficiently random". Specifically, Mitzenmacher and Vadhan showed in [MV08] that when the distribution that governs $\{x_1, \ldots, x_n\}$ has a low enough collision probability, then even using a hash function h from a 2-wise independent hash class $\mathcal{H} \subseteq \{h \mid h\colon U \to R\}$ makes the sequence $(h, h(x_1), \ldots, h(x_n))$ distributed close to the uniform distribution on $\mathcal{H} \times R^n$ (see also [Die12]). An alternative is the so-called *split-and-share* technique [Fot+05; Die07; DR09; BPZ13], in which S is first partitioned by a top-level hash function into smaller sets of keys, called bins. Then, a problem solution is computed for each bin, but all bins share the same hash functions. Since the size of each bin is significantly smaller than the size of S, it is possible to use a hash function that

behaves like a true random hash function on each bin. Finally, the problem solution of all bins is combined to a solution of the original problem. This technique cannot be employed uniformly to all applications, as ad-hoc algorithms depending on the application are required to merge the individual solutions for each bin to a solution of the original problem. In some scenarios, e. g., balanced allocation with high loads, the small deviations in the bin sizes incurred by the top-level hash function are undesirable. Moreover, additional costs in space and time are caused by the top-level splitting hash function and by compensating for a larger failure probability in each of the smaller bins.

The Contribution. We generalize a hash family construction proposed by Dietzfelbinger and Woelfel in [DW03]. To put our contribution in perspective, we first review some background. Building upon the work of Dietzfelbinger and Meyer auf der Heide [DM90], Dietzfelbinger and Woelfel showed in [DW03] that a class of simple hash functions has strong randomness properties in many different applications, e. g., in standard cuckoo hashing [PR04], to simulate a uniform hash function, and in the context of simulations of shared memory situations on distributed memory machines. Their analysis is based on studying randomness properties of graphs built in the following way: Consider a set S of n keys chosen from a finite set U and a pair (h_1, h_2) of hash functions $h_1, h_2 \colon U \to [m] = \{0, \ldots, m-1\}$ for some positive integer m. Then, S and (h_1, h_2) naturally define a bipartite graph $G(S, h_1, h_2) := (V, E)$ with $V = V_{m,2}$, where $V_{m,2}$ is the union of two disjoint copies of $[m]$ and $E = \{(h_1(x), h_2(x)) \mid x \in S\}$. Dietzfelbinger and Woelfel studied the randomness properties of $G(S, h_1, h_2)$ when it is constructed using a certain explicit hash family. They showed that the connected components of this graph behave, in some technical sense, very close to what is expected of the graph $G(S, h_1, h_2)$ when h_1, h_2 were to be fully random. Later, Woelfel described in [Woe06a] how the construction from [DW03] extends to hypergraphs and analyzed the allocation algorithm of Voecking [Vöc03] using this hash class.

We extend the hash class described in [DW03; Woe06a] to a hash class we call $\mathcal{Z}$. We provide a general framework that allows us to analyze applications whose analysis is based on arguments on the random graph described above when hash functions from $\mathcal{Z}$ are used instead of fully random hash functions. To argue whether the hash class can run a certain application or not, only random graph theory is applied, no details of the actual hash class are exposed. Using this framework, we show that hash functions from $\mathcal{Z}$ have randomness properties strong enough for many different applications, e. g., cuckoo hashing with a stash as described by Kirsch, Mitzenmacher, and Wieder in [KMW09], generalized cuckoo hashing as proposed by Fotakis, Pagh, Sanders, and Spirakis in [Fot+05] with two recently discovered insertion algorithms due to Khosla [Kho13] and Eppstein, Goodrich, Mitzenmacher and Pszona [Epp+14] (in a sparse setting), the construction of a perfect hash function of Botelho, Pagh and Ziviani [BPZ13], the simulation of a uniform hash function of Pagh and Pagh [PP08], and different types of load balancing as studied by Schickinger and Steger [SS00]. The analysis is done in a unified way which we hope will be of independent interest. We will find sufficient conditions under which it is possible to replace the full randomness assumption of a sequence of hash functions with explicit hash functions. Furthermore, our

small modification of the construction of [DW03; Woe06a] makes the analysis easier and the hash functions faster in practice.

The General Idea. We will describe the class $\mathcal{Z}$ of hash function tuples $\vec{h} = (h_1, \ldots, h_d)$, $h_i \colon U \to [m]$. For each key $x \in U$, the hash function values $h_i(x)$ can be computed with a small (constant) number of arithmetic operations and lookups in small (cache-friendly) tables. For a set $S \subseteq U$ we then consider properties of the random graph $G(S, \vec{h})$, which is the obvious hypergraph extension of $G(S, h_1, h_2)$ to $d \geq 3$ hash functions, motivated by the following observation.

The analysis of hashing applications is often concerned with bounding (from above) the probability that random hash functions $h_1, \ldots, h_d$ map a given set $S \subseteq U$ of keys to some "bad" hash function values. Those undesirable events can often be described by certain properties exhibited by the random graph $G(S, \vec{h})$. For example, in the dictionary application cuckoo hashing, a bad event occurs when $G(S, h_1, h_2)$ contains a very long simple path or a connected component with at least two cycles.

If $h_1, \ldots, h_d$ are uniform hash functions, then often a technique called *first moment method* (see, e.g., [Bol85]) is employed to bound the probability of undesired events: In the standard analysis, one calculates the expectation of the random variable X that counts the number of subsets $T \subseteq S$ such that the subgraph $G\big(T, \vec{h}\big)$ forms a "bad" substructure, as e.g., a connected component with two or more cycles. This is done by summing the probability that the subgraph $G\big(T, \vec{h}\big)$ forms a "bad" substructure over all subsets $T \subseteq S$. One then shows that $\mathrm{E}(X) = O(n^{-\alpha})$ for some $\alpha > 0$ and concludes that $\Pr(X > 0)$—the probability that an undesired event happens—is at most $O(n^{-\alpha})$ by Markov's inequality.

We give general sufficient conditions allowing us to replace uniform hash functions $h_1, \ldots, h_d$ with hash function sequences from $\mathcal{Z}$ without significantly changing the probability of the occurrence of certain undesired substructures $G\big(T, \vec{h}\big)$. On a high level, the idea is as follows: We assume that for each $T \subseteq U$ we can split $\mathcal{Z}$ into two disjoint parts: hash function sequences being *T-good*, and hash function sequences being *T-bad*. Choosing $\vec{h} = (h_1, \ldots, h_d)$ at random from the set of T-good hash functions ensures that the hash values $h_i(x)$ with $x \in T$ and $1 \leq i \leq d$ are uniformly and independently distributed. Fix some set $S \subseteq U$. We identify some "exception set" $B_S \subseteq \mathcal{Z}$ (intended to be very small) such that for all $T \subseteq S$ we have: If $G(T, \vec{h})$ has an undesired property (e.g., a connected component with two or more cycles) and $\vec{h}$ is T-bad, then $\vec{h} \in B_S$.

For $T \subseteq S$, disregarding the hash functions from B_S will allow us to calculate the probability that $G(T, \vec{h})$ has an undesired property as if $\vec{h}$ were a sequence of uniform hash functions. It is critical to find subsets B_S of sufficiently small probability. Whether or not this is possible depends on the substructures we are interested in. However, we provide criteria that allow us to bound the size of B_S from above entirely by using graph theory. This means that details about the hash function construction need not be known to argue that random hash functions from $\mathcal{Z}$ can be used in place of uniform random hash functions for certain applications.

Outline and Suggestions. Section 11 introduces the considered class $\mathcal{Z}$ of hash functions and provides the general framework of our analysis. Because of its abstract nature, the details of the framework might be hard to understand. A simple application of the framework is provided in Section 11.4. There, we will discuss the use of hash class $\mathcal{Z}$ in static cuckoo hashing. The reader might find it helpful to study the example first to get a feeling of how the framework is applied. Another way to approach the framework is to first read the paper [ADW14]. This paper discusses one example of the framework with an application-specific focus, which might be easier to understand.

The following sections then deal with applications of the hash function construction. Because of the diverse applications, the background of each application will be provided in the respective subsection right before the analysis.

Sections 12 and 13 deal with randomness properties of $\mathcal{Z}$ on (multi-)graphs. Here, Section 12 provides some groundwork for bounding the impact of using $\mathcal{Z}$ in our applications. Section 13 discusses the use of $\mathcal{Z}$ in cuckoo hashing (with a stash), the simulation of a uniform hash function, the construction of a perfect hash function, and the behavior of $\mathcal{Z}$ on connected components of $G(S, h_1, h_2)$.

The next section (Section 14) discusses applications whose analysis builds upon hypergraphs. As an introduction, we study generalized cuckoo hashing with $d \geq 3$ hash functions when the hash table load is low. Then, we will discuss two recently described, alternative insertion algorithms for generalized cuckoo hashing. Finally, we will prove that hash class $\mathcal{Z}$ provides strong enough randomness properties for many different load balancing schemes.

In Section 15 we show how our analysis generalizes to the case that we use more involved hash functions as building blocks of hash class $\mathcal{Z}$, which lowers the total number of needed hash functions and the space consumption.

As performance is a key component of a good hash function, we evaluate the running time of hash functions from class $\mathcal{Z}$ and compare it to many other hash functions, e. g., simple tabulation hashing [PT12] and deterministic hash functions such as Murmur3 [App] in Section 16.

Summary of Results. The most important result of this part of the thesis is the general framework developed in Section 11. It states sufficient (and often "easy to check") conditions when one can use hash class $\mathcal{Z}$ in a specific application. Its usefulness is demonstrated by analyzing many different, sometimes very recent algorithms and data structures. In some cases, we are the first to prove that an explicit construction has good enough randomness properties for a specific application. In some applications, we get guarantees that match what one would get in the fully random case, e. g., for cuckoo hashing (with a stash). In other cases, the analysis does only allow to get close to what one achieves with fully random hash functions, e. g., in the construction of a perfect hash function. Sometimes, our theoretical bounds are far away from what we get in the fully random case, e. g., for generalized cuckoo hashing. The results of our experiments suggest that variants of hash class $\mathcal{Z}$ are quite fast while providing theoretical guarantees not known from other hash function constructions.

11. Basic Setup and Groundwork

Let U and R be two finite sets with $1 < |R| \leq |U|$. A *hash function with range R* is a mapping from U to R. In our applications, a hash function is applied on some key set $S \subseteq U$ with $|S| = n$. Furthermore, the range of the hash function is the set $[m] = \{0, \ldots, m-1\}$ where often $m = \Theta(n)$. In measuring space, we always assume that $\log |U|$ is a small enough term that vanishes in big-Oh notation when compared with terms depending on n. If this is not the case, one first applies a hash function to collapse the universe to some size polynomial in n [Sie04]. We say that a pair $x, y \in U, x \neq y$ *collides under a hash function g* if $g(x) = g(y)$.

The term *universal hashing* introduced by Carter and Wegman in [CW77] refers to the technique of choosing a hash function at random from a set $\mathcal{H}_m \subseteq \{h \mid h\colon U \to [m]\}$. Here, $\mathcal{H}_m$ is an indexed family $\{h_i\}_{i \in I}$. Such an indexed family is called a *hash class* or *hash family*, and selecting a hash function from $\mathcal{H}_m$ means choosing its index $i \in I$ uniformly at random. Next, we define two important notions for such hash families: *universality* and *independence*.

Definition 11.0.1 ***[CW77; CW79]***
For a constant $c \geq 1$, a hash class $\mathcal{H}$ with functions from U to $[m]$ is called *c-universal* if for an arbitrary distinct pair of keys $x, y \in U$ we have

$$\Pr_{h \in \mathcal{H}}\big(h(x) = h(y)\big) \leq c/m.$$

We remark that there exists the concept of *optimal universality*, where two distinct keys collide with probability at most $(|U| - m)/(|U| \cdot m - m)$, see [Woe99]. However, 2-universal hash classes suffice for our applications. Examples for c-universal hash families can be found in, e. g., [CW77; Die+97; Woe99]. In the following, $\mathcal{F}_m^c$ denotes an arbitrary c-universal hash family with domain U and range $[m]$.

Definition 11.0.2 ***[WC79; WC81]***
For an integer $\kappa \geq 2$, a hash class $\mathcal{H}$ with functions from U to $[m]$ is called a *κ-wise independent* hash family if for arbitrary distinct keys $x_1, \ldots, x_\kappa \in U$ and for arbitrary $j_1, \ldots, j_\kappa \in [m]$ we have

$$\Pr_{h \in \mathcal{H}}\big(h(x_1) = j_1 \wedge \ldots \wedge h(x_\kappa) = j_\kappa\big) = 1/m^\kappa.$$

In other terms, choosing a hash function uniformly at random from a κ-wise independent class of hash functions guarantees that the hash values are uniform in $[m]$ and that each key from

an arbitrary set of at most k keys from the universe is mapped independently. The classical κ-wise independent hash family construction is based on polynomials of degree $\kappa - 1$ over a finite field [WC79]. Other constructions are based on combining values that are picked from small tables filled with random elements from $[m]$ with bitwise exclusive or (*tabulation-based hashing*). To pick these values, we can, e. g., split a key x into characters $x_1, \ldots, x_c$ over some alphabet and pick as the i-th value the value in cell x_i in table i [PT12] (and, with a small twist, [PT13]). However, this scheme is only 3-wise independent. To achieve a higher degree of independence, one needs to derive additional keys. See [DW03; TZ12; KW12; PT12] for constructions using this approach. Tabulation-based constructions are often much faster in practice than polynomial-based hashing (*cf.* [TZ12]) on the cost of using slightly more memory. Throughout this thesis, $\mathcal{H}_m^\kappa$ denotes an arbitrary κ-wise independent hash family with domain U and range $[m]$.

11.1. The Hash Class

The hash class presented in this work draws ideas from many different papers. So, we first give a detailed overview of related work and key concepts.

Building upon the work on k-independent hash families and two-level hashing strategies, e. g., the FKS-scheme of Fredman *et al.* [FKS84], Dietzfelbinger and Meyer auf der Heide studied in [DM90; DM92] randomness properties of hash functions from U to $[m]$ constructed in the following way: For given $k_1, k_2, m, n \geq 2$, and δ with $0 < \delta < 1$, set $\ell = n^\delta$. Let $f\colon U \to [m]$ be chosen from a k_1-wise independent hash family, and let $g\colon U \to [\ell]$ be chosen from a k_2-wise independent hash family. Fill a table $z[1..\ell]$ with random values from $[m]$. To evaluate a key x, evaluate the function

$$h(x) = f(x) + z[g(x)] \mod m. \tag{11.1}$$

The idea is as follows: The g-function splits a key set S into buckets $S_j = \{x \in S \mid g(x) = j\}$, for $0, \ldots, \ell - 1$. To an element x from bucket S_j, the hash functions $f(x) + z[j]$ is applied. So, all elements in one row are rotated with the same random offset. Since these offsets are chosen randomly, collisions of keys that lie in different buckets happen like the hash values would be fully random, and one has only to care about the dependency of keys in a fixed bucket. Here, the focus of the analysis was the behavior of the hash class with regard to collisions of keys. The data structure needs $O(n^\delta)$ words and can be evaluated in time $O(\max\{k_1, k_2\})$.

For $m = n$, the hash class of [DM90] had many randomness properties that were only known to hold for fully random hash functions: When throwing n balls into n bins, where each candidate bin is chosen by "applying the hash function to the ball", the expected maximum bin load is $O(\log n / \log \log n)$, and the probability that a bin contains $i \geq 1$ balls decreases exponentially with i. Other explicit hash families that share this property were discovered by Pătraşcu and Thorup [PT12] and Celis *et al.* [Cel+13] only about two decades later.

In 2003, Dietzfelbinger and Woelfel [DW03] generalized the construction from [DM90] to pairs (h_1, h_2) of hash functions with $h_i\colon U \to [m]$, for $i \in \{1, 2\}$. Naïvely, one could just duplicate

the construction of [DM90]. They showed, however, that one should choose two f-functions, two z-tables, but *only one* g-function that is shared among h_1 and h_2. The key idea of the analysis was that when the g-function distributes a fixed set $T \subseteq S$ "well enough", then h_1 and h_2 can be seen as fully random hash functions on T. They used this insight to study the randomness properties of the graph $G(S, h_1, h_2)$ whose vertex set consists of two copies of $[m]$ and whose edge set is $\{(h_1(x), h_2(x)) \mid x \in S\}$. They showed that this graph behaves "almost fully randomly", in some technical sense, inside its connected components. Using this result, they proved that this explicit hash family has strong enough randomness properties that allows us to use it in, e. g., cuckoo hashing, the simulation of a uniform hash function, and the simulation of shared memory situations.

In 2006, Woelfel [Woe06a] generalized this construction from two to $d \geq 2$ hash functions using d f-functions, d z-tables and one shared g-function. He showed that it can run the *GoLeft* algorithm of Vöcking [Vöc03] for sequential balanced allocation where each ball can choose from $d \geq 2$ bins.

We modify the construction of the hash class in two different ways: First, we restrict f and g to be from very simple, two-independent and two-universal, resp., hash classes. Second, we compensate for this restriction by using $c \geq 1$ g-functions and $d \cdot c$ z-tables. This modification has two effects: it makes the analysis simpler and it seems to yield faster hash functions in practice, as we shall demonstrate in Section 16.

Definition 11.1.1
Let $c \geq 1$ and $d \geq 2$. For integers $m, \ell \geq 1$, and given $f_1, \ldots, f_d \colon U \to [m]$, $g_1, \ldots, g_c \colon U \to [\ell]$, and d two-dimensional tables $z^{(i)}[1..c, 0..\ell-1]$ with elements from $[m]$ for $i \in \{1, \ldots, d\}$, we let $\vec{h} = (h_1, \ldots, h_d) = (h_1, \ldots, h_d)\langle f_1, \ldots, f_d, g_1, \ldots, g_c, z^{(1)}, \ldots, z^{(d)}\rangle$, where

$$h_i(x) = \Big(f_i(x) + \sum_{1 \leq j \leq c} z^{(i)}[j, g_j(x)]\Big) \bmod m, \text{ for } x \in U, i \in \{1, \ldots, d\}.$$

Let $\mathcal{F}_\ell^2$ be an arbitrary two-universal class of hash functions from U to $[\ell]$, and let $\mathcal{H}_m^2$ be an arbitrary two-wise independent hash family from U to $[m]$. Then $\mathcal{Z}_{\ell,m}^{c,d}(\mathcal{F}_\ell^2, \mathcal{G}_m^2)$ is the family of all sequences $(h_1, \ldots, h_d)\langle f_1, \ldots, f_d, g_1, \ldots, g_c, z^{(1)}, \ldots, z^{(d)}\rangle$ for $f_i \in \mathcal{H}_m^2$ with $1 \leq i \leq d$ and $g_j \in \mathcal{F}_\ell^2$ with $1 \leq j \leq c$.

Obviously, this hash class can be generalized to use arbitrary κ-wise independent hash families as building blocks for the functions f_i, for $1 \leq i \leq d$, and g_j, for $1 \leq j \leq c$. However, the simpler hash functions are much easier to deal with in the proofs of this section. We defer the discussion of such a generalization to Section 15.

While this is not reflected in the notation, we consider $(h_1, \ldots, h_d)$ as a structure from which the components $g_1, \ldots, g_c$ and $f_i, z^{(i)}$, $i \in \{1, \ldots, d\}$, can be read off again. It is family $\mathcal{Z} = \mathcal{Z}_{\ell,m}^{c,d}(\mathcal{F}_\ell^2, \mathcal{G}_m^2)$ for some $c \geq 1$ and $d \geq 2$, made into a probability space by the uniform distribution, that we will study in the following. We usually assume that c and d are fixed

and that m and ℓ are known. Also, the hash families $\mathcal{F}_\ell^2$ and $\mathcal{G}_m^2$ are arbitrary hash families (providing the necessary degree of universality or independence) and will be omitted in the further discussion.

Definition 11.1.2
For $T \subseteq U$, define the random variable d_T, the "deficiency" of $\vec{h} = (h_1, \ldots, h_d)$ with respect to T, by $d_T(\vec{h}) = |T| - \max\{|g_1(T)|, \ldots, |g_c(T)|\}$. Further, define

(i) bad_T as the event that $d_T > 1$;

(ii) good_T as $\overline{\text{bad}_T}$, i. e., the event that $d_T \leq 1$;

(iii) crit_T as the event that $d_T = 1$.

Hash function sequences $(h_1, \ldots, h_d)$ in these events are called *"T-bad"*, *"T-good"*, and *"T-critical"*, respectively.

It will turn out that if a function g_j is injective on a set $T \subseteq U$, then all hash values on T are independent. The deficiency d_T of a sequence $\vec{h}$ of hash functions measures how far away the hash function sequence is from this "ideal" situation. If $\vec{h}$ is T-bad, then for each component g_j there are at least two collisions on T. If $\vec{h}$ is T-good, then there exists a g_j-component with at most one collision on T. A hash function $\vec{h}$ is T-critical if there exists a function g_j such that exactly one collision on T occurs, and for all other functions there is at least one collision. Note that the deficiency only depends on the g_j-components of a hash function. In the following, we will first fix these g_j-components when choosing a hash function. If $d(T) \leq 1$ then the yet unfixed parts of the hash function (i. e., the entries in the tables $z^{(i)}$ and the f-functions) are sufficient to guarantee strong randomness properties of the hash function on T.

Our framework will build on the randomness properties of hash class $\mathcal{Z}$ that are summarized in the next lemma. It comes in two parts. The first part makes the role of the deficiency of a hash function sequence from $\mathcal{Z}$ precise, as described above. The second part states that for a fixed set $T \subseteq S$ three parameters govern the probability of the events crit_T or bad_T to occur: The size of T, the range $[\ell]$ of the g-functions, and their number. To be precise, this probability is at most $(|T|^2/\ell)^c$, which yields two consequences. When $|T|$ is much smaller than ℓ, the factor $1/\ell^c$ will make the probability of a hash function behaving badly on a small key set vanishingly small. But when $|T|$ is larger than ℓ, the influence of the failure term of the hash class is significant. We will see later how to tackle this problem.

Lemma 11.1.3
Assume $d \geq 2$ and $c \geq 1$. For $T \subseteq U$, the following holds:

(a) Conditioned on good_T (or on crit_T), the hash values $(h_1(x), \ldots, h_d(x))$, $x \in T$, are distributed uniformly and independently in $[m]^d$.

(b) $\Pr(\text{bad}_T \cup \text{crit}_T) \leq \left(|T|^2/\ell\right)^c$.

Proof. Part (a): If $|T| \leq 2$, then $h_1, \ldots, h_d$ are fully random on T simply because $f_1, \ldots, f_d$ are drawn independently from 2-wise independent hash classes. So suppose $|T| > 2$. First, fix an arbitrary g-part of $(h_1, \ldots, h_d)$ so that crit_T occurs. (The statement follows analogously for good_T.) Let $j_0 \in \{1, \ldots, c\}$ be such that there occurs exactly one collision of keys in T using g_{j_0}. Let $x, y \in T, x \neq y$, be this pair of keys (i. e., $g_{j_0}(x) = g_{j_0}(y)$). Arbitrarily fix all values in the tables $z^{(i)}[j, k]$ with $i \in \{1, \ldots, d\}, j \neq j_0$, and $0 \leq k \leq \ell - 1$. Furthermore, fix $z^{(i)}[j_0, g_{j_0}(x)]$ with $i \in \{1, \ldots, d\}$. The hash functions $(h_1, \ldots, h_d)$ are fully random on x and y since $f_1, \ldots, f_d$ are 2-wise independent. Furthermore, the function g_{j_0} is injective on $T - \{x, y\}$ and for each $x' \in (T - \{x, y\})$ the table cell $z^{(i)}[j_0, g_{j_0}(x')]$ is yet unfixed, for $i \in \{1, \ldots, d\}$. Thus, the hash values $h_1(x), \ldots, h_d(x)$, $x \in T - \{x, y\}$, are distributed fully randomly and are independent of the hash values of x and y.

Part (b): Assume $|T| \geq 2$. (Otherwise the events crit_T or bad_T cannot occur.) Suppose crit_T (or bad_T) is true. Then for each component $g_i, 1 \leq i \leq c$, there exists a pair $x, y \in T, x \neq y$, such that $g_i(x) = g_i(y)$. Since g_i is chosen uniformly at random from a 2-universal hash class, the probability that such a pair exists is at most $\binom{|T|}{2} \cdot 2/\ell \leq |T|^2/\ell$. Since all g_i-components are chosen independently, the statement follows. □

11.2. Graph Properties and the Hash Class

We assume that the notion of a simple bipartite multigraph is known to the reader. A nice introduction to graph theory is given by Diestel [Die05]. We also consider hypergraphs (V, E) which extend the notion of a graph by allowing edges to consist of more than two vertices. For an integer $d \geq 2$, a hypergraph is called *d-uniform* if each edge contains exactly d vertices. It is called *d-partite* if V can be split into d sets $V_1, \ldots, V_d$ such that no edge contains two vertices of the same class. A hypergraph (V', E') is a *subgraph* of a hypergraph (V, E) if $V' \subseteq V$ and for each edge $e' \in E'$ there exists an edge $e \in E$ with $e' \subseteq e$. More notation for graphs and hypergraphs will be provided in Section 11.4 and Section 14, respectively.

We build graphs and hypergraphs from a set of keys $S = \{x_1, \ldots, x_n\}$ and a sequence of hash functions $\vec{h} = (h_1, \ldots, h_d), h_i : U \to [m]$, in the following way: The d-partite hypergraph $G(S, \vec{h}) = (V, E)$ has d copies of $[m]$ as vertex set and edge set $E = \{(h_1(x), \ldots, h_d(x)) \mid$

$x \in S\}$.[1] Also, the edge $(h_1(x_i), \ldots, h_d(x_i))$ is labeled "i".[2] Since keys correspond to edges, the graph $G(S, \vec{h})$ has n edges and $d \cdot m$ vertices, which is the standard notation from a "data structure" point of view, but is a non-standard notation in graph theory. For a set S and an edge-labeled graph G, we let $T(G) = \{x_i \mid x_i \in S, G \text{ contains an edge labeled } i\}$.

In the following, our main objective is to prove that with high probability certain subgraphs do not occur in $G(S, \vec{h})$. Formally, for $n, m, d \in \mathbb{N}, d \geq 2$, let $\mathcal{G}^d_{m,n}$ denote the set of all d-partite hypergraphs with vertex set $[m]$ in each class of the partition whose edges are labeled with distinct labels from $\{1, \ldots, n\}$. A set $\mathsf{A} \subseteq \mathcal{G}^d_{m,n}$ is called a *graph property*. If for a graph G we have that $G \in \mathsf{A}$, we say that G *has property* A. We shall always disregard isolated vertices.

For a key set S of size n, a sequence $\vec{h}$ of hash functions from $\mathcal{Z}$, and a graph property $\mathsf{A} \subseteq \mathcal{G}^d_{m,n}$, we define the following random variables: For each $G \in \mathsf{A}$, let I_G be the indicator random variable that indicates whether G is a subgraph of $G(S, \vec{h})$ or not. (We demand the edge labels to coincide.) Furthermore, the random variable N^{A}_S counts the number of graphs $G \in \mathsf{A}$ which are subgraphs of $G(S, \vec{h})$, i. e., $N^{\mathsf{A}}_S = \sum_{G \in \mathsf{A}} I_G$.

Let A be a graph property. Our main objective is then to estimate (from below) the probability that no subgraph of $G(S, \vec{h})$ has property A. Formally, for given $S \subseteq U$ we wish to bound (from above)

$$\Pr_{\vec{h} \in \mathcal{Z}} \left(N^{\mathsf{A}}_S > 0 \right). \tag{11.2}$$

In the analysis of randomized algorithm, bounding (11.2) is often a classical application of the *first moment method*, which says that

$$\Pr_{\vec{h} \in \mathcal{Z}} \left(N^{\mathsf{A}}_S > 0 \right) \leq \mathrm{E}_{\vec{h} \in \mathcal{Z}} \left(N^{\mathsf{A}}_S \right) = \sum_{G \in \mathsf{A}} \Pr_{\vec{h} \in \mathcal{Z}} \left(I_G = 1 \right). \tag{11.3}$$

However, we cannot apply the first moment method directly to bound (11.2), since hash functions from $\mathcal{Z}$ do not guarantee full independence on the key set, and thus the right-hand side of (11.3) is hard to calculate. However, we will prove an interesting connection to the expected number of subgraphs having property A when the hash function sequence $\vec{h}$ is fully random.

To achieve this, we will start by collecting "bad" sequences of hash functions. Intuitively, a sequence $\vec{h}$ of hash functions is *bad* with respect to a key set S and a graph property A if $G(S, \vec{h})$ has a subgraph G with $G \in \mathsf{A}$ and for the keys $T \subseteq S$ which form G the g-components of $\vec{h}$ distribute T "badly". (Recall the formal definition of "bad" from Definition 11.1.2.)

[1] In this thesis, whenever we refer to a graph or a hypergraph we mean a multi-graph or multi-hypergraph, i. e., the edge set is a multiset. We also use the words "graph" and "hypergraph" synonymously in this section. Finally, note that our edges are tuples instead of sets to avoid problems with regard to the fact that the hash functions use the same range.

[2] We assume (w.l.o.g.) that the universe U is ordered and that each set $S \subseteq U$ of n keys is represented as $S = \{x_1, \ldots, x_n\}$ with $x_1 < x_2 < \cdots < x_n$.

Definition 11.2.1
For $S \subseteq U$ and a graph property A let $B_S^{\mathsf{A}} \subseteq \mathcal{Z}$ be the event

$$\bigcup_{G \in \mathsf{A}} \left(\{I_G = 1\} \cap \mathrm{bad}_{T(G)} \right).$$

This definition is slightly different to the corresponding definition in the paper [ADW14, Definition 3], which considers one application of hash class $\mathcal{Z}$ with an application-specific focus.[3]

In addition to the probability space $\mathcal{Z}$ together with the uniform distribution, we also consider the probability space in which we use d fully random hash functions from U to $[m]$, chosen independently. From here on, we will denote probabilities of events and expectations of random variables in the former case by $\Pr$ and E; we will use $\Pr^*$ and E^* in the latter. The next lemma shows that for bounding $\Pr\left(N_S^{\mathsf{A}} > 0\right)$ we can use $\mathrm{E}^*\left(N_S^{\mathsf{A}}\right)$, i. e., the expected number of subgraphs having property A in the fully random case, and have to add the probability that the event B_S^{A} occurs. We call this additional summand the *failure term of $\mathcal{Z}$ on* A.

Lemma 11.2.2
Let $S \subseteq U$ be given. For an arbitrary graph property A we have

$$\Pr\left(N_S^{\mathsf{A}} > 0\right) \leq \Pr\left(B_S^{\mathsf{A}}\right) + \mathrm{E}^*\left(N_S^{\mathsf{A}}\right). \tag{11.4}$$

Proof. We calculate:

$$\Pr\left(N_S^{\mathsf{A}} > 0\right) \leq \Pr\left(B_S^{\mathsf{A}}\right) + \Pr\left(\left\{N_S^{\mathsf{A}} > 0\right\} \cap \overline{B_S^{\mathsf{A}}}\right).$$

We only have to focus on the second term on the right-hand side. Using the union bound, we

[3] In [ADW14] we defined $B_S^{\mathsf{A}} = \bigcup_{T \subseteq S} \left\{ \{G(T, \vec{h}) \text{ has property } A\} \cap \mathrm{bad}_T \right\}$. This works well in the case that we only consider randomness properties of the graph $G(S, h_1, h_2)$. During the preparation of this thesis, however, it turned out that in the hypergraph setting this approach was cumbersome. In that setting, "important" subgraphs of $G(S, \vec{h})$ often occurred not in terms of the graph $G(T, \vec{h})$, for some set $T \subseteq S$, but by removing some vertices from the edges of $G(T, \vec{h})$. In Definition 11.2.1, we may consider exactly such subgraphs of $G(T, \vec{h})$ by defining A properly. The edge labels of a graph are used to identify which keys of S form the graph.

continue as follows:

$$\begin{aligned}
\Pr\left(\left\{N_S^{\mathsf{A}} > 0\right\} \cap \overline{B_S^{\mathsf{A}}}\right) &\leq \sum_{G \in \mathsf{A}} \Pr\left(\{I_G = 1\} \cap \overline{B_S^{\mathsf{A}}}\right) \\
&= \sum_{G \in \mathsf{A}} \Pr\left(\{I_G = 1\} \cap \left(\bigcup_{G' \in \mathsf{A}} \left(\{I_{G'} = 0\} \cup \text{good}_{T(G')}\right)\right)\right) \\
&\leq \sum_{G \in \mathsf{A}} \Pr\left(\{I_G = 1\} \cap \text{good}_{T(G)}\right) \\
&\leq \sum_{G \in \mathsf{A}} \Pr\left(I_G = 1 \mid \text{good}_{T(G)}\right) \\
&\overset{\text{(i)}}{=} \sum_{G \in \mathsf{A}} \Pr^*\left(I_G = 1\right) = \mathrm{E}^*\left(N_S^{\mathsf{A}}\right),
\end{aligned}$$

where (i) holds by Lemma 11.1.3(b). □

This lemma encapsulates our overall strategy for bounding $\Pr(N_S^{\mathsf{A}} > 0)$. The second summand in (11.4) can be calculated assuming full randomness and is often well known from the literature in the case that the original analysis was conducted using the first moment method. The task of bounding the first summand is tackled separately in the next subsection.

11.3. Bounding the Failure Term of Hash Class $\mathcal{Z}$

As we have seen, using hash class $\mathcal{Z}$ gives an additive failure term (*cf.* (11.4)) compared to the case that we bound $\Pr^*\left(N_S^{\mathsf{A}} > 0\right)$ by the first moment method in the fully random case. Calculating $\Pr\left(B_S^{\mathsf{A}}\right)$ looks difficult since we have to calculate the probability that there exists a subgraph G of $G(S, \vec{h})$ that has property A and where $\vec{h}$ is $T(G)$-bad. Since we know the probability that $\vec{h}$ is $T(G)$-bad from Lemma 11.1.3(b), we could tackle this task by calculating the probability that there exists such a subgraph G under the condition that $\vec{h}$ is $T(G)$-bad, but then we cannot assume full randomness of $\vec{h}$ on $T(G)$ to obtain a bound that a certain subgraph is realized by the hash values. Since this is hard, we will take another approach. We will find suitable events that contain B_S^{A} and where $\vec{h}$ is guaranteed to behave well on the key set in question.

Observe the following relationship that is immediate from Definition 11.2.1.

Lemma 11.3.1
Let $S \subseteq U, |S| = n$, and let $\mathsf{A} \subseteq \mathsf{A}' \subseteq \mathcal{G}_{m,n}^d$. Then $\Pr\left(B_S^{\mathsf{A}}\right) \leq \Pr\left(B_S^{\mathsf{A}'}\right)$. □

We will now introduce two concepts that will allow us to bound the failure probability of $\mathcal{Z}$ for "suitable" graph properties A.

Definition 11.3.2 ***Peelability***
A graph property A is called **peelable** if for all $G = (V, E) \in \mathsf{A}, |E| \geq 1$, there exists an edge $e \in E$ such that $(V, E - \{e\}) \in \mathsf{A}$.

A peelable graph property for bipartite graphs, i. e., in the case $d = 2$, is the set of all connected bipartite graphs (disregarding isolated vertices), because removing an edge that lies on a cycle or an edge incident to a vertex of degree 1 does not destroy connectivity.

Peelable graph properties will help us in the following sense: Assume that B_S^{A} occurs, i. e., for the chosen $\vec{h} \in \mathcal{Z}$ there exists some graph $G \in \mathsf{A}$ that is a subgraph of $G(S, \vec{h})$ and $\vec{h}$ is $T(G)$-bad. Let $T = T(G)$. In terms of the "deficiency" d_T of $\vec{h}$ (*cf.* Definition 11.1.2) it holds that $d_T(\vec{h}) > 1$. If A is peelable, we can iteratively remove edges from G such that the resulting graphs still have property A. Let G' be a graph that results from G by removing a single edge. Then $d_{T(G)} - d_{T(G')} \in \{0, 1\}$. Eventually, because $d_\emptyset = 0$, we will obtain a subgraph $G' \in \mathsf{A}$ of G such that $\vec{h}$ is $T(G')$-critical. In this case, we can again make use of Lemma 11.1.3(b) and bound the probability that G' is realized by the hash function sequence by assuming that the hash values are fully random.

However, peelability does not suffice to obtain low enough bounds for failure terms $\Pr\left(B_S^{\mathsf{A}}\right)$; we need the following auxiliary concept, whose idea will become clear in the proof of the next lemma.

Definition 11.3.3 ***Reducibility***
Let $c \in \mathbb{N}$, and let A and B be graph properties. A is called B-$2c$-**reducible** if for all graphs $(V, E) \in \mathsf{A}$ and sets $E^* \subseteq E$ the following holds: if $|E^*| \leq 2c$ then there exists an edge set E' with $E^* \subseteq E' \subseteq E$ such that $(V, E') \in \mathsf{B}$.

If a graph property A is B-$2c$-reducible, we say that A *reduces to* B. The parameter c shows the connection to hash class $\mathcal{Z}$: it is the same parameter as the number of g_j-functions in hash class $\mathcal{Z}$.

To shorten notation, we let

$$\mu_t^{\mathsf{A}} := \sum_{G \in \mathsf{A}, |E(G)| = t} \Pr{}^*\left(I_G = 1\right)$$

be the expected number of subgraphs with exactly t edges having property A in the fully random case. The following lemma is the central result of this section and encapsulates our overall strategy to bound the additive failure term introduced by using hash class $\mathcal{Z}$ instead of fully random hash functions.

Lemma 11.3.4
Let $c \geq 1$, $S \subseteq U$ with $|S| = n$, and let A, B, and C be graph properties such that $\mathsf{A} \subseteq \mathsf{B}$, B is a peelable graph property, and B reduces to C. Then

$$\Pr\left(B_S^{\mathsf{A}}\right) \leq \Pr\left(B_S^{\mathsf{B}}\right) \leq \ell^{-c} \cdot \sum_{t=2}^{n} t^{2c} \cdot \mu_t^{\mathsf{C}}.$$

Proof. By Lemma 11.3.1 we have $\Pr\left(B_S^{\mathsf{A}}\right) \leq \Pr(B_S^{\mathsf{B}}) = \Pr\left(\bigcup_{G\in\mathsf{B}}(\{I_G = 1\} \cap \mathrm{bad}_{T(G)})\right)$. Assume that $\vec{h}$ is such that B_S^{B} occurs. Then there exists a subgraph G of $G(S, \vec{h})$ such that $G \in \mathsf{B}$ and $d_{T(G)}(\vec{h}) > 1$. Fix such a graph.

Since B is peelable, we iteratively remove edges from G until we obtain a graph $G' = (V, E')$ such that $G' \in \mathsf{B}$ and $\mathrm{crit}_{T(G')}$ occurs. The latter is guaranteed, for $d_\emptyset(\vec{h}) = 0$ and for two graphs G and G', where G' results from G by removing a single edge, it holds that $d_{T(G)}(\vec{h}) - d_{T(G')}(\vec{h}) \in \{0, 1\}$. Since $\mathrm{crit}_{T(G')}$ happens, for each g_i-component of $\vec{h}, 1 \leq i \leq c$, there is at least one collision on $T(G')$. Furthermore, there exists one component $g_{j_0}, j_0 \in \{1, \ldots, c\}$, such that exactly one collision on $T(G')$ occurs. For each $g_i, i \in \{1, \ldots, c\}$, let $\{e_i, e_i'\}, e_i \neq e_i'$, be two edges of G' such that the keys x_i, y_i which correspond to e_i and e_i' collide under g_i. Let $E^* = \bigcup_{1\leq i\leq c}\{e_i, e_i'\}$.

By construction $|E^*| \leq 2c$. Since B reduces to C, there exists some set E'' with $E^* \subseteq E'' \subseteq E'$ such that $G'' = (V, E'') \in \mathsf{C}$. By construction of E^*, each g_i-component has at least one collision on $T(G'')$. Moreover, g_{j_0} has exactly one collision on $T(G'')$. Thus, $\vec{h}$ is $T(G'')$-critical.

We calculate:

$$\begin{aligned}
\Pr\left(B_S^{\mathsf{A}}\right) \leq \Pr\left(B_S^{\mathsf{B}}\right) &= \Pr\left(\bigcup_{G\in\mathsf{B}}\left(\{I_G = 1\} \cap \mathrm{bad}_{T(G)}\right)\right) \overset{\text{(i)}}{\leq} \Pr\left(\bigcup_{G'\in\mathsf{B}}\left(\{I_{G'} = 1\} \cap \mathrm{crit}_{T(G')}\right)\right) \\
&\overset{\text{(ii)}}{\leq} \Pr\left(\bigcup_{G''\in\mathsf{C}}\left(\{I_{G''} = 1\} \cap \mathrm{crit}_{T(G'')}\right)\right) \leq \sum_{G''\in\mathsf{C}} \Pr\left(\{I_{G''} = 1\} \cap \mathrm{crit}_{T(G'')}\right) \\
&\leq \sum_{G''\in\mathsf{C}} \Pr\left(I_{G''} = 1 \mid \mathrm{crit}_{T(G'')}\right) \cdot \Pr\left(\mathrm{crit}_{T(G'')}\right) \\
&\overset{\text{(iii)}}{\leq} \ell^{-c} \cdot \sum_{G''\in\mathsf{C}} \Pr^*\left(\{I_{G''} = 1\}\right) \cdot |T(G'')|^{2c} \\
&= \ell^{-c} \cdot \sum_{t=2}^{n} \left(t^{2c} \sum_{\substack{G''\in\mathsf{C} \\ |E(G'')|=t}} \Pr^*\left(\{I_{G''} = 1\}\right) \right) = \ell^{-c} \cdot \sum_{t=2}^{n} t^{2c} \cdot \mu_t^{\mathsf{C}},
\end{aligned}$$

where (i) holds for B is peelable, (ii) is due to reducibility, and (iii) follows by Lemma 11.1.3. □

We summarize the results of Lemma 11.2.2 and Lemma 11.3.4 in the following lemma.

Lemma 11.3.5
Let $c \geq 1, m \geq 1$, $S \subseteq U$ with $|S| = n$, and let A, B, and C be graph properties such that $\mathsf{A} \subseteq \mathsf{B}$, B is a peelable graph property, and B reduces to C. Assume that there are constants α, β such that

$$\mathrm{E}^*\left(N_S^{\mathsf{A}}\right) := \sum_{t=1}^{n} \mu_t^{\mathsf{A}} = O\left(n^{-\alpha}\right), \tag{11.5}$$

and

$$\sum_{t=2}^{n} t^{2c} \mu_t^{\mathsf{C}} = O\left(n^{\beta}\right). \tag{11.6}$$

Then setting $\ell = n^{(\alpha+\beta)/c}$ and choosing $\vec{h}$ at random from $\mathcal{Z}_{\ell,m}^{c,d}$ yields

$$\Pr\left(N_S^{\mathsf{A}} > 0\right) = O\left(n^{-\alpha}\right).$$

Proof. Follows immediately by plugging the failure probability bound from Lemma 11.3.4 into Lemma 11.2.2. □

Remark 11.3.6. In the statement of Lemma 11.2.2 and Lemma 11.3.5 graph properties B and C can be the same graph properties, since every graph property reduces to itself.

Lemma 11.3.5 shows the power of our framework. The conditions of this lemma can be checked without looking at the details of the hash functions, only by finding suitable graph properties that have a low enough expected number of subgraphs in the fully random case. Let us compare properties (11.5) and (11.6). Property (11.5) is the standard first moment method approach. So, it can often be checked from the literature whether a particular application seems suitable for an analysis with our framework or not. Property (11.6) seems very close to a first moment method approach, but there is one important difference to (11.5). The additional factor t^{2c}, coming from the randomness properties of the hash class, means that to obtain low enough bounds for (11.6), the average number of graphs with property C must decrease rapidly, e. g., exponentially, fast in t. This will be the case for almost all graph properties considered in this thesis.

In the analysis, we will use Lemma 11.2.2 and Lemma 11.3.4 instead of Lemma 11.3.5. Often, one auxiliary graph property suffices for many different applications and we think it is cleaner to first bound the failure term of $\mathcal{Z}$ on this graph property using Lemma 11.3.4; then we only have to care about the fully random case and apply Lemma 11.2.2 at the end.

At the end of this section we discuss one generalization of the notion of "reducibility".

Definition 11.3.7 ***Generalized Reducibility***
Let $c \in \mathbb{N}$, and let A and B be graph properties. A is called **weak** B-$2c$-**reducible** if for all graphs $(V, E) \in \mathsf{A}$ and sets $E^* \subseteq E$ the following holds: if $|E^*| \leq 2c$ then there exists a subgraph $(V, E') \in \mathsf{B}$ of (V, E) such that for each edge $e^* \in E^*$ there exists an edge $e' \in E'$ with $e' \subseteq e^*$ having the same label as e^*.

In difference to Definition 11.3.3, we can remove vertices from the edges in edge set E^*. This notion will be used in applications of our framework to hypergraphs. A proof analogous to the proof of Lemma 11.3.4 shows that the statement of Lemma 11.3.4 is also true if B is weak C-$2c$-reducible.

This constitutes the theoretical basis of the second part of this thesis.

11.4. Step by Step Example: Analyzing Static Cuckoo Hashing

Graph Notation. We start by fixing graph-related notation: We call an edge that is incident to a vertex of degree 1 a *leaf edge*. We call an edge a *cycle edge* if removing it does not disconnect any two nodes. A connected graph is called *acyclic* if it does not contain cycles. It is called *unicyclic* if it contains exactly one cycle. The 2-core of a graph G is the maximal subgraph of G in which each vertex has minimum degree 2. For a (hyper-)graph $G = (V, E)$, a function $f\colon E \to V$ is a 1-*orientation* of G if f is injective. (For each edge we pick one vertex such that each vertex is picked at most once.)

Background. Cuckoo hashing [PR04] is a dictionary algorithm that stores a (dynamically changing) set $S \subseteq U$ of size n in two hash tables, T_1 and T_2, each of size $m \geq (1+\varepsilon)n$ for some $\varepsilon > 0$. It employs two hash functions h_1 and h_2 with $h_1, h_2\colon U \to [m]$. A key x can be stored either in $T_1[h_1(x)]$ or in $T_2[h_2(x)]$, and all keys are stored in distinct table cells. Thus, to find or remove a key it suffices to check these two possible locations.

Cuckoo hashing deals with collisions by moving keys between the two tables. A new key x is always inserted into $T_1[h_1(x)]$. If this cell is occupied by some key x', then that key is evicted from the hash table and becomes "nestless" before x is inserted. Whenever a key x' has been evicted from $T_i[h_i(x')]$, $i \in \{1, 2\}$, it is afterwards reinserted in the other table, $T_{3-i}[h_{3-i}(x')]$, after possibly evicting the element stored there. This process continues until an empty cell is found, i. e., no eviction is necessary. The procedure may cycle forever, so if it does not terminate after a given number, $\mathrm{MaxLoop} = \Theta(\log n)$, of steps, new hash functions h_1 and h_2 are chosen, and the data structure is rebuilt from scratch.

In this section, we deal with the static setting whether or not a key set S of size n can be stored in the two tables of size $(1+\varepsilon)n$ each, for some $\varepsilon > 0$, using a pair of hash functions (h_1, h_2) according to the cuckoo hashing rules. To this end, we look at the bipartite graph $G(S, h_1, h_2)$ built from S and (h_1, h_2). Recall that the vertices of G are two copies of $[m]$ and that each key $x_i \in S$ gives rise to an edge $(h_1(x), h_2(x))$ labeled i. If (h_1, h_2) allow storing S according to the cuckoo hashing rules, i. e., independent of the actual insertion algorithm, we call (h_1, h_2) *suitable* for S.

This section is meant as an introductory example. Already Pagh and Rodler showed in [PR04] that using a $\Theta(\log n)$-wise independent hash class suffices to run cuckoo hashing. Dietzfelbinger and Woelfel showed in [DW03] that this is also possible using a specific variant of hash class $\mathcal{Z}$. Since standard cuckoo hashing is a special case of cuckoo hashing with a stash, the results here can also be proven using the techniques presented in the author's diploma thesis [Aum10] and the paper [ADW14]. However, the proofs here are notably simpler than the proofs needed for the analysis of cuckoo hashing with a stash, as we shall see in Section 12 and Section 13.

Result. We will prove the following theorem:

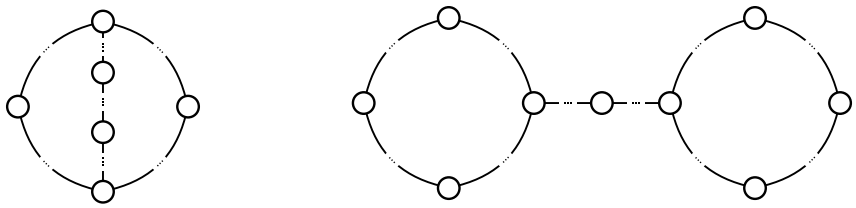

Figure 11.1.: The minimal obstruction graphs for cuckoo hashing.

Theorem 11.4.1
Let $\varepsilon > 0$ and $0 < \delta < 1$ be given. Assume $c \geq 2/\delta$. For $n \geq 1$ consider $m \geq (1+\varepsilon)n$ and $\ell = n^{\delta}$. Let $S \subseteq U$ with $|S| = n$. Then for (h_1, h_2) chosen at random from $\mathcal{Z} = \mathcal{Z}_{\ell,m}^{c,2}$ the following holds:

$$\Pr\left((h_1, h_2) \text{ is not suitable for } S\right) = O(1/n).$$

In the following, all statements of lemmas and claims use the parameter settings of Theorem 11.4.1.

By the cuckoo hashing rules, the pair (h_1, h_2) of hash functions is suitable if and only if $G(S, h_1, h_2)$ has a 1-*orientation*, i. e., if every edge can be directed in such a way that each vertex has in-degree at most 1. It is not hard to see that (h_1, h_2) is suitable for S if and only if every connected component of $G(S, h_1, h_2)$ has at most one cycle [DM03]. So, if (h_1, h_2) is not suitable, $G(S, h_1, h_2)$ has a connected component with more than one cycle. This motivates to consider the following graph property.

Definition 11.4.2
Let MOG ("minimal obstruction graphs") be the set of all labeled graphs from $G_{m,n}^2$ (disregarding isolated vertices) that form either a cycle with a chord or two cycles connected by a path of length $t \geq 0$.

These two types of graphs form minimal connected graphs with more than one cycle, see Figure 11.1. So, if (h_1, h_2) is not suitable for S, then $G(S, h_1, h_2)$ contains a subgraph with property MOG. We summarize:

$$\Pr\left((h_1, h_2) \text{ is not suitable for } S\right) = \Pr\left(N_S^{\mathsf{MOG}} > 0\right). \tag{11.7}$$

According to Lemma 11.2.2, we can bound the probability on the right-hand side of (11.7) as follows

$$\Pr\left(N_S^{\mathsf{MOG}} > 0\right) \leq \Pr\left(B_S^{\mathsf{MOG}}\right) + \mathrm{E}^*\left(N_S^{\mathsf{MOG}}\right). \tag{11.8}$$

We first study the expected number of minimal obstruction graphs in the fully random case.

Bounding $\mathrm{E}^*\big(N_S^{\mathsf{MOG}}\big)$. The expected number of minimal obstruction graphs in the fully random case is well known from other work, see, e. g., [PR04; DM03]. For completeness, we give a full proof, which can also be found in [Aum10].

Lemma 11.4.3

$$\mathrm{E}^*\left(N_S^{\mathsf{MOG}}\right) = O(1/m).$$

Proof. We start by counting unlabeled graphs with exactly t edges that form a minimal obstruction graph. Every minimal obstruction graphs consists of a simple path of exactly $t-2$ edges and two further edges which connect the endpoints of this path with vertices on the path. Since a minimal obstruction graph with t edges has exactly $t-1$ vertices, there are no more than $(t-1)^2$ unlabeled minimal obstruction graphs having exactly t edges. Fix an unlabeled minimal obstruction graph G. First, there are two ways to split the vertices of G into the two parts of the bipartition. When this is fixed, there are no more than m^{t-1} ways to label the vertices with labels from $[m]$, and there are no more than n^{t+1} ways to label the edges with labels from $\{1,\dots,n\}$. Fix such a fully labeled graph G'.

Now draw t labeled edges[4] at random from $[m]^2$. The probability that these edges realize G' is exactly $1/m^{2t}$. We calculate:

$$\mathrm{E}^*\left(N_S^{\mathsf{MOG}}\right) \leq \sum_{t=3}^{n} \frac{2n^t \cdot m^{t-1} \cdot (t-1)^2}{m^{2t}} \leq \frac{2}{m}\sum_{t=3}^{n}\frac{t^2 n^t}{m^t} = \frac{2}{m}\sum_{t=3}^{n}\frac{t^2}{(1+\varepsilon)^t} = O\Big(\frac{1}{m}\Big),$$

where the last step follows from the convergence of the series $\sum_{t=0}^{\infty} t^2/q^t$ for every $q > 1$. □

We summarize:

$$\Pr\left(N_S^{\mathsf{MOG}} > 0\right) \leq \Pr\left(B_S^{\mathsf{MOG}}\right) + O\left(\frac{1}{m}\right). \tag{11.9}$$

It remains to bound the failure term $\Pr\big(B_S^{\mathsf{MOG}}\big)$.

Bounding $\Pr\big(B_S^{\mathsf{MOG}}\big)$. In the light of Definition 11.3.2, we first note that MOG is not peelable. So, we first find a peelable graph property that contains MOG. Since paths are peelable, and a minimal obstruction graph is "almost path-like" (*cf.* proof of Lemma 11.4.3), we relax the notion of a minimal obstruction graph in the following way.

[4] The labels of these edges are equivalent to the edge labels of G'.

Definition 11.4.4
Let RMOG ("relaxed minimal obstruction graphs") consist of all graphs in $\mathcal{G}^2_{m,n}$ that form either (i) a minimal obstruction graph, (ii) a simple path, or (iii) a simple path and exactly one edge which connects an endpoint of the path with a vertex on the path, disregarding isolated vertices.

By the definition, we obviously have that MOG $\subseteq$ RMOG.

Lemma 11.4.5
RMOG is peelable.

Proof. Let $G \in$ RMOG. We may assume that G has at least two edges. We distinguish three cases:
Case 1: G is a minimal obstruction graph. Let G' be the graph that results from G when we remove an arbitrary cycle edge incident to a vertex of degree 3 in G. Then G' has property (iii) of Definition 11.4.4.
Case 2: G has property (iii) of Definition 11.4.4. Then, let G' be the graph that results from G when we remove an edge in the following way: If G contains a vertex of degree 3 then remove an arbitrary cycle edge incident to this vertex of degree 3, otherwise remove an arbitrary cycle edge. Then G' is a path and thus has property (ii) of Definition 11.4.4.
Case 3: G is a simple path. Let G' be the graph that results from G when we remove an endpoint of G with the incident edge. G' is a path and has property (ii) of Definition 11.4.4. □

Standard cuckoo hashing is an example where we do not need every component of our framework, because there are "few enough" graphs having property RMOG to obtain low enough failure probabilities.

Lemma 11.4.6

$$\Pr\left(B_S^{\mathsf{MOG}}\right) = O\left(\frac{n}{\ell^c}\right).$$

Proof. We aim to apply Lemma 11.3.4, where MOG takes the role of A and RMOG takes the role of B and C (*cf.* Remark 11.3.6), respectively, in the statement of that lemma.

Claim 11.4.7
For $t \geq 2$, we have

$$\mu_t^{\mathsf{RMOG}} \leq \frac{6mt^2}{(1+\varepsilon)^t}.$$

Proof. We first count labeled graphs with exactly t edges having property RMOG. From the proof of Lemma 11.4.3, we know that there are fewer than $2 \cdot t^2 \cdot n^t \cdot m^{t-1}$ labeled graphs which form minimal obstruction graphs ((i) of Def. 11.4.4). Similarly, there are not more than $2 \cdot n^t \cdot m^{t+1}$ labeled paths ((ii) of Def. 11.4.4), and not more than $2 \cdot t \cdot n^t \cdot m^t$ graphs having property (iii) of Def. 11.4.4. Fix a labeled graph G with property RMOG having exactly t edges. Draw t labeled edges at random from $[m]^2$. The probability that these t edges realize G is exactly $1/m^{2t}$. We calculate:

$$\mu_t^{\mathsf{RMOG}} \leq \frac{6t^2 n^t m^{t+1}}{m^{2t}} = \frac{6mt^2}{(1+\varepsilon)^t}.$$

□

Using Lemma 11.3.4, we proceed as follows:

$$\Pr\left(B_S^{\mathsf{MOG}}\right) \leq \ell^{-c} \sum_{t=2}^{n} t^{2c} \cdot \mu_t^{\mathsf{RMOG}} \leq \ell^{-c} \sum_{t=2}^{n} \frac{6mt^{2(c+1)}}{(1+\varepsilon)^t} = O\left(\frac{n}{\ell^c}\right).$$

□

Putting Everything Together. Plugging the results of Lemma 11.4.3 and Lemma 11.4.6 into (11.8) gives:

$$\Pr\left(N_S^{\mathsf{MOG}} > 0\right) \leq \Pr\left(B_S^{\mathsf{MOG}}\right) + \mathrm{E}^*\left(N_S^{\mathsf{MOG}}\right) = O\left(\frac{n}{\ell^c}\right) + O\left(\frac{1}{m}\right).$$

Using that $m = (1+\varepsilon)n$ and setting $\ell = n^\delta$ and $c \geq 2/\delta$ yields Theorem 11.4.1.

Remarks and Discussion. As mentioned in the background remarks at the beginning of this section, the actual insertion algorithm only tries to insert a new key for $\Theta(\log n)$ steps, and declares the insertion a failure if it did not finish in that many steps. This means that an insertion could fail although $G(S, h_1, h_2)$ did not contain a component with more than one cycle. To analyze this situation, one also has to consider the existence of paths of logarithmic length in $G(S, h_1, h_2)$. The analysis is a generalization of what we did here. In particular, long paths are included in the graph property RMOG, so we can use Lemma 11.4.6 to bound the failure term of $\mathcal{Z}$ on long paths. Calculations very similar to the ones in the proof of Claim 11.4.7 show that the expected number of paths having at least a certain logarithmic length in the fully random case can be made as small as $O(n^{-\alpha})$, for $\alpha \geq 1$.

This example also gives detailed insight into the situation in which our framework can be applied. The graph property under consideration (MOG) had the property that the expected number of subgraphs with this property is polynomially small in n. The peeling process—however—yields

graphs which are much more likely to occur, e. g., paths of a given length. The key in our analysis is finding suitable graph properties of "small enough" size. (That is the reason why the concept of "reducibility" from Definition 11.3.3 is needed in other applications: It makes the number of graphs that must be considered smaller.) The g-components of the hash functions from $\mathcal{Z}$ provide a boost of ℓ^{-c}, which is then used to make the overall failure term again polynomially small in n.

The reader might find it instructive to apply Lemma 11.3.5 directly. Then, graph property MOG plays the role of graph property A in that lemma; graph property RMOG plays the role of B *and* C.

12. Randomness Properties of $\mathcal{Z}$ on Leafless Graphs

In this section we study the additive failure term of hash functions from $\mathcal{Z}$ on a graph property that will be a key ingredient in future applications. First, we present a basic counting argument for unlabeled graphs. (The graphs we shall consider here are much more complicated than the minimal obstruction graphs of the previous section.) Subsequently, we study the failure term of $\mathcal{Z}$ on the class of graphs which contain no leaf edges.

We note that the counting argument below already appeared in [Aum10]. We give the proof for completeness. It is also present in [ADW14].

12.1. A Counting Argument

The cyclomatic number $\gamma(G)$ is the dimension of the *cycle space* of a graph G. It is equal to the smallest number of edges we have to remove from G such that the remaining graph is a forest (an acyclic, possibly disconnected graph) [Die05]. Also, let $\zeta(G)$ denote the number of connected components of G (ignoring isolated vertices).

Definition 12.1.1
Let $N(t, \ell, \gamma, \zeta)$ be the number of unlabeled (multi-)graphs with ζ connected components and cyclomatic number γ that have $t - \ell$ inner edges and ℓ leaf edges.

The following lemma generalizes a result of Dietzfelbinger and Woelfel [DW03] with regard to the number of unlabeled *connected* graphs with a given cyclomatic number and a given number of leaf edges.

Lemma 12.1.2
$N(t, \ell, \gamma, \zeta) = t^{O(\ell+\gamma+\zeta)}$.

Proof. We will proceed in three steps:

1. $N(t, \ell, 0, 1) = t^{O(\ell)}$
2. $N(t, \ell, \gamma, 1) = t^{O(\ell+\gamma)}$
3. $N(t, \ell, \gamma, \zeta) = t^{O(\ell+\gamma+\zeta)}$

Part 1. We first consider the case $\gamma = 0$, thus we consider trees. For $\ell = 2$, the tree is a path of length t. We refer to this tree with G_2 (the index refers to the number of leaf edges in the graph). For $\ell = 3, \ldots, \ell$, G_i is constructed using G_{i-1} by taking a new path of length $t_i \geq 1$ such that $t_2 + \cdots + t_i \leq t - (\ell - i)$ and identify one endpoint of the path with a vertex in G_{i-1}. The length of the last path is uniquely determined by $t_\ell = t - t_2 - \cdots - t_{\ell-1}$. There are fewer than $t^{\ell-2}$ choices for picking these lengths. Furthermore, there are at most $t^{\ell-2}$ choices for the inner vertex a new path is connected to. It follows

$$N(t, \ell, 0, 1) = t^{O(\ell)}.$$

Part 2. Assume cyclomatic number $\gamma \geq 1$ and $\ell \geq 0$ leaf edges. In this case, removing γ cycle edges yields a tree. There are not more than $t^{2\gamma}$ choices for the endpoints of these edges and the remaining tree has at most $\ell + 2\gamma$ leaf edges. Thus,

$$N(t, \ell, \gamma, 1) = t^{O(\gamma)} \cdot N(t - \gamma, \ell + 2\gamma, 0, 1) = t^{O(\gamma)} \cdot t^{O(\ell+\gamma)} = t^{O(\ell+\gamma)}.$$

Part 3. Each graph G with cyclomatic number γ, ζ connected components, $t - \ell$ non-leaf edges, and ℓ leaf edges can be obtained from some connected graph G' with cyclomatic number γ, $t - \ell + \zeta - 1$ non-leaf edges, and ℓ leaf edges by removing $\zeta - 1$ non-leaf, non-cycle edges. There are no more than $(t - \ell + \zeta - 1)^{\zeta-1}$ ways for choosing the edges to be removed. This implies:

$$\begin{aligned} N(t, \ell, \gamma, \zeta) &\leq N(t + \zeta - 1, \ell, \gamma, 1) \cdot (t - \ell + \zeta - 1)^{\zeta-1} \\ &\leq (t + \zeta)^{O(\ell+\gamma)} \cdot (t + \zeta)^{\zeta} = (t + \zeta)^{O(\ell+\gamma+\zeta)} = t^{O(\ell+\gamma+\zeta)}. \end{aligned}$$

□

12.2. The Leafless Part of $G(S, h_1, h_2)$

We let $\mathsf{LL} \subseteq \mathcal{G}^2_{m,n}$ consist of all bipartite graphs that contain no leaf edge. It will turn out that for all our applications LL will be a suitable "intermediate" graph property, i. e., for the graph property A interesting for the application it will hold $\mathsf{A} \subseteq \mathsf{LL}$, which will allow us to apply Lemma 11.3.1. (For example, graph property LL could have been used instead of graph property RMOG in the example of the previous section.) Hence our goal in this section is to show that there exists a constant $\alpha > 0$, which depends on the parameters ℓ and c of the hash class $\mathcal{Z}^{c,2}_{\ell,m}$, such that

$$\Pr_{(h_1,h_2)\in\mathcal{Z}}\left(B^{\mathsf{LL}}_S\right) = O\left(n^{-\alpha}\right).$$

Luckily, bounding $\Pr\left(B^{\mathsf{LL}}_S\right)$ is an example par excellence for applying Lemma 11.3.4. To use this lemma we have to find a suitable peelable graph property (note that LL is not peelable) and a

suitable graph property to which this graph property reduces.

We let LC consist of all graphs G from $\mathcal{G}^2_{m,n}$ that contain at most one connected component that has leaves, disregarding isolated vertices. If such a component exists, we call it the *leaf component* of G.

Lemma 12.2.1
LC is peelable.

Proof. Suppose $G \in$ LC has at least one edge. If G has no leaf component then all edges are cycle edges, and removing an arbitrary cycle edge creates a leaf component. So, the resulting graph has property LC. If G has a leaf component C, remove a leaf edge. This makes the component smaller, but maintains property LC. So, the resulting graph has again property LC. □

We will also need the following auxiliary graph property:

Definition 12.2.2
Let $K \in \mathbb{N}$. Let $\mathsf{LCY}^{(K)} \subseteq \mathcal{G}^2_{m,n}$ be the set of all bipartite graphs $G = (V, E)$ with the following properties (disregarding isolated vertices):

1. at most one connected component of G contains leaves (i. e., $\mathsf{LCY}^{(K)} \subseteq$ LC);
2. the number $\zeta(G)$ of connected components is bounded by K;
3. if present, the leaf component of G contains at most K leaf and cycle edges;
4. the cyclomatic number $\gamma(G)$ is bounded by K.

Lemma 12.2.3
Let $c \geq 1$. LC is $\mathsf{LCY}^{(4c)}$-$2c$-reducible.

Proof. Choose an arbitrary graph $G = (V, E) \in$ LC and an arbitrary edge set $E^* \subseteq E$ with $|E^*| \leq 2c$. We say that an edge that belongs to E^* is *marked*. G satisfies Property 1 of graphs from $\mathsf{LCY}^{(4c)}$. We process G in three stages:

Stage 1: Remove all components of G without marked edges. Afterwards at most $2c$ components are left, and G satisfies Property 2.

Stage 2: If G has a leaf component C, repeatedly remove unmarked leaf and cycle edges from C, while C has such edges. The remaining leaf and cycle edges in C are marked, and thus their number is at most $2c$; Property 3 is satisfied.

Stage 3: If there is a leaf component C with z marked edges (where $z \leq 2c$), then $\gamma(C) \leq z-1$. Now consider a leafless component C' with cyclomatic number z. We need the following graph theoretic claim:

Claim 12.2.4
Every leafless connected graph with i marked edges has a leafless connected subgraph with cyclomatic number $\leq i+1$ that contains all marked edges.

Proof. Let $G = (V, E)$ be a leafless connected graph. If $\gamma(G) \leq i + 1$, there is nothing to prove. So suppose $\gamma(G) \geq i + 2$. Choose an arbitrary spanning tree (V, E_0) of G.

There are two types of edges in G: *bridge edges* and *cycle edges*. A bridge edge is an edge whose deletion disconnects the graph, cycle edges are those whose deletion does not disconnect the graph.

Clearly, all bridge edges are in E_0. Let $E_{\text{mb}} \subseteq E_0$ denote the set of marked bridge edges. Removing the edges of E_{mb} from G splits V into $|E_{\text{mb}}|+1$ connected components $V_1, \ldots, V_{|E_{\text{mb}}|+1}$; removing the edges of E_{mb} from the spanning tree (V, E_0) will give exactly the same components. For each *cyclic* component V_j we choose one edge $e_j \notin E_0$ that connects two nodes in V_j. The set of these $|E_{\text{mb}}| + 1$ edges is called E_1. Now each marked bridge edge lies on a path connecting two cycles in $(V, E_0 \cup E_1)$.

Recall from graph theory [Die05] the notion of a fundamental cycle: Clearly, each edge $e \in E - E_0$ closes a unique cycle with E_0. The cycles thus obtained are called the fundamental cycles of G w. r. t. the spanning tree (V, E_0). Each cycle in G can be obtained as an XOR-combination of fundamental cycles. (This is just another formulation of the standard fact that the fundamental cycles form a basis of the "cycle space" of G, see [Die05].) From this it is immediate that every cycle edge of G lies on some fundamental cycle. Now we associate an edge $e' \notin E_0$ with each marked cycle edge $e \in E_{\text{mc}}$. Given e, let $e' \notin E_0$ be such that e is on the fundamental cycle of e'. Let E_2 be the set of all edges e' chosen in this way. Clearly, each $e \in E_{\text{mc}}$ is a cycle edge in $(V, E_0 \cup E_2)$.

Now let $G' = (V, E_0 \cup E_1 \cup E_2)$. Note that $|E_1 \cup E_2| \leq (|E_{\text{mb}}| + 1) + |E_{\text{mc}}| \leq i + 1$ and thus $\gamma(G') \leq i + 1$. In G', each marked edge is on a cycle or on a path that connects two cycles. If we iteratively remove leaf edges from G' until no leaf is left, none of the marked edges will be affected. In this way we obtain the desired leafless subgraph G^* with $\gamma(G^*) = \gamma(G') \leq i + 1$. □

This claim gives us a leafless subgraph C'' of C' with $\gamma(C'') \leq z + 1$ that contains all marked edges of C'. We remove from G all vertices and edges of C' that are not in C''. Doing this for all leafless components yields the final graph G. Summing contributions to the cyclomatic number of G over all (at most $2c$) connected components, we see that $\gamma(G) \leq 4c$; Property 4 is satisfied. □

We now bound the additive failure term $\Pr\left(B_S^{\mathsf{LL}}\right)$.

Lemma 12.2.5
Let $S \subseteq U$ with $|S| = n$, $\varepsilon > 0$, $c \geq 1$, and let $\ell \geq 1$. Consider $m \geq (1+\varepsilon)n$. If (h_1, h_2) are chosen at random from $\mathcal{Z}^{c,2}_{\ell,m}$, then

$$\Pr\left(B^{\mathsf{LL}}_S\right) \leq \Pr\left(B^{\mathsf{LC}}_S\right) = O\left(n/\ell^c\right).$$

Proof. According to Lemma 11.3.4 and Lemma 12.2.3 it holds that

$$\Pr\left(B^{\mathsf{LL}}_S\right) \leq \Pr\left(B^{\mathsf{LC}}_S\right) \leq \ell^{-c} \cdot \sum_{t=2}^{n} t^{2c} \cdot \mu_t^{\mathsf{LCY}^{(4c)}}.$$

Claim 12.2.6

$$\mu_t^{\mathsf{LCY}^{(4c)}} = \frac{2n \cdot t^{O(1)}}{(1+\varepsilon)^{t-1}}.$$

Proof. By Lemma 12.1.2, there are at most $t^{O(c)} = t^{O(1)}$ ways to choose a bipartite graph G in $\mathsf{LCY}^{(4c)}$ with t edges. Graph G cannot have more than $t+1$ nodes, since cyclic components have at most as many nodes as edges, and in the single leaf component, if present, the number of nodes is at most one bigger than the number of edges. In each component of G, there are two ways to assign the vertices to the two sides of the bipartition. After such an assignment is fixed, there are at most m^{t+1} ways to label the vertices with elements of $[m]$, and there are not more than n^t ways to label the t edges of G with labels from $\{1, \ldots, n\}$. Assume now such labels have been chosen for G. Draw t labeled edges according to the labeling of G from $[m]^2$ uniformly at random. The probability that they exactly fit the labeling of nodes and edges of G is $1/m^{2t}$. Thus,

$$\mu_t^{\mathsf{LCY}^{(4c)}} \leq \frac{2 \cdot m^{t+1} \cdot n^t \cdot t^{O(1)}}{m^{2t}} \leq \frac{2n \cdot t^{O(1)}}{(1+\varepsilon)^{t-1}}.$$

□

We use this claim and prove the lemma by the following calculation:

$$\Pr(B^{\mathsf{LC}}_S) \leq \ell^{-c} \sum_{t=2}^{n} t^{2c} \cdot \mu_t^{\mathsf{LCY}^{(4c)}} \leq \frac{2n}{\ell^c} \cdot \sum_{t=2}^{n} \frac{t^{O(1)}}{(1+\varepsilon)^{t-1}} = O\left(\frac{n}{\ell^c}\right).$$

□

13. Applications on Graphs

In this section, we will study different applications of our hash class in algorithms and data structures whose analysis relies on properties of the graph $G(S, h_1, h_2)$. We shall study four different applications:

- A variant of cuckoo hashing called *cuckoo hashing with a stash* introduced by Kirsch, Mitzenmacher, and Wieder in [KMW08].
- A construction for the simulation of a uniform hash function due to Pagh and Pagh [PP08].
- A construction of a (minimal) perfect hash function as described by Botelho, Pagh, and Ziviani [BPZ13].
- The randomness properties of hash class $\mathcal{Z}$ on connected components of $G(S, h_1, h_2)$.

As in the example from Section 11.4, each section will be divided into three parts. In the first part "Background", the data structure or algorithm will be introduced and other related work will be mentioned. The subsequent part "Result" will state the main result and give its proof. At the end, the part "Remarks and Discussion" will provide pointers to other results and discuss future work.

13.1. Cuckoo Hashing (with a Stash)

Background. The starting point of the ESA 2008 paper [KMW08] by Kirsch, Mitzenmacher, and Wieder was the observation that the rehash probability in cuckoo hashing is as large as $\Theta(1/n)$ (see Section 11.4), which can be too large for practical applications. They proposed adding a *stash*, an additional segment of storage that can hold up to s keys for some (constant) parameter s, and showed that this change reduces the rehash probability to $\Theta(1/n^{s+1})$. For details of the algorithm, see [KMW09]. The analysis given by Kirsch *et al.* requires the hash functions to be fully random. In the journal version [KMW09] Kirsch *et al.* posed "proving the above bounds for explicit hash families that can be represented, sampled, and evaluated efficiently" as an open problem.

Remark: The analysis of cuckoo hashing with a stash with a hash class similar to $\mathcal{Z}$ was the main topic of the author's diploma thesis [Aum10]. The full analysis of cuckoo hashing with a stash using hash class $\mathcal{Z}$ has been published in the paper [ADW14]. Here, it mainly serves as an example for the power of the framework developed in Section 11 in connection with the results of Section 12.

We focus on the question whether the pair (h_1, h_2) allows storing the key set S in the two tables with a stash of size s. This is equivalent to the question whether or not $G(S, h_1, h_2)$ is 1-orientable if we are allowed to remove not more than s edges from it.

It is known from [KMW09; Aum10] that a single parameter of $G = G(S, h_1, h_2)$ determines whether a stash of size s is sufficient to store S using (h_1, h_2), namely the *excess* $\mathrm{ex}(G)$.

Definition 13.1.1
The excess $\mathrm{ex}(G)$ of a graph G is defined as the minimum number of edges one has to remove from G so that all connected components of the remaining graph are acyclic or unicyclic.

The following lemma shows how the excess of a graph can be calculated.

Lemma 13.1.2 ***[KMW09]***
Let $G = (V, E)$ be a graph. Then

$$\mathrm{ex}(G) = \gamma(G) - \zeta_{\mathrm{cyc}}(G),$$

where $\zeta_{\mathrm{cyc}}(G)$ is the number of cyclic connected components in G.

Lemma 13.1.3 ***[KMW09]***
The keys from S can be stored in the two tables and a stash of size s using (h_1, h_2) if and only if $\mathrm{ex}(G(S, h_1, h_2)) \leq s$.

Result. The following theorem shows that one can replace the full randomness assumption of [KMW09] by hash functions from hash class $\mathcal{Z}$.

Theorem 13.1.4 ***[Aum10; ADW14]***
Let $\varepsilon > 0$ and $0 < \delta < 1$, let $s \geq 0$ be given. Assume $c \geq (s+2)/\delta$. For $n \geq 1$ consider $m \geq (1+\varepsilon)n$ and $\ell = n^{\delta}$. Let $S \subseteq U$ with $|S| = n$. Then for (h_1, h_2) chosen at random from $\mathcal{Z} = \mathcal{Z}_{\ell,m}^{c,2}$ the following holds:

$$\Pr(\mathrm{ex}(G(S, h_1, h_2)) \geq s+1) = O(1/n^{s+1}).$$

In view of Lemma 13.1.3, we identify minimal graphs with excess $s+1$.

Definition 13.1.5
An *excess-*$(s+1)$ *core graph* is a leafless graph G with excess exactly $s+1$ in which all connected components have at least two cycles. By CG^{s+1} we denote the set of all excess-$(s+1)$ core graphs in $\mathcal{G}_{m,n}^2$.

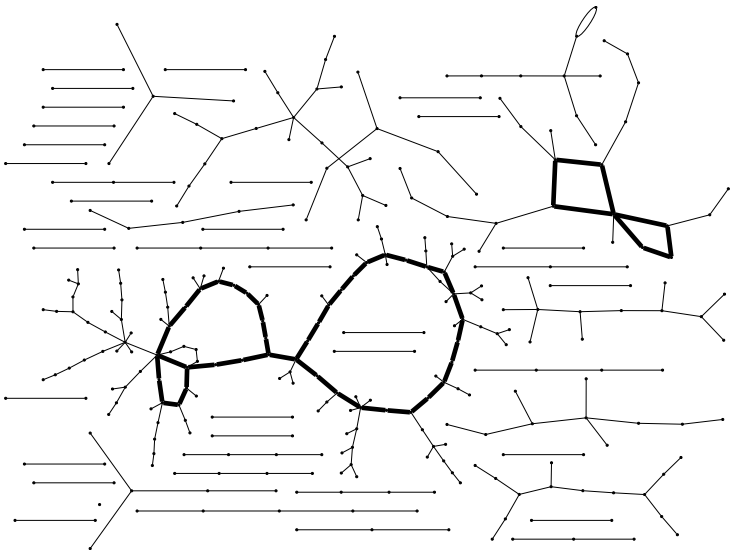

Figure 13.1.: An example of a graph that contains an excess-3 core graph (bold edges). This subgraph certifies that a stash of size at most 2 does not suffice to accommodate the key set. This figure can also be found in [Aum10].

An example for an excess-$(s+1)$ core graph is given in Figure 13.1.

Lemma 13.1.6
Let $G = G(S, h_1, h_2)$ with $\mathrm{ex}(G) \geq s + 1$. Then G contains an excess-$(s+1)$ core graph as a subgraph.

Proof. We obtain the excess-$(s+1)$ core graph by a peeling process, i. e., by repeatedly removing edges or connected components. Since $\mathrm{ex}(G) > 0$, G contains a connected component that is neither acyclic nor unicyclic (see Definition 13.1.1). Removing a cycle edge in such a component decreases the cyclomatic number by 1, but leaves the component cyclic. By Lemma 13.1.2, this decreases the excess by 1. We remove cycle edges in this way until the remaining graph has excess exactly $s + 1$. Subsequently we remove components that are trees or unicyclic. It is clear from Lemma 13.1.2 that this keeps the excess at $s + 1$. Finally we remove leaf edges one by one until the remaining graph is leafless. Again by Lemma 13.1.2, this does not change the excess. The resulting graph has excess exactly $s + 1$, no tree or unicyclic components, and is leafless. Thus, it is an excess-$(s + 1)$ core graph. □

Hence, to prove Theorem 13.1.4, it suffices to show that $\Pr\left(N_S^{\mathsf{CG}^{s+1}} > 0\right) = O\left(1/n^{s+1}\right)$. By Lemma 11.2.2, we know that

$$\Pr\left(N_S^{\mathsf{CG}^{s+1}} > 0\right) \leq \Pr\left(B_S^{\mathsf{CG}^{s+1}}\right) + \mathrm{E}^*\left(N_S^{\mathsf{CG}^{s+1}}\right). \tag{13.1}$$

Since $\mathsf{CG}^{s+1} \subseteq \mathsf{LL}$, we may apply Lemma 12.2.5 and write

$$\begin{aligned}\Pr\left(N_S^{\mathsf{CG}^{s+1}} > 0\right) &\leq O\left(\frac{n}{\ell^c}\right) + \mathrm{E}^*\left(N_S^{\mathsf{CG}^{s+1}}\right) \\ &= O\left(\frac{1}{n^{s+1}}\right) + \mathrm{E}^*\left(N_S^{\mathsf{CG}^{s+1}}\right),\end{aligned} \tag{13.2}$$

for the parameters used in Theorem 13.1.4. Thus, it remains to analyze the fully random case.

Lemma 13.1.7 ***[Aum10; ADW14]***
Let $\varepsilon > 0$ and let $s \geq 0$. Furthermore, let $S \subseteq U$ with $|S| = n$ be given. Set $m = (1+\varepsilon)n$. Then

$$\mathrm{E}^*\left(N_S^{\mathsf{CG}^{s+1}}\right) = O\left(\frac{1}{n^{s+1}}\right).$$

Before starting with the proof of this lemma, we remark that plugging its result into (13.2) proves Theorem 13.1.4. The following calculations also give an alternative, simpler proof of [KMW08, Theorem 2.1] for the fully random case, even if the effort needed to prove Lemma 12.1.2 is taken into account.

Proof of Lemma 13.1.7. We start by counting (unlabeled) excess-$(s+1)$ core graphs with t edges. By Lemma 13.1.2, a connected component C of such a graph G with cyclomatic number $\gamma(C)$ (which is at least 2) contributes $\gamma(C) - 1$ to the excess of G. This means that if G has $\zeta = \zeta(G)$ components, then $s + 1 = \gamma(G) - \zeta$ and $\zeta \leq s + 1$, and hence $\gamma = \gamma(G) \leq 2(s+1)$. Using Lemma 12.1.2, there are at most $N(t, 0, \gamma, \zeta) = t^{O(\gamma+\zeta)} = t^{O(s)}$ such graphs G. If from each component C of such a graph G we remove $\gamma(C) - 1$ cycle edges, we get unicyclic components, which have as many nodes as edges. This implies that G has $t - (s+1)$ nodes.

Now fix a bipartite (unlabeled) excess-$(s+1)$ core graph G with t edges and ζ components. There are $2^\zeta \leq 2^{s+1}$ ways of assigning the $t - s - 1$ nodes to the two sides of the bipartition, and then at most m^{t-s-1} ways of assigning labels from $[m]$ to the nodes. Thus, the number of bipartite graphs with property CG^{s+1}, where each node is labeled with one side of the bipartition and an element of $[m]$, and where the t edges are labeled with distinct elements from $\{1, \ldots, n\}$ is smaller than $n^t \cdot 2^{s+1} \cdot m^{t-s-1} \cdot t^{O(s)}$.

Now if a labeled $(s+1)$-core graph G is fixed, and we choose t edges with the labels used in G from $[m]^2$ uniformly at random, the probability that all edges match the labeling is $1/m^{2t}$.

For constant s, this yields the following bound:

$$\begin{aligned}\mathrm{E}^*\left(N_S^{\mathrm{CG}^{s+1}}\right) &\leq \sum_{s+3\leq t\leq n} \frac{2^{s+1}\cdot m^{t-s-1}\cdot n^t\cdot t^{O(s)}}{m^{2t}} \leq \frac{2^{s+1}}{n^{s+1}}\cdot \sum_{s+3\leq t\leq n}\frac{n^t\cdot t^{O(s)}}{m^t}\\ &= \frac{2^{s+1}}{n^{s+1}}\cdot \sum_{s+3\leq t\leq n}\frac{t^{O(s)}}{(1+\varepsilon)^t} = O\left(\frac{1}{n^{s+1}}\right). \end{aligned} \tag{13.3}$$

□

Remarks and Discussion. As in the previous example, our result only shows that the key set can be stored according to the rules of cuckoo hashing with a stash with a failure probability as low as in the fully random case. The analysis of the insertion algorithm has to consider the probability that there exist paths of length $\Theta((s+1)\cdot\log n)$ in $G(S,h_1,h_2)$. The exact analysis can be found in [ADW14] and can be summarized as follows. For bounding the impact of using hash class $\mathcal{Z}$ when considering long paths, graph property LC from the previous section shows that the failure term of using hash class $\mathcal{Z}$ can be made as low as $O(1/n^{s+1})$. (The graph property LC contains such long paths.) Then, only the fully random case must be analyzed, but the calculations are quite similar to standard calculations for cuckoo hashing, as already presented in [PR04; DM03; DW03].

Using the result that all simple paths in $G(S,h_1,h_2)$ have length $\Theta((s+1)\log n)$ with high probability also makes it possible to show that $\left((s+1)^2\log n\right)$-wise independence suffices to run cuckoo hashing with a stash, which can be achieved with constant evaluation time using the construction of Siegel [Sie04] or Thorup [Tho13]. In a nutshell, the reason for this degree of independence being sufficient is that the edges of an excess-$(s+1)$ core graph can be covered by $s+1$ simple paths and at most $2(s+1)$ additional edges. Since the length of a simple path is $O\left((s+1)\log n\right)$ with high probability, the required degree of independence follows. (The sum in (13.3) must only consider values $t\leq c\cdot(s+1)^2\log n$ for a suitable constant c.) Details can also be found in [ADW14, Section 5.3].

In another line of research, Goodrich and Mitzenmacher [GM11] considered an application whose analysis required certain properties of cuckoo hash tables with a stash of non-constant size. We can establish these properties even in the case of using hash class $\mathcal{Z}$. The calculations are analogous, but the failure bound we get is only $O(1/n^{s/2})$. For a rough idea of the proof, consider the calculations made in (13.3). There the $t^{O(s)}$ term requires special care, because s is non-constant. In [ADW14, Section 5.2], we approached this issue by splitting the n^{s+1} term into two terms being roughly $n^{s/2}$. One such term in the denominator of the calculations in (13.3) is sufficient to make the fraction with nominator $t^{O(s)}$ in the sum small enough.

13.2. Simulation of a Uniform Hash Function

Background. Consider a universe U of keys and a finite set R. Suppose we want to construct a hash function that takes on fully random values from R on a key set $S \subseteq U$ of size n. The naïve construction just assigns a random hash value to each key $x \in S$ and stores the key-value pair in a hash table that supports lookup in constant time and construction in expected time $O(n)$, e. g., cuckoo hashing (with a stash). For information theoretical reasons, this construction needs space at least $n \log |R|$. (See, e. g., [Rin14, Lemma 5.3.1].) We will now see that we can achieve much more in (asymptotically) almost the same space.

By the term "*simulating uniform hashing for U and R*" we mean an algorithm that does the following. On input $n \in \mathbb{N}$, a randomized procedure sets up a data structure DS_n that represents a hash function $h\colon U \to R$, which can then be evaluated efficiently for keys in U. For each set $S \subseteq U$ of cardinality n there is an event B_S that occurs with small probability such that conditioned on $\overline{B_S}$ the values $h(x)$, $x \in S$, are fully random. So, in contrast to the naïve construction from above, one h can be shared among many applications and works on each set $S \subseteq U$ of size n with high probability. The quality of the algorithm is determined by the space needed for DS_n, the evaluation time for h, and the probability of the event B_S, which we call the *failure probability of the construction*. It should be possible to evaluate h in constant time. Again, the information theoretical lower bound implies that at least $n \log |R|$ bits are needed to represent DS_n.

The first randomized constructions which matched this space bound up to constant factors were proposed independently by Dietzfelbinger and Woelfel [DW03] and Östlin and Pagh [ÖP03]. For $S \subseteq U$ and a pair (h_1, h_2) of hash functions, both constructions rely on properties of the bipartite graph $G(S, h_1, h_2)$. Next, we sketch these constructions. In the following, let R be the range of the hash function to be constructed, and assume that $(R, \oplus)$ is a commutative group. (For example, we could use $R = [t]$ with addition mod t.)

For the construction of Dietzfelbinger and Woelfel [DW03], let V and W be the vertex sets on the two sides of the bipartition of $G(S, h_1, h_2)$. In the construction phase, choose a pair (h_1, h_2) from $\mathcal{Z}$. Next, for an integer $s \geq 2$, with each vertex $v \in V$ associate a hash function $h_v\colon U \to R$ chosen at random from an s-independent hash family. With each vertex $w \in W$ associate a random element x_w from R. The evaluation of the function on a key $x \in U$ works as follows. Let $(v, w) = (h_1(x), h_2(x))$ be the edge that corresponds to x in $G(S, h_1, h_2)$. Then the hash value of x is just $h(x) = h_v(x) \oplus x_w$. Dietzfelbinger and Woelfel showed that for any given $S \subseteq U$ of size at most n this function is fully random on S with probability $1 - O(n^{-s/2})$, uses $(1+\varepsilon)n \log |R| + 1.5s^2(1+\varepsilon)n \log n + O(\log \log |U|) + o(n)$ bits of space[1] and has evaluation time $O(s)$. The construction runs in time $O(n)$.

The construction of Östlin and Pagh [ÖP03] works as follows: Each vertex v of $G(S, h_1, h_2)$ is associated with a random element x_v from R. The construction uses a third hash function

[1] Here, we applied the technique *collapsing the universe*, see, e. g., [DW03, Section 2.1] or [Rin14, Section 5.4.1], to reduce the universe to size $n^{(s+4)/2}$. The $o(n)$ term comes from the description length of a hash function from $\mathcal{Z}$.

$h_3 \colon U \to R$. All three hash functions have to be chosen from a n^δ-wise independent class of hash functions. (When this construction has been proposed, only Siegel's class of hash functions from [Sie04] achieved this degree of independence with constant evaluation time. Nowadays, one can use the construction of Thorup [Tho13] to make the evaluation more efficient.) Let $x \in U$ be an arbitrary key and let (v, w) be the edge that corresponds to x in $G(S, h_1, h_2)$. The hash value of x is $h(x) = x_v \oplus x_w \oplus h_3(x)$. This construction uses $8n \cdot \log |R| + o(n) + O(\log \log |U|)$ bits of space ([ÖP03] assumed that $m \geq 4n$ in their analysis) and achieves a failure probability of $O(1/n^s)$ for each $s \geq 1$. (The influence of s on the description length of the data structure is in the $o(n)+O(\log \log |U|)$ term. It is also in the construction time of the hash functions h_1, h_2, h_3.) The evaluation time is dominated by the evaluation time of the three highly-independent hash functions. The construction of [ÖP03] runs in time $O(n)$. In their full paper [PP08], Pagh and Pagh (the same authors as of [ÖP03]) introduced a general method to reduce the description length of their data structure to $(1+\varepsilon)n \log |R| + o(n) + O(\log \log |U|)$ bits, which is essentially optimal. This technique adds a summand of $O(1/\varepsilon^2)$ to the evaluation time.

Another construction was presented by Dietzfelbinger and Rink in [DR09]. It is based on results of Calkin [Cal97] and the "split-and-share" approach. Without going into details too much, the construction can roughly be described as follows: Choose $d \geq 1$, δ with $0 < \delta < 1$, and λ with $\lambda > 0$. Set $m = (1+\varepsilon) \cdot n^\delta$. Then, choose $n^{1-\delta}$ random tables $t_i[1..m]$, for $i \in [n^{1-\delta}]$, filled with random elements from R^m and two hash functions $h_{\text{split}} \colon U \to [n^{1-\delta}]$ and $h_{\text{map}} \colon U \to \binom{[m]}{d}$. The hash value of a key $x \in U$ is then $\bigoplus_{j \in h_{\text{map}}(x)} t_{h_{\text{split}}(x)}[j]$. A detailed overview over the construction and the involved hash functions can be found in [Rin14]. (Note that h_{split} uses a variant of $\mathcal{Z}$.) This construction has the currently asymptotically best performance parameters: $(1+\varepsilon)n \log |R| + o(n) + O(\log \log |U|)$ bits of space and evaluation time $O(\max\{\log^2(1/\varepsilon), s^2\})$ for failure probability $O(n^{1-(s+2)/9})$.

Our construction essentially duplicates the construction in [PP08] by replacing the highly independent hash functions with functions from hash class $\mathcal{Z}$. The data structure consists of a hash function pair (h_1, h_2) from our hash class, two tables of size $m = (1 + \varepsilon)n$ each, filled with random elements from R, a two-wise independent hash function with range R, $O(s)$ small tables with entries from R, and $O(s)$ two-independent hash functions to pick elements from these tables. The evaluation time of h is $O(s)$, and for $S \subseteq U$, $|S| = n$, the event B_S occurs with probability $O(1/n^{s+1})$. The construction requires roughly twice as much space as the most space-efficient solutions [DR09; PP08]. However, it seems to be a good compromise combining simplicity and fast evaluation time with moderate space consumption.

Result. The following result is also present in [ADW12].

Theorem 13.2.1
Given $n \geq 1, 0 < \delta < 1, \varepsilon > 0$, and $s \geq 0$, we can construct a data structure DS_n that allows us to compute a function $h\colon U \to R$ such that:

(i) For each $S \subseteq U$ of size n there is an event B_S of probability $O(1/n^{s+1})$ such that conditioned on $\overline{B_S}$ the function h is distributed uniformly on S.
(ii) For arbitrary $x \in U$, $h(x)$ can be evaluated in time $O(s/\delta)$.
(iii) DS_n comprises $2(1+\varepsilon)n \log |R| + o(n) + O(\log\log |U|)$ bits.

Proof. Choose an arbitrary integer $c \geq (s+2)/\delta$. Given U and n, set up DS_n as follows. Let $m = (1+\varepsilon)n$ and $\ell = n^\delta$, and choose and store a hash function pair (h_1, h_2) from $\mathcal{Z} = \mathcal{Z}^{c,2}_{\ell,m}$, with component functions $g_1, \ldots, g_c$ from $\mathcal{F}^2_\ell$. In addition, choose two random vectors $t_1, t_2 \in R^m$, c random vectors $y_1, \ldots, y_c \in R^\ell$, and choose f at random from a 2-wise independent family of hash functions from U to R.

Using DS_n, the mapping $h\colon U \to R$ is defined as follows:

$$h(x) = t_1[h_1(x)] \oplus t_2[h_2(x)] \oplus f(x) \oplus y_1[g_1(x)] \oplus \ldots \oplus y_c[g_c(x)].$$

DS_n satisfies (ii) and (iii) of Theorem 13.2.1. (If the universe is too large, it must be collapsed to size n^{s+3} first.) We show that it satisfies (i) as well. For this, let $S \subseteq U$ with $|S| = n$ be given.

First, consider only the hash functions (h_1, h_2) from $\mathcal{Z}$. By Lemma 12.2.5 we have $\Pr(B_S^{\mathsf{LL}}) = O(n/\ell^c) = O(1/n^{s+1})$. Now fix $(h_1, h_2) \notin B_S^{\mathsf{LL}}$, which includes fixing the components $g_1, \ldots, g_c$. Let $T \subseteq S$ be such that $G(T, h_1, h_2)$ is the 2-core of $G(S, h_1, h_2)$. The graph $G(T, h_1, h_2)$ is leafless, and since $(h_1, h_2) \notin B_S^{\mathsf{LL}}$, we have that (h_1, h_2) is T-good. Now we note that the part $f(x) \oplus \bigoplus_{1 \leq j \leq c} y_j[g_j(x)]$ of $h(x)$ acts exactly as one of our hash functions h_1 and h_2, where f and $y_1, \ldots, y_c$ are yet unfixed. So, arguing as in the proof of Lemma 11.1.3 we see that h is fully random on T.

Now assume that f and the entries in the tables $y_1, \ldots, y_c$ are fixed. Following [PP08], we show that the random entries in t_1 and t_2 alone make sure that $h(x)$, $x \in S - T$, is fully random. For the proof, let $(x_1, \ldots, x_p)$ be the keys in $S \setminus T$, ordered in such a way that the edge corresponding to x_i is a leaf edge in $G(T \cup \{x_1, \ldots, x_i\}, h_1, h_2)$, for each $i \in \{1, \ldots, p\}$. (To obtain such an ordering, repeatedly remove leaf edges from $G = G(S, h_1, h_2)$, as long as this is possible. The sequence of corresponding keys removed in this way is $x_p, \ldots, x_1$.) By induction, we show that h is uniform on $T \cup \{x_1, \ldots, x_p\}$. In the base case, we consider the set T and reasoning as above shows that h is fully random on T. For the step, we first observe that, by construction, the edge which corresponds to the key x_i is a leaf edge in $G(T \cup \{x_1, \ldots, x_i\}, h_1, h_2)$. Without loss of generality we may assume that $h_1(x_i)$ is the leaf. By the induction hypothesis, h is fully random on $T \cup \{x_1, \ldots, x_{i-1}\}$ and no key in $T \cup \{x_1, \ldots, x_{i-1}\}$ depends on the value $t_1[h_1(x_i)]$. Fix all values $t_1[h_1(x)]$ for $x \in T \cup \{x_1, \ldots, x_{i-1}\}$ and $t_2[h_2(x)]$ for $x \in T \cup \{x_1, \ldots, x_i\}$. Then $h(x_i)$ is uniformly distributed when choosing $t_1[h_1(x_i)]$ at random. □

Remarks and Discussion. When this construction was first described in [ADW12], it was the easiest to implement data structure to simulate a uniform hash function in almost optimal space. Nowadays, the construction of Pagh and Pagh can use the highly-independent hash family construction of Thorup [Tho13] instead of Siegel's construction. However, in the original analysis of Pagh and Pagh [PP08], the hash functions are required to be from an n^δ-wise independent hash class. It needs to be demonstrated by experiments that the construction of Pagh and Pagh in connection with Thorup's construction is efficient. We believe that using hash class $\mathcal{Z}$ is much faster.

Applying the same trick as in [PP08], the data structure presented here can be extended to use only $(1+\varepsilon)n$ words from R. The evaluation time of this construction is $O\left(\max\{\frac{1}{\varepsilon^2}, s\}\right)$.

13.3. Construction of a (Minimal) Perfect Hash Function

Background. A hash function $h : U \to [m]$ is perfect on $S \subseteq U$ if it is injective (or 1-on-1) on S. A perfect hash function is *minimal* if $|S| = m$. Here, S is assumed to be a static set. Perfect hash functions are usually applied when a large set of items is frequently queried and allows fast retrieval and efficient memory storage in this situation.

We start by giving a short overview over the history of perfect hashing. The content is mostly based on the survey papers of Czech, Havas and Majewski [CHM97] (for an extensive study of the developments until 1997) and Dietzfelbinger [Die07] (for developments until 2007), and the recent paper [BPZ13] of Botelho, Pagh, and Ziviani. For a given key set $S \subseteq U$ with size n and $m = n^2$, several constructions are known to build a perfect hash function with constant evaluation time and space consumption $O(\log n + \log\log|U|)$, see, e. g., [JEB86]. With respect to minimal perfect hash functions, Fredman and Komlós [FK84] proved that at least $n\log e + \log\log|U| - O(\log n)$ bits are required to represent a minimal perfect hash function when $|U| \geq n^\alpha$ for some constant $\alpha > 2$. Mehlhorn [Meh84] showed that this bound is essentially tight by providing a construction that needs at most $n\log e + \log\log|U| + O(\log n)$ bits. However, his construction has setup and evaluation time exponential in n, so it is not practical. Hagerup and Tholey [HT01] showed how to improve this construction using a randomized approach. Their construction achieves space consumption $n\log e + \log\log|U| + O(n \cdot (\log\log n)^2/\log n + \log\log\log|U|)$, with constant evaluation time and $O(n + \log\log|U|)$ construction time in expectation. However, for values of n arising in practice, it is not practical, as discussed in [BPZ13].

In their seminal paper, Fredman, Komlós, and Szemerédi [FKS84] introduced the *FKS scheme*, which can be used to construct a minimal perfect hash function that has description length $O(n\log n + \log\log|U|)$ bits and constant evaluation time. Building upon [FKS84], Schmidt and Siegel [SS90] gave the first algorithm to construct a minimal perfect hash function with constant evaluation time and space $O(n + \log\log|U|)$ bits. According to [BPZ13], high constants are hidden in the big-Oh notation.

There exist many constructions that are (assumed to be) more practical. Majewski, Wormald,

Havas and Czech [Maj+96] proposed a family of algorithms to construct a minimal perfect hash function with constant evaluation time and description length $O(n \log n)$ bits based on a hypergraph approach. Other hypergraph approaches that achieve essentially the same description length (up to a constant factor) are [CHM92] and [BKZ05]. A different approach was described in Pagh [Pag99]. His approach used a method called *hash-and-displace* and achieved constant evaluation time with space $O(n \log n)$ bits, as well. The constant in the description length of his construction was decreased by about a factor of two by Dietzfelbinger and Hagerup [DH01]. Later, Woelfel [Woe06b] extended this construction to use only $O(n \log \log n)$ bits. Finally, Belazzougui, Botelho and Dietzfelbinger [BBD09] showed how to decrease the space further to only $O(n)$ bits with a practical algorithm.

The first explicit practical construction of a (minimal) perfect hash function which needs only $O(n)$ bits is due to Botelho, Pagh, and Ziviani [BPZ07] (full version [BPZ13]) and is again based on the hypergraph approach. The idea of their algorithm was discovered before by Chazelle, Kilian, Rubinfeld, and Tal [Cha+04], but without a reference to its use in perfect hashing. Furthermore, Botelho *et al.* discovered that *acyclic* hypergraphs admit a very efficient algorithm for the construction of a perfect hash function, a connection that was not described in [Cha+04]. We will focus our work on their construction. In [BPZ13], explicit hash functions based on the "split-and-share" approach were used. This technique builds upon a general strategy described by Dietzfelbinger in [Die07] and Dietzfelbinger and Rink in [DR09] to make the "full randomness assumption" feasible in the construction of a perfect hash function. Botelho *et al.* showed in experiments that their construction is very practical, even when realistic hash functions are used. Our goal is to show that hash functions from class $\mathcal{Z}$ can be used in a specific version of their construction as well. In the remarks at the end of this section, we will speculate about differences in running time between the split-and-share approach of [BPZ13] and hash class $\mathcal{Z}$.

We start by explaining the construction of [BPZ13] to build a perfect hash function from S to $[2m]$. We restrict the discussion to graphs to show its simplicity. The first step in the construction is to pick two hash functions $h_1, h_2 \colon U \to [m]$ and build the graph $G(S, h_1, h_2)$. If $G(S, h_1, h_2)$ contains cycles, choose a new pair of hash functions (h_1, h_2) and rebuild the graph. This is iterated until $G(S, h_1, h_2)$ is acyclic. Then build a perfect hash function from S to $[2m]$ from $G = G(S, h_1, h_2)$ as follows: First, obtain a 1-orientation of G by iteratively removing leaf edges. Now, initialize two bit vectors $\mathsf{g}_1[1..m]$ and $\mathsf{g}_2[1..m]$. Our goal is to set the bits in such a way that for each edge (u, v) in G, given the 1-orientation, we have that

$$\mathsf{g}_1[u] + \mathsf{g}_2[v] \mod 2 = \begin{cases} 0, & (u,v) \text{ points towards } u, \\ 1, & \text{otherwise.} \end{cases}$$

The perfect hash function h from S to $[2m]$ is defined by the mapping

$$\begin{aligned} x &\mapsto j \cdot m + h_{1+j}(x), \qquad \text{for } x \in S, \\ \text{with } j &\leftarrow \mathsf{g}_1[h_1(x)] + \mathsf{g}_2[h_2(x)] \mod 2. \end{aligned} \tag{13.4}$$

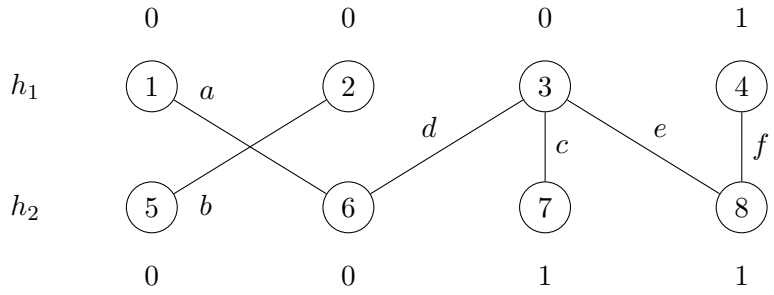

Figure 13.2.: Example for the construction of a perfect hash function for $S = \{a, b, c, d, e, f\}$ from acyclic $G(S, h_1, h_2)$. The peeling process peels the labeled edges in the order b, f, c, e, d, a. The g-value of each node is written next to it. The reader is invited to check that the hash function described in (13.4) uses the mapping "$a \mapsto 1$, $b \mapsto 2$, $c \mapsto 7$, $d \mapsto 3$, $e \mapsto 8$, $f \mapsto 4$".

Setting the bits for each vertex can be done very efficiently when $G(S, h_1, h_2)$ is acyclic and thus can be "peeled" by iteratively removing leaf edges. It works in the following way: Initialize both bit vectors to contain only 0-entries. Mark all vertices as being "unvisited". Let $e_1, \ldots, e_n$ be the sequence of removed edges *in reversed order*, i. e., e_n is the edge that was removed first when obtaining the 1-orientation, and so on. Next, consider each edge in the order $e_1, \ldots, e_n$. For an edge $e_i = (u, v)$ do the following: If u is already visited, set $\mathsf{g}_2[v] \leftarrow 1 - \mathsf{g}_1[u]$. (The hash value of the key associated with the edge is fixed to be $m + h_2(x)$.) If v is already visited, set $\mathsf{g}_1[u] \leftarrow \mathsf{g}_2[v]$. (The hash value of the key associated with the edge is fixed to be $h_1(x)$.) Next, mark u and v as being visited and proceed with the next edge. Note that by the ordering of the edges it cannot happen that both vertices have been visited before. Figure 13.2 gives an example of the construction. To build a minimal perfect hash function, Botelho *et al.* use a compression technique called "ranking" well-known from succinct data structures to compress the perfect hash function further. (See References [40,42,45] in [BPZ13].)

In [BPZ13] it is shown that the probability of $G(S, h_1, h_2)$ being acyclic, i. e., the probability that the construction succeeds, for fully random h_1, h_2 and $m = (1 + \varepsilon)|S|$ is

$$\sqrt{1 - \left(\frac{1}{1+\varepsilon}\right)^2}. \tag{13.5}$$

Result. We will show that for a key set $S \subseteq U$ of size n, and for $m \geq 1.08n$, we can build a perfect hash function for a key set S by applying the construction of Botelho *et al.* a constant number of times (in expectation). To see this, we only have to prove the following lemma.

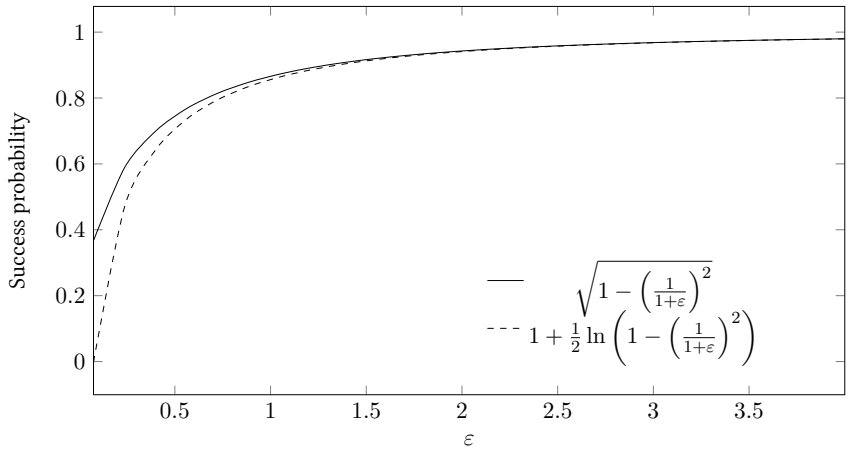

Figure 13.3.: Comparison of the probability of a random graph being acyclic and the theoretical bound following from a first moment approach for values $\varepsilon \in [0.08, 4]$.

Lemma 13.3.1
Let $S \subseteq U$ with $S = n$. Let $\varepsilon \geq 0.08$, and let $m \geq (1+\varepsilon)n$. Set $\ell = n^\delta$ and $c \geq 1.25/\delta$. For a randomly chosen pair $(h_1, h_2) \in \mathcal{Z}^{c,2}_{\ell,m}$, we then have

$$\Pr(G(S, h_1, h_2) \text{ is acyclic}) \geq 1 + \frac{1}{2}\ln\left(1 - \left(\frac{1}{1+\varepsilon}\right)^2\right) - o(1). \tag{13.6}$$

Figure 13.3 depicts the difference between the functions (13.5) and (13.6). The theoretical bound using a first moment approach is close to the behavior in a random graph when $\varepsilon \geq 1$.

Proof of Lemma 13.3.1. For the proof, we let CYC be the set of all cycles in $\mathcal{G}^2_{m,n}$. (Note that all these cycles have even length, since we consider bipartite graphs.) By Lemma 11.2.2, we may bound $\Pr(N_S^{\mathsf{CYC}} > 0)$ by $\Pr(B_S^{\mathsf{CYC}}) + \mathrm{E}^*(N_S^{\mathsf{CYC}})$. Since $\mathsf{CYC} \subseteq \mathsf{LL}$, we know that $\Pr(B_S^{\mathsf{CYC}}) = O(n/\ell^c)$, see Lemma 12.2.5. For the parameter choices $\ell = n^\delta$ and $c \geq 1.25/\delta$ we have $\Pr(B_S^{\mathsf{CYC}}) = o(1)$. We now focus on the second summand and calculate (as in [BPZ13]):

$$\begin{aligned}\mathrm{E}^*\left(N_S^{\mathsf{CYC}}\right) &= \sum_{t=1}^{n/2} \mu_{2t}^{\mathsf{CYC}} \leq \sum_{t=1}^{n/2} \frac{\binom{n}{2t}(2t)! \cdot m^{2t}}{2t \cdot m^{2\cdot 2t}} = \sum_{t=1}^{n/2} \frac{\binom{n}{2t} \cdot (2t)!}{2t \cdot m^{2t}} \leq \sum_{t=1}^{n/2} \frac{n^{2t}}{2t \cdot m^{2t}} \\ &= \sum_{t=1}^{n/2} \frac{1}{2t \cdot (1+\varepsilon)^{2t}} \leq \sum_{t=1}^{\infty} \frac{1}{2t \cdot (1+\varepsilon)^{2t}} = -\frac{1}{2}\ln\left(1 - \left(\frac{1}{1+\varepsilon}\right)^2\right),\end{aligned}$$

where the last step is Maclaurin expansion. □

Remarks and Discussion. According to Lemma 13.3.1, we can build a perfect hash function with range $[2.16n]$ with a constant number of constructions of $G(S, h_1, h_2)$ (in expectation). To store the data structure we need $2.16n$ bits (to store g_1 and g_2), and $o(n)$ bits to store the pair (h_1, h_2) from $\mathcal{Z}$. For example, for a set of $n = 2^{32}$ keys, i. e., about 4.3 billion keys, the pair (h_1, h_2) may consist of ten tables with 256 entries each, five 2-universal hash functions, and two 2-independent hash functions, see Lemma 13.3.1 with parameters $c = 5$ and $\delta = 1/4$. This seems to be more practical than the split-and-share approach from [BPZ13] which uses more and larger tables per hash function, *cf.* [BPZ13, Section 4.2]. However, it remains future work to demonstrate in experiments how both approaches compare to each other. To obtain a minimal perfect hash function, one has to compress the perfect hash function further. This roughly doubles the description length, see [BPZ13] for details.

In their paper [BPZ13], Botelho *et al.* showed that minimal space usage is achieved when using three hash functions h_1, h_2, h_3 to build the hypergraph $G(S, h_1, h_2, h_3)$. In this case, one can construct a perfect hash function with range $[1.23n]$ with high probability. Since the g-values must then index three hash functions, $1.23n \cdot \log_2 3 \approx 1.95n$ bits are needed to store the bit vectors. According to [BPZ13], the minimal perfect hash function needs about $2.62n$ bits.

In Section 14.1, we will consider the orientability of the hypergraph $G(S, h_1, h_2, h_3)$ when hash functions are picked from $\mathcal{Z}_{\ell,m}^{c,3}$. Our result will be far away from the result of Botelho *et al.*: We can prove the construction to work (with high probability) only when we aim to build a perfect hash function with range of size at least $6(1 + \varepsilon)n$, for $\varepsilon > 0$, which is inferior to the construction with only two hash functions discussed here.

13.4. Connected Components of $G(S, h_1, h_2)$ are small

Background. As is well known from the theory of random graphs, for a key set $S \subseteq U$ of size n and $m = (1 + \varepsilon)n$, for $\varepsilon > 0$, and fully random hash functions $h_1, h_2 : U \to [m]$ the graph $G(S, h_1, h_2)$ contains w.h.p. only components of at most logarithmic size which are trees or unicyclic. (This is the central argument for standard cuckoo hashing to work.) We show here that hash class $\mathcal{Z}$ can provide this behavior if one is willing to accept a density that is smaller by a constant factor. Such situations have been considered in the seminal work of Karp, Luby, and Meyer auf der Heide [KLM96] on the simulation of shared memory in distributed memory machines.

Result. We give the following result as a corollary. It has first appeared in [KLM96].

Corollary 13.4.1 ***[KLM96, Lemma 6.3]***
Let $S \subseteq U$ with $|S| = n$. Let $m \geq 6n$. Then for each $\alpha \geq 1$, there are $\beta, \ell, c, s \geq 1$ such that for $G = G(S, h_1, h_2)$ with $(h_1, h_2) \in \mathcal{Z}^{c,2}_{\ell,m}$ we have that

(a) $\Pr(G$ has a connected component with at least $\beta \log n$ vertices$) = O(n^{-\alpha})$.

(b) $\Pr(G$ has a connected component with k vertices and $\geq k + s - 1$ edges$) = O(n^{-\alpha})$.

Proof. We start with the proof of (b). If $G = G(S, h_1, h_2)$ has a connected component A with k vertices and at least $k + s - 1$ edges, then $\mathrm{ex}(G) \geq s - 1$. According to Theorem 13.1.4, the probability that such a component appears is $O(1/n^\alpha)$, for $s = \alpha$ and $c \geq 2(\alpha + 1)$. The proof of Part (a) requires more care.

To prove (a) we may focus on the probability that G contains a tree with $k = \beta \log n$ vertices. We let T consist of all trees with k vertices in $\mathcal{G}_{m,n}$ and apply Lemma 11.2.2 to get

$$\Pr\left(N_S^{\mathsf{T}} > 0\right) \leq \Pr\left(B_S^{\mathsf{T}}\right) + \mathrm{E}^*\left(N_S^{\mathsf{T}}\right). \tag{13.7}$$

Since $\mathsf{T} \subseteq \mathsf{LCY}$, we have that $\Pr(B_S^{\mathsf{T}}) = O(n/\ell^c)$, see Lemma 12.2.5. We now bound the second summand of (13.7). Note that the calculations are essentially the same as the ones made in [KLM96] to prove their Lemma 6.3. By Caley's formula we know that there are k^{k-2} labeled trees with vertex set $\{1, \ldots, k\}$. Fix such a tree T^*. We can label the edges of T^* with $k-1$ keys from S in $\binom{n}{k-1} \cdot (k-1)!$ many ways. Furthermore, there are two ways to fix which vertices of T^* belong to which side of the bipartition. After this, there are not more than $\binom{2m}{k}$ ways to assign the vertices of T^* to vertices in the bipartite graph $G(S, h_1, h_2)$. Once all these labels of T^* are fixed, the probability that the hash values of (h_1, h_2) realize T^* is $1/m^{2(k-1)}$. We can thus calculate:

$$\begin{aligned}\mathrm{E}^*(N_S^{\mathsf{T}}) &\leq \frac{\binom{n}{k-1} \cdot k^{k-2} \cdot 2 \cdot (k-1)! \cdot \binom{2m}{k}}{m^{2(k-1)}} \leq \frac{2^{k+1} \cdot m^2 \cdot k^{k-2}}{6^k \cdot k!} \\ &\leq \frac{2^{k+1} \cdot m^2 \cdot k^{k-2}}{6^k \cdot \sqrt{2\pi k} \cdot (k/e)^k} \leq 2m^2 \cdot \left(\frac{e}{3}\right)^k\end{aligned}$$

Part (*a*) follows for $k = \Omega(\log n)$. □

Remarks and Discussion. The authors of [KLM96] use a variant of hash class $\mathcal{Z}$ combined with functions from Siegel's hash class to obtain a hash class with high ($\sqrt{n}$-wise) independence. They need this high level of independence in the proof of their Lemma 6.3, which states properties of the connected components in the graph built from the key set and this highly independent hash functions. Replacing Lemma 6.3 in [KLM96] with our Corollary 13.4.1 immediately implies

that the results of [KLM96], in particular, their Theorem 6.4, also hold when (only) using hash functions from $\mathcal{Z}$. In particular, in [KLM96] the sparse setting where m is at least $6n$ was considered as well.

Moreover, this result could be applied to prove results for cuckoo hashing (with a stash), and de-amortized cuckoo hashing of Arbitman *et al.* [ANS09]. However, note that while in the fully random case the statement of Corollary 13.4.1 holds for $m = (1 + \varepsilon)n$, here we had to assume $m \geq 6n$, which yields only very low hash table load. We note that this result cannot be improved to $(1 + \varepsilon)n$ using the first moment approach inherent in our approach and the approach of [KLM96] (for $\sqrt{n}$-wise independence), since the number of unlabeled trees that have to be considered in the first moment approach is too large [Ott48]. It remains open to show that graphs built with our class of hash functions have small connected components for all $\varepsilon > 0$.

14. Applications on Hypergraphs

In this chapter we discuss some applications of hash class $\mathcal{Z}$ in the setting that we use more than two hash functions, i. e., each edge of $G(S, \vec{h})$ contains at least three vertices. We will study three different applications: Generalized cuckoo hashing with $d \geq 3$ hash functions as proposed by Fotakis, Pagh, Sanders, and Spirakis [Fot+05], two recently described insertion algorithms for generalized cuckoo hashing due to Khosla [Kho13] and Eppstein, Goodrich, Mitzenmacher, and Pszona [Epp+14], and different schemes for load balancing as studied by Schickinger and Steger [SS00].

For applications regarding generalized cuckoo hashing, we will study the failure term of $\mathcal{Z}$ on the respective graph properties directly. We will show that $\mathcal{Z}$ allows running these applications efficiently. However, we have to assume that the load of the hash table is rather low. For the application with regard to load balancing schemes, the failure term of $\mathcal{Z}$ will be analyzed by means of a very general graph property. However, it requires higher parameters when setting up a hash function from $\mathcal{Z}$, which degrades the performance of these hash functions. We will start by introducing some notation.

Hypergraph Notation. A hypergraph extends the notion of an undirected graph by allowing edges to consist of more than two vertices. We use the hypergraph notation from [SPS85; KŁ02]. A hypergraph is called *d-uniform* if every edge contains exactly d vertices. Let $H = (V, E)$ be a hypergraph. A *hyperpath from u to v* in H is a sequence $(u = u_1, e_1, u_2, e_2, \ldots, e_{t-1}, u_t = v)$ such that $e_i \in E$ and $u_i, u_{i+1} \in e_i$, for $1 \leq i \leq t-1$. The hypergraph H is *connected* if for each pair of vertices $u, v \in V$ there exists a hyperpath from u to v.

The *bipartite representation* of a hypergraph H is the bipartite graph $\mathrm{bi}(H)$ where vertices of H are the vertices on the right side of the bipartition, the edges of H correspond to vertices on the left side of the bipartition, and two vertices are connected by an edge in the bipartite graph if the corresponding edge in the hypergraph contains the corresponding vertex.

We will use a rather strict notion of cycles in hypergraphs. A connected hypergraph is called a *hypertree* if $\mathrm{bi}(H)$ is a tree. A connected hypergraph is called *unicyclic* if $\mathrm{bi}(H)$ is unicyclic. A connected hypergraph that is neither a hypertree nor unicyclic is called *complex*. Using the standard formula to calculate the cyclomatic number of a graph[1] [Die05], we get the following (in)equalities for a connected d-uniform hypergraph H with n edges and m vertices: $(d-1) \cdot n = m - 1$ if H is a hypertree, $(d-1) \cdot n = m$ if H is unicyclic, and $(d-1) \cdot n > m$ if H is complex.

We remark that there are different notions with respect to cycles in hypergraphs. In other

[1] The cyclomatic number of a connected graph G with m vertices and n edges is $n - m + 1$.

papers, e. g., [CHM97; Die07; BPZ13], a hypergraph is called *acyclic* if and only if there exists a sequence of repeated deletions of edges containing at least one vertex of degree 1 that yields a hypergraph without edges. (Formally, we can arrange the edge set $E = \{e_1, \ldots, e_n\}$ of the hypergraph in a sequence $(e'_1, \ldots, e'_n)$ such that $e'_j - \bigcup_{s<j} e'_s \neq \emptyset$, for $1 \leq j \leq n$.) We will call this process of repeatedly removing edges incident to a vertex of degree 1 the *peeling process*, see, e. g., [Mol05]. With respect to this definition, a hypergraph H is acyclic if and only if the 2-core of H is empty, where the 2-core is the maximal subgraph of H in which each vertex has minimum degree 2. An acyclic hypergraph, according to this definition, can have unicyclic and complex components according to the definition from above. In the analysis, we will remark why it is important for our work to use the notation introduced above.

14.1. Generalized Cuckoo Hashing

Background. The obvious extension of cuckoo hashing is to use a sequence $\vec{h} = (h_1, \ldots, h_d)$ of $d \geq 3$ hash functions. For a given integer $d \geq 3$ and a key set $S \subseteq U$ with $|S| = n$, our hash table consists of d tables $T_1, \ldots, T_d$, each of size $m = O(n)$, and uses d hash functions $h_1, \ldots, h_d$ with $h_i \colon U \to [m]$, for $i \in \{1, \ldots, d\}$. A key x can be stored either in $T_1[h_1(x)], T_2[h_2(x)], \ldots,$ or $T_d[h_d(x)]$. Each table cell contains at most one key. Searching and removing a key works in the obvious way. For the insertion procedure, note that evicting a key y from a table T_j leaves, in contrast to standard cuckoo hashing, $d - 1$ other choices where to put the key. To think about insertion procedures, it helps to introduce the concept of a certain directed graph. Given a set S of keys stored in a cuckoo hash table with tables $T_1, \ldots, T_d$ using $\vec{h}$, we define the following (directed) cuckoo allocation graph $G = (V, E)$, see, e. g., [Kho13]: The vertices V correspond to the memory cells in $T_1, \ldots, T_d$. The edge set E consists of all edges $(u, v) \in V \times V$ such that there exists a key $x \in S$ so that x is stored in the table cell which corresponds to vertex u (x *occupies* u) and v corresponds to one of the $d - 1$ other choices of key x. If $u \in V$ has out-degree 0, we call u *free*. (The table cell which corresponds to vertex u does not contain a key.) Traditionally, see [Fot+05], the following strategies were suggested for inserting a key x:

- **Random Walk**: Test whether one of the d possible table cells for key x is empty or not. If this is the case, then store x in such a cell; the insertion terminates successfully. Otherwise pick one out of the d table cells at random. Let y be the key that occupies this table cell. The key y is evicted before x is stored into the cell. Now the insertion continues analogously with the key y. This strategy corresponds to a random walk in the cuckoo allocation graph. The insertion fails if the number of evicted keys is larger than a given threshold, e. g., after poly-logarithmically many steps, see [FPS13].

- **Breadth-First Search**: Starting from the d choices of x, use breadth-first search in the cuckoo allocation graph to systematically scan all possible eviction sequences of length $1, 2, \ldots$ until a free vertex, i. e., an empty cell, has been found. If such a cell exists, move

elements along the path in the cuckoo allocation graph to accommodate x. Otherwise the insertion fails.

(In the next section, we will study two alternative insertion strategies that were suggested recently.) If an insertion fails, a new sequence of hash functions is chosen and the data structure is built anew.

In the original analysis, Fotakis *et al.* [Fot+05] proved that for given $\varepsilon > 0$, $d \geq 2(1+\varepsilon)\ln(e/\varepsilon)$ fully random hash functions suffice to store n elements into $(1+\varepsilon)n$ table cells with high probability according to the cuckoo hashing rules. Later, it was fully understood what table sizes m makes it possible to store w.h.p. a key set according to the cuckoo hashing rules for a given number of hash functions. In 2009,[2] this case was settled independently by Dietzfelbinger *et al.* [Die+10], Fountoulakis and Panagiotou [FP10], and Frieze and Melsted [FM12]. Later, the random walk insertion algorithm was partially analyzed by Frieze, Melstedt, and Mitzenmacher [FMM11] and Fountoulakis, Panagiotou, and Steger [FPS13].

Here, we study the static setting in which we ask if $\vec{h}$ allows accommodating a given key set $S \subseteq U$ in the hash table according to the cuckoo hashing rules. As in the case of standard cuckoo hashing, this is equivalent to the question whether the hypergraph $G(S, \vec{h})$ built from S and $\vec{h}$ is 1-orientable or not, recall the definition in Section 11.4. If $G(S, \vec{h})$ is 1-orientable, we call $\vec{h}$ *suitable* for S.

We now discuss some known results for random hypergraphs. As for simple random graphs [ER60] there is a sharp transition phenomenon for random hypergraphs [KŁ02]. When a random hypergraph with m vertices has at most $(1-\varepsilon)m/(d(d-1))$ edges, all components are small and all components are *hypertrees* or *unicyclic* with high probability. On the other hand, when it has at least $(1+\varepsilon)m/(d(d-1))$ edges, there exists one large, *complex* component. We will analyze generalized cuckoo hashing under the assumption that each table has size $m \geq (1+\varepsilon)(d-1)n$, for $\varepsilon > 0$. Note that this result is rather weak: The load of the hash table is at most $1/(d(d-1))$, i. e., the more hash functions we use, the weaker our bounds for provable working hash functions are. At the end of this section, we will discuss whether this result can be improved or not with the methodology used here.

Result. We will show the following theorem.

Theorem 14.1.1
Let $\varepsilon > 0, 0 < \delta < 1$, $d \geq 3$ be given. Assume $c \geq 2/\delta$. For $n \geq 1$, consider $m \geq (1+\varepsilon)(d-1)n$ and $\ell = n^\delta$. Let $S \subseteq U$ with $|S| = n$. Then for $\vec{h} = (h_1, \ldots, h_d)$ chosen at random from $\mathcal{Z} = \mathcal{Z}_{\ell,m}^{c,d}$ the following holds:

$$\Pr\left(\vec{h} \text{ is not suitable for } S\right) = O(1/n).$$

[2] Technical report versions of all these papers were published at `www.arxiv.org`. We refer to the final publications.

Lemma 14.1.2
Let H be a hypergraph. If H contains no complex component then H is 1-orientable.

Proof. We may consider each connected component of H separately. We claim that a hypertree and a unicyclic component always contains an edge that is incident to a vertex of degree 1. By contraposition, we will show that if every vertex in a connected hypergraph with m vertices and n edges has degree at least 2, then $(d-1)\cdot n > m$, i. e., the hypergraph is complex. So, let C be an arbitrary connected hypergraph with m vertices and n edges, and let c^* be the minimum degree of a vertex in C. By assumption, it holds that $d \cdot n \geq c^* \cdot m$, and we may calculate:

$$(d-1)\cdot n \geq m + (c^*-1)\cdot m - n \overset{(!)}{>} m.$$

So, all we have to do is to show that $(c^*-1)m > n$. This follows since $n \geq c^* m/d$ and $c^*(1-1/d) > 1$ for $d \geq 3$ and $c^* \geq 2$. So, every hypertree or unicyclic component contains a vertex of degree 1.

Suppose C is such a hypertree or a unicyclic component. A 1-orientation of C is obtained via the well-known "peeling process", see, e. g., [Mol05]. It works by iteratively peeling edges incident to a vertex of degree 1 and orienting each edge towards such a vertex. □

In light of Lemma 14.1.2 we bound the probability of $G(S,\vec{h})$ being 1-orientable by the probability that $G(S,\vec{h})$ contains no complex component. A connected complex component of H contains at least two cycles in $\mathrm{bi}(G)$. So, minimal obstruction hypergraphs that show that a hypergraph contains a complex component are very much like the obstruction graphs that showed that a graph contains more than one cycle, see Figure 11.1 on Page 120. For a clean definition of obstruction hypergraphs, we will first introduce the concept of a *strict path* in a hypergraph. A sequence $(e_1,\ldots,e_t), t \geq 1, e_i \in E, 1 \leq i \leq t$, is a *strict path* in H if $|e_i \cap e_{i+1}| = 1, 1 \leq i \leq t-1$, and $|e_i \cap e_j| = 0$ for $j \geq i+2$ and $1 \leq i \leq t-2$. According to [KŁ02], a complex connected component contains a subgraph of one of the following two types:

Type 1: A strict path $e_1,\ldots,e_t, t \geq 1$, and an edge f such that $|f \cap e_1| \geq 1$, $|f \cap e_t| \geq 1$, and

$$\left| f \cap \bigcup_{i=1}^{t} e_i \right| \geq 3.$$

Type 2: A strict path $e_1,\ldots,e_{t-1}, t \geq 2$, and edges f_1, f_2 such that $|f_1 \cap e_1| \geq 1$, $|f_2 \cap e_{t-1}| \geq 1$, and

$$\left| f_j \cap \bigcup_{i=1}^{t-1} e_i \right| \geq 2, \quad \text{for } j \in \{1,2\}.$$

Hypergraphs of Type 1 have a cycle with a chord in their bipartite representation, hypergraphs of Type 2 contain two cycles connected by a path of length $t \geq 0$ in their bipartite representation. We call a hypergraph H in $\mathcal{G}^d_{m,n}$ which is of Type 1 or Type 2 a *minimal complex obstruction hypergraph*. Let MCOG denote the set of all minimal complex obstruction hypergraphs in $\mathcal{G}^d_{m,n}$. In the following, our objective is to apply Lemma 11.2.2, which says that

$$\Pr\left(N_S^{\mathsf{MCOG}} > 0\right) \leq \Pr\left(B_S^{\mathsf{MCOG}}\right) + \mathrm{E}^*\left(N_S^{\mathsf{MCOG}}\right). \tag{14.1}$$

Bounding $\mathrm{E}^*\left(N_S^{\mathsf{MCOG}}\right)$. We will now prove the following lemma:

Lemma 14.1.3
Let $S \subseteq U$ with $|S| = n$, $d \geq 3$, and $\varepsilon > 0$ be given. Set $m \geq (1+\varepsilon)(d-1)n$. Then

$$\mathrm{E}^*\left(N_S^{\mathsf{MCOG}}\right) = O(1/n).$$

Proof. The proof follows [KŁ02]. We begin by counting the number $w_1(d, t+1)$ of fully labeled hypergraphs of Type 1 having exactly $t+1$ edges. There are not more than $dm^d \cdot ((d-1)m)^{t-1}$ ways to choose a strict path $e_1, \ldots, e_t$. When such a path is fixed, there are at most $td^3 \cdot m^{d-3}$ ways to choose the additional edge f from the definition of Type 1 minimal complex obstruction hypergraphs. So, we obtain the bound

$$w_1(d, t+1) \leq dm^d \cdot \left((d-1) \cdot m^{d-1}\right)^{t-1} \cdot td^3 \cdot m^{d-3}. \tag{14.2}$$

By a similar argument, for the number $w_2(d, t+1)$ of fully labeled hypergraphs of Type 2 having exactly $t+1$ edges we get

$$w_2(d, t+1) \leq dm^d \cdot \left((d-1) \cdot m^{d-1}\right)^{t-2} \cdot t^2 d^4 \cdot m^{2(d-2)}. \tag{14.3}$$

In total, the number of fully labeled minimal complex obstruction hypergraphs with exactly $t+1$ edges not larger than

$$\begin{aligned}
&\left(\binom{n}{t+1} \cdot (t+1)!\right) \cdot (w_1(d, t+1) + w_2(d, t+2)) \\
&\leq n^{t+1} \cdot dm^d \cdot \left((d-1)\, m^{d-1}\right)^{t-2} \cdot t^2 d^4 \cdot \left((d-1)\, m^{d-1} \cdot m^{d-3} + m^{2(d-2)}\right) \\
&= n^{t+1} \cdot d^6 \cdot m^{(d-1)(t+1)-1} \cdot t^2 \cdot (d-1)^{t-2} \\
&\leq n^{d(t+1)-1} \cdot d^6 \cdot t^2 \cdot (1-\varepsilon)^{(d-1)(t+1)-1} \cdot (d-1)^{(d-1)(t+1)+t-3} \\
&= n^{d(t+1)-1} \cdot d^6 \cdot t^2 \cdot (1+\varepsilon)^{(d-1)(t+1)-1} \cdot (d-1)^{d(t+1)-4}.
\end{aligned}$$

Let H be a fully labeled minimal complex obstruction hypergraph with $t+1$ edges.

Draw $t+1$ edges at random from $[m]^d$ according to the labels in H. The probability that the hash values realize H is $1/m^{d(t+1)} \leq 1/\left((1+\varepsilon)\,(d-1)\,n\right)^{d(t+1)}$.

We calculate

$$\begin{aligned}\mathrm{E}^*\left(N_S^{\mathsf{MCOG}}\right) &\leq \sum_{t=1}^{n} \frac{d^6 \cdot t^2 \cdot (1+\varepsilon)^{(d-1)\cdot(t+1)-1} \cdot (d-1)^{d\cdot(t+1)-4} \cdot n^{d\cdot(t+1)-1}}{\left((1+\varepsilon)\,(d-1)\,n\right)^{d(t+1)}} \\ &\leq \frac{d^6}{(d-1)^4 n} \cdot \sum_{t+1}^{n} \frac{t^2}{(1+\varepsilon)^{t-1}} = O\left(\frac{1}{n}\right).\end{aligned}$$

□

Bounding $\Pr\left(B_S^{\mathsf{MCOG}}\right)$. We will now prove the following lemma.

Lemma 14.1.4
Let $S \subseteq U$ with $|S| = n, d \geq 3$, and $\varepsilon > 0$ be given. Set $m \geq (1+\varepsilon)(d-1)n$. Let $\ell, c \geq 1$. Choose $\vec{h} \in \mathcal{Z}_{\ell,m}^{c,d}$ at random. Then

$$\Pr\left(B_S^{\mathsf{MCOG}}\right) = O\left(\frac{n}{\ell^c}\right).$$

To use our framework from Section 11, we have to find a suitable peelable hypergraph property that contains MCOG. Since minimal complex obstruction hypergraphs are path-like, we relax their notion in the following way.

Definition 14.1.5
Let P^* be the set of all hypergraphs H from $\mathcal{G}_{m,n}^d$ which have one of the following properties:

1. H has property MCOG.
2. H is a strict path.
3. H consists of a strict path $e_1, \ldots, e_t, t \geq 1$, and an edge f such that $|f \cap (e_1 \cup e_t)| \geq 1$ and
$$\left| f \cap \bigcup_{i=1}^{t} e_i \right| \geq 2.$$

Note that property 3 is somewhat artificial to deal with the case that a single edge of a minimal complex obstruction hypergraph of Type 2 is removed. Obviously, MCOG is contained in P^* and P^* is peelable. We can now prove Lemma 14.1.4.

Proof of Lemma 14.1.4. We apply Lemma 11.3.4 which says that

$$\Pr\left(B_S^{\mathsf{P}^*}\right) \leq \frac{1}{\ell^c}\sum_{t=1}^{n} t^{2c}\mu_t^{\mathsf{P}^*}.$$

We start by counting fully labeled hypergraphs $G \in \mathsf{P}^*$ having exactly $t+1$ edges. For the hypergraphs having Property 1 of Definition 14.1.5, we may use the bounds (14.2) and (14.3) on $w_1(d,t+1)$ and $w_2(d,t+1)$ in the proof of Lemma 14.1.3. Let $w_0(d,t+1)$ be the number of such hypergraphs which are strict paths, i. e., have Property 2 of Definition 14.1.5. We obtain the following bound:

$$w_0(d,t+1) \leq dm^d \cdot \left((d-1)\cdot m^{d-1}\right)^t.$$

Let $w_3(d,t+1)$ be the number of hypergraphs having Property 3 of Definition 14.1.5. We observe that

$$w_3(d,t+1) \leq dm^d \cdot \left((d-1)\cdot m^{d-1}\right)^{t-1} \cdot 2d^2 \cdot t \cdot m^{d-2}.$$

So, the number of fully labeled hypergraphs having exactly $t+1$ edges is at most

$$\begin{aligned}
&\left(\binom{n}{t+1}\cdot(t+1)!\right)\cdot\left(w_0(d,t+1)+w_1(d,t+1)+w_2(d,t+1)+w_3(d,t+1)\right)\\
&\leq n^{t+1}\cdot dm^d\cdot\left((d-1)\cdot m^{d-1}\right)^{t-2}\cdot\\
&\quad\left(d^2m^{2(d-1)}+d^4tm^{2(d-2)}+d^4t^2m^{2(d-2)}+2d^3tm^{2d-3}\right)\\
&\leq 4\cdot d^5\cdot t^2\cdot n^{t+1}\cdot m^{(d-1)(t+1)+1}\cdot(d-1)^{t-2}\\
&\leq 4\cdot d^5\cdot t^2\cdot n^{d(t+1)+1}\cdot(1+\varepsilon)^{(d-1)(t+1)+1}\cdot(d-1)^{d(t+1)-2}.
\end{aligned}$$

We may thus calculate:

$$\begin{aligned}
\Pr\left(B_S^{\mathsf{P}^*}\right) &\leq \frac{1}{\ell^c}\sum_{t=1}^{n} t^{2c}\cdot\frac{4\cdot d^5\cdot t^2\cdot(1+\varepsilon)^{(d-1)(t+1)+1}\cdot(d-1)^{d(t+1)-2}\cdot n^{d(t+1)+1}}{((1+\varepsilon)(d-1)n)^{d(t+1)}}\\
&\leq \frac{n}{\ell^c}\sum_{t=1}^{n}\frac{4\cdot d^5\cdot t^{2+2c}}{(d-1)^2\cdot(1+\varepsilon)^{t-1}} = O\left(\frac{n}{\ell^c}\right).
\end{aligned}$$

□

Putting Everything Together. Substituting the results of Lemma 14.1.3 and Lemma 14.1.4 into (14.1) yields

$$\Pr\left(N_S^{\mathsf{MCOG}} > 0\right) \leq O\left(\frac{1}{n}\right) + O\left(\frac{n}{\ell^c}\right).$$

Theorem 14.1.1 follows by setting $c \geq 2/\delta$ and $\ell = n^\delta$.

Remarks and Discussion. Our result shows that with hash functions from $\mathcal{Z}$ there exists w.h.p. an assignment of the keys to memory cells (according to the cuckoo hashing rules) when each table has size at least $(1+\varepsilon)(d-1)n$, where n is the size of the key set. Thus, the load of the hash table is smaller than $1/(d(d-1))$. In the fully random case, the load of the hash table rapidly grows towards 1, see, e. g., the table on Page 5 of [Die+10]. For example, using 5 hash functions allows the hash table load to be ≈ 0.9924. The approach followed in this section cannot yield such bounds for the following reason. When we look back at the proof of Lemma 14.1.2, we notice that it gives a stronger result: It shows that when a graph does not contain a complex component, it has an empty two-core, i. e., it does not contain a non-empty subgraph in which each vertex has minimum degree 2. It is known from random hypergraph theory that the appearance of a non-empty two-core becomes increasingly likely for d getting larger.[3] So, we cannot rely on hypergraphs with empty two-cores to prove bounds for generalized cuckoo hashing that improve for increasing values of d.

However, one can also approach orientation thresholds by means of matchings in random hypergraphs. It is well known from Hall's theorem that a bipartite graph $G(L, R, E)$ has an L-perfect matching if and only if every set of vertices $X \subseteq L$ has at least $|X|$ adjacent vertices in R [Die05]. (An L-perfect matching is a matching $M \subseteq E$ that contains every vertex of L.) Now, think of the hypergraph $G = G(S, \vec{h})$ as a bipartite graph $\mathrm{bi}(G) = (L, R, E)$ in the standard way. An L-perfect matching M induces a 1-orientation of G in the obvious way.

The statement of Hall's theorem in hypergraph terminology is: *There exists a* 1*-orientation of* $G(S, \vec{h})$ *if and only if for every* $T \subseteq S$ *the graph* $G(T, \vec{h})$ *(disregarding isolated vertices) has at least as many vertices as it has edges.* For the analysis, let D ("dense") contain all connected hypergraphs from $\mathcal{G}_{m,n}^d$ which have more edges than vertices. Then $G(S, \vec{h})$ contains a 1-orientation if and only if $N_S^{\mathsf{D}} = 0$. As before, we apply Lemma 11.2.2 and obtain the bound

$$\Pr\left(N_S^{\mathsf{D}} > 0\right) \leq \Pr\left(B_S^{\mathsf{D}}\right) + \mathrm{E}^*\left(N_S^{\mathsf{D}}\right). \tag{14.4}$$

The calculations from the proof in [Fot+05, Lemma 1] show that for $d \geq 2(1+\varepsilon)\ln(e/\varepsilon)$ we have that $\mathrm{E}^*\left(N_S^{\mathsf{D}}\right) = O(n^{4-2d})$. To bound the failure term of hash class $\mathcal{Z}$, we first need to find a peelable graph property. For this, let C denote the set of all connected hypergraphs in $\mathcal{G}_{m,n}^d$.

[3] According to [MM09, p. 418] (see also [Maj+96]) for large d the 2-core of a random d-uniform hypergraph with m vertices and n edges is empty with high probability if m is bounded from below by $dn/\log d$.

Of course, $\mathsf{D} \subseteq \mathsf{C}$, and C is peelable. However, it remains an open problem to find good enough bounds on $\Pr\left(B_S^{\mathsf{C}}\right)$. Section 14.3 will contain some progress towards achieving this goal. There we will prove that when considering connected hypergraphs with at most $O(\log n)$ edges, we can bound the failure term of $\mathcal{Z}$ by $O(n^{-\alpha})$ for an arbitrary constant $\alpha > 0$ which depends on d and the parameters of the hash class. However, the successful analysis of generalized cuckoo hashing requires this result to hold for connected hypergraphs with more edges. The situation is much easier in the sparse setting, since we can compensate for the additional label choices of a d-partite hypergraph with the $d-1$ factor in the number of vertices, see, e. g., the calculations in the proof of Lemma 14.1.4.

Another approach is the one used by Chazelle, Kilian, Rubinfeld, and Tal in [Cha+04, Section 4]. They proved that a 1-orientation of the hypergraph $G(S, \vec{h})$ can be obtained by the simple peeling process, if for each set $T \subseteq S$ the graph $G(T, \vec{h})$ (disregarding isolated vertices) has at most $d/2$ times more edges than it has vertices. This is weaker than the requirement of Hall's theorem, but we were still not able to prove low enough failure terms for hash class $\mathcal{Z}$.

14.2. Labeling-based Insertion Algorithms For Generalized Cuckoo Hashing

In the previous section we showed that when the tables are large enough, the hash functions allow storing S according to the cuckoo hashing rules with high probability. In this section we prove that such an assignment can be obtained (with high probability) with hash functions from $\mathcal{Z}$ using two recently described insertion algorithms.

Background. In the last section, we pointed out two natural insertion strategies for generalized cuckoo hashing: breadth-first search and random walk, described in [Fot+05]. Very recently, Khosla [Kho13] (2013) and Eppstein *et al.* [Epp+14] (2014) gave two new insertion strategies, which will be described next. In both algorithms, each table cell i in table T_j has a label (or counter) $l(j,i) \in \mathbb{N}$, where initially $l(j,i) = 0$ for all $j \in \{1,\dots,d\}$ and $i \in \{0,\dots,m-1\}$. The insertion of a key x works as follows: Both strategies find the table index

$$j = \operatorname*{arg\,min}_{j \in \{1,\dots,d\}} \{l(j, h_j(x))\}.$$

If $T_j[h_j(x)]$ is free then x is stored in this cell and the insertion terminates successfully. Otherwise, let y be the key which resides in $T_j[h_j(x)]$. Store x in $T_j[h_j(x)]$. The difference between the two algorithms is how they adjust the labeling. The algorithm of Khosla sets

$$l(j, h_j(x)) \leftarrow \min\{l(j', h_{j'}(x)) \mid j' \in (\{1,\dots,d\} \setminus \{j\})\} + 1,$$

while the algorithm of Eppstein *et al.* sets $l(j, h_j(x)) \leftarrow l(j, h_j(x)) + 1$. Now insert y in the same way. This is iterated until an empty cell is found or it is noticed that the insertion cannot be

performed successfully.[4] In Khosla's algorithm, the content of the label $l(j,i)$ is a lower bound for the minimal length of an eviction sequences that allows to store a new element into $T_j[i]$ (moving other elements around)[Kho13, Proposition 1]. In the algorithm of Eppstein *et al.*, the label $l(j,i)$ contains the number of times the memory cell $T_j[i]$ has been overwritten. According to [Epp+14], it aims to minimize the number of write operations to a memory cell. This so-called "wear" of memory cells is an important issue in modern flash memory. In our analysis, we show that in the sparse setting with $m \geq (1+\varepsilon)(d-1)n$, the maximum label in the algorithm of Eppstein *et al.* is $\log\log n + O(1)$ with high probability and the maximum label in the algorithm of Khosla is $O(\log n)$ with high probability.

Result. Our result when using hash functions from $\mathcal{Z}$ is as follows. We only study the case that we want to insert the keys from a set S sequentially without deletions.

Theorem 14.2.1
Let $\varepsilon > 0, 0 < \delta < 1$, $d \geq 3$ be given. Assume $c \geq 2/\delta$. For $n \geq 1$ consider $m \geq (1+\varepsilon)(d-1)n$ and $\ell = n^\delta$. Let $S \subseteq U$ with $|S| = n$. Choose $\vec{h} \in \mathcal{Z}_{\ell,m}^{c,d}$ at random. Insert all keys from S in an arbitrary order using the algorithm of Khosla using $\vec{h}$. Then with probability $O(1/n)$ (i) all key insertions are successful and (ii) $\max\{l(j,i) \mid i \in \{0,\dots,m-1\}, j \in \{1,\dots,d\}\} = O(\log n)$.

Theorem 14.2.2
Let $\varepsilon > 0, 0 < \delta < 1$, $d \geq 3$ be given. Assume $c \geq 2/\delta$. For $n \geq 1$ consider $m \geq (1+\varepsilon)(d-1)n$ and $\ell = n^\delta$. Let $S \subseteq U$ with $|S| = n$. Choose $\vec{h} \in \mathcal{Z}_{\ell,m}^{c,d}$ at random. Insert all keys from S in an arbitrary order using the algorithm of Eppstein *et al.* using $\vec{h}$. Then with probability $O(1/n)$ (i) all key insertions are successful and (ii) $\max\{l(j,i) \mid i \in \{0,\dots,m-1\}, j \in \{1,\dots,d\}\} = \log\log n + O(1)$.

For the analysis of both algorithms we assume that the insertion of an element fails if there exists a label of size $n+1$. (In this case, new hash functions are chosen and the data structure is built anew.) Hence, to prove Theorem 14.2.1 and Theorem 14.2.2 it suffices to show that statement (ii) holds. (An unsuccessful insertion yields a label with value $> n$.)

Analysis of Khosla's Algorithm. We first analyze the algorithm of Khosla. We remark that in our setting, Khosla's algorithm finds an assignment with high probability. (In [Kho13, Section 2.1] Khosla gives an easy argument why her algorithm always finds an assignment when this

[4] Neither in [Epp+14] nor in [Kho13] it is described how this should be done in the cuckoo hashing setting. From the analysis presented there, when deletions are forbidden, one should do the following: Both algorithms have a counter MaxLabel, and if there exists a label $l(j,i) \geq$ MaxLabel, then one should choose new hash functions and re-insert all items. For Khosla's algorithm, MaxLabel $= \Theta(\log n)$; for the algorithm of Eppstein *et al.*, one should set MaxLabel $= \Theta(\log\log n)$.

is possible. In the previous section, we showed that such an assignment exists with probability $1 - O(1/n)$.) It remains to prove that the maximum label has size $O(\log n)$. We first introduce the notation used by Khosla in [Kho13]. Recall the definition of the cuckoo allocation graph from the beginning of Section 14.1. Let G be a cuckoo allocation graph. Let $F_G \subseteq V$ consist of all free vertices in G. Let $d_G(u, v)$ be the distance between u and v in G. Define

$$d_G(u, F) := \min\left(\{d_G(u, v) \mid v \in F\} \cup \{\infty\}\right).$$

Now assume that the key set S is inserted in an arbitrary order. Khosla defines a *move* as every action that writes an element into a table cell. (So, the i-th insertion is decomposed into $k_i \geq 1$ moves.) The allocation graph at the end of the p-th move is denoted by $G_p = (V, E_p)$. Let M denote the number of moves necessary to insert S. (Recall that we assume that $\vec{h}$ is suitable for S.) Khosla shows the following connection between labels and distances to a free vertex.

Proposition 14.2.3 ***[Kho13, Proposition 1]***
For each $p \in \{0, 1, \ldots, M\}$ and each $v \in V$ it holds that

$$d_{G_p}(v, F_{G_p}) \geq l(j, i), \tag{14.5}$$

where $T_j[i]$ is the table cell that corresponds to vertex v.

Now fix an integer $L \geq 1$. Assume that there exists an integer p, for $0 \leq p \leq M$, and a vertex v such that $d(v, F_{G_p}) = L$. Let $(v = v_0, v_1, \ldots, v_{L-1}, v_L)$ be a simple path p of length L in G_p such that v_L is free. Let $x_0, \ldots, x_{L-1} \subseteq S$ be the keys which occupy $v_0, \ldots, v_{L-1}$. Then the hypergraph $G(S, \vec{h})$ contains a subgraph H that corresponds to p in the obvious way.

Definition 14.2.4
For given integers $L \geq 1, m \geq 1, n \geq 1, d \geq 3$, let SP^L ("simple path") consist of all hypergraphs $H = (V, \{e_1, \ldots, e_L\})$ in $\mathcal{G}^d_{m,n}$ with the following properties:

1. For all $i \in \{1, \ldots, L\}$ we have that $|e_i| = 2$. (So, H is a graph.)
2. For all $i \in \{1, \ldots, L-1\}$, $|e_i \cap e_{i+1}| = 1$.
3. For all $i \in \{1, \ldots, L-2\}$ and $j \in \{i+2, \ldots, L\}$, $|e_i \cap e_j| = 0$.

Our goal in the following is to show that there exists a constant c such that for all $L \geq c \log n$ we have $\Pr\left(N_S^{\mathrm{SP}^L} > 0\right) = O(1/n)$. From Lemma 11.2.2 we obtain the bound

$$\Pr\left(N_S^{\mathrm{SP}^L} > 0\right) \leq \mathrm{E}^*\left(N_S^{\mathrm{SP}^L}\right) + \Pr\left(B_S^{\mathrm{SP}^L}\right). \tag{14.6}$$

Bounding $\mathrm{E}^*\left(N_S^{\mathsf{SP}^L}\right)$. We show the following lemma.

Lemma 14.2.5
Let $S \subseteq U$ with $|S| = n, d \geq 3$, and $\varepsilon > 0$ be given. Consider $m \geq (d-1)(1+\varepsilon)n$. Then

$$\mathrm{E}^*\left(N_S^{\mathsf{SP}^L}\right) \leq \frac{md}{(1+\varepsilon)^L}.$$

Proof. We count fully labeled hypergraphs with property SP^L. Let P be an unlabeled simple path of length L. There are $d \cdot (d-1)^L$ ways to label the vertices on P with $\{1, \ldots, d\}$ to fix the class of the partition they belong to. Then there are not more than m^{L+1} ways to label the vertices with labels from $[m]$. There are fewer than n^L ways to label the edges with labels from $\{1, \ldots, n\}$. Fix such a fully labeled path P'. Now draw $2L$ hash values from $[m]$ according to the labels of P'. The probability that these random choices realize P' is $1/m^{2L}$. We calculate:

$$\mathrm{E}^*\left(N_S^{\mathsf{SP}^L}\right) \leq \frac{d \cdot (d-1)^L \cdot m^{L+1} \cdot n^L}{m^{2L}} = \frac{m \cdot d \cdot (d-1)^L}{\left((d-1)\,(1+\varepsilon)\right)^L} = \frac{md}{(1+\varepsilon)^L}.$$

□

Bounding $\Pr\left(B_S^{\mathsf{SP}^L}\right)$. Note that SP^L is not peelable. We relax SP^L in the obvious way and define $\mathsf{RSP}^L = \bigcup_{0 \leq i \leq L} \mathsf{SP}^L$. Graph property RSP^L is peelable.

Lemma 14.2.6
Let $S \subseteq U$ with $|S| = n$ and $d \geq 3$ be given. For an $\varepsilon > 0$, set $m \geq (1+\varepsilon)(d-1)n$. Let $\ell, c \geq 1$. Choose $\vec{h} \in \mathcal{Z}_{\ell,m}^{c,d}$ at random. Then

$$\Pr\left(B_S^{\mathsf{SP}^L}\right) = O\left(\frac{n}{\ell^c}\right).$$

Proof. Since $\mathsf{SP}^L \subseteq \mathsf{RSP}^L$ and RSP^L is peelable, we may apply Lemma 11.3.4 and obtain the bound

$$\Pr\left(B_S^{\mathsf{SP}^L}\right) \leq \frac{1}{\ell^c} \cdot \sum_{t=1}^{n} t^{2c} \cdot \mu_t^{\mathsf{RSP}^L}.$$

By the definition of RSP^L and using the same counting argument as in the proof of Lemma 14.2.5,

we calculate:

$$\Pr\left(B_S^{\mathrm{SP}^L}\right) \leq \frac{1}{\ell^c} \cdot \sum_{t=1}^{n} t^{2c} \cdot \frac{md}{(1+\varepsilon)^t} = O\left(\frac{n}{\ell^c}\right).$$

□

Putting Everything Together. Plugging the results of Lemma 14.2.5 and Lemma 14.2.6 into (14.6) shows that

$$\Pr\left(N_S^{\mathrm{SP}^L} > 0\right) \leq \frac{md}{(1+\varepsilon)^L} + O\left(\frac{n}{\ell^c}\right).$$

Setting $L = 2\log_{1+\varepsilon}(n), \ell = n^\delta$, and $c \geq 2/\delta$ finishes the proof of Theorem 14.2.1.

Analysis of the Algorithm of Eppstein et al. We now analyze the algorithm of Eppstein *et al.* [Epp+14]. We use the *witness tree technique* to prove Theorem 14.2.2. This proof technique was introduced by Meyer auf der Heide, Scheideler, and Stemann [HSS96] in the context of shared memory simulations, and is one of the main techniques to analyze load balancing processes, see, e. g., [Col+98a; Col+98b; Ste96; SS00; Vöc03], which will be the topic of the next section.

Central to our analysis is the notion of a *witness tree for wear* k, for an integer $k \geq 1$. (Recall that in the algorithm of Eppstein *et al.*, the label $l(j,i)$ denotes the number of times the algorithm has put a key into the cell $T_j[i]$. This is also called the *wear* of the table cell.) For given values n and m, a witness tree for wear k is a $(d-1)$-ary tree with $k+1$ levels in which each non-leaf node is labeled with a tuple (j,i,κ), for $1 \leq j \leq d, 0 \leq i \leq m-1$, and $1 \leq \kappa \leq n$, and each leaf is labeled with a tuple (j,i), $1 \leq j \leq d$ and $0 \leq i \leq m-1$. Two children of a non-leaf node v must have different first components (j-values) and, if they exist, third components (κ-values) Also the κ-values of a node and its children must differ.

We say that a witness tree is *proper* if no two different non-leaf nodes have the same labeling. We say that a witness tree T can be *embedded into* $G(S,\vec{h})$ if for each non-leaf node v with label (j_0,i_0,κ) with children labeled $(j_1,i_1),\ldots,(j_{d-1},i_{d-1})$ in the first two label components in T, $h_{j_k}(x_\kappa) = i_k$, for each $0 \leq k \leq d-1$. We can think of a proper witness tree as an edge-labeled hypergraph from $\mathcal{G}_{m,n}^d$ by building from each non-leaf node labeled (j_0,i_0,κ) together with its $d-1$ children with label components $(j_1,i_1),\ldots,(j_{d-1},i_{d-1})$ a hyperedge $(i'_0,\ldots,i'_{d-1})$ labeled "κ", where $i'_0,\ldots,i'_{d-1}$ are ordered according to the j-values.

Suppose that there exists a label $l(j,i)$ with content k for an integer $k > 0$. We now argue about what must have happened that $l(j,i)$ has such a label. In parallel, we construct the witness tree for wear k. Let T be an unlabeled $(d-1)$-ary tree with $k+1$ levels. Let y be the key residing in $T_j[i]$. Label the root of T with (j,i,κ), where $y = x_\kappa \in S$. Then for all other choices of y in tables $T_{j'}, j' \in \{1,\ldots,d\}, j' \neq j$, we have $l(j',h_{j'}(y)) \geq k-1$. (When y was written into $T_j[i]$, $l(j,i)$ was $k-1$ and this was minimal among all choices of key y. Labels are never

decreased.) Let $x_1, \ldots, x_{d-1}$ be the keys in these $d-1$ other choices of y. Label the children of the root of T with the $d-1$ tuples $(j', h_{j'}(y)), 1 \leq j' \leq d, j' \neq j$, and the respective key indices. Arguing in the same way as above, we see that for each key $x_i, i \in \{1, \ldots, d-1\}$, its $d-1$ other table choices must have had a label of at least $k-2$. Label the children of the node corresponding to key x_i on the second level of T with the $d-1$ other choices, for each $i \in \{1, \ldots, d-1\}$. (Note that already the third level may include nodes with the same label.) Proceeding with this construction on the levels $3, \ldots, k$ gives the witness tree T for wear k. By construction, this witness tree can be embedded into $G(S, \vec{h})$.

So, all we have to do to prove Theorem 14.2.2 is to obtain a (good enough) bound on the probability that a witness tree for wear k can be embedded into $G(S, \vec{h})$. If a witness tree is not proper, it seems difficult to calculate the probability that this tree can be embedded into $G(S, \vec{h})$, because different parts of the witness tree correspond to the same key in S, which yields dependencies among hash values. However, we know from the last section that when $G(S, \vec{h})$ is sparse enough, i. e., $m \geq (1+\varepsilon)(d-1)n$, it contains only hypertrees and unicyclic components with probability $1 - O(1/n)$. Using a basic pruning argument, Eppstein *et al.* show that this simplifies the situation in the following way.

Lemma 14.2.7 ***[Epp+14, Observation 2 & Observation 3]***
Let H be a hypergraph that consists of only hypertrees and unicyclic components. Suppose H contains an embedded witness tree for wear k. Then there exists a proper witness tree for wear $k-1$ that can be embedded into H.

Proof. Let T be a witness tree for wear k that can be embedded into a unicyclic component of H. (If it is embedded into a hypertree, there is nothing to prove.) Observe that T can have at most one label that occurs at more than one non-leaf node, because the paths from the root to a non-leaf node correspond to paths in $G(S, \vec{h})$. So, two different labels that occur both more than once in non-leaf nodes certify that the component would be complex. Now, traverse T using a breadth-first search and let v be the first non-leaf node that has a label which occurred before. Observe that all other occurrences of that labeling are in nodes at the subtree rooted at v. (Otherwise, the component would be complex.) Let p be the unique path from this node to the root of T. Then removing the child of the root that lies on p (and the whole subtree rooted at that child) makes T have distinct non-leaf labels. The tree rooted at each of its remaining children is a witness tree for wear $k-1$ and there exists at least one such child. □

Let $\mathcal{E}_{S,k}$ be the event that there exists a witness tree for wear k that can be embedded into $G(S, \vec{h})$. To prove Theorem 14.2.2, we have to show that for the parameter choices in the Theorem

$$\Pr(\mathcal{E}_{S,k}) = O(1/n).$$

We separate the cases whether $G(S, \vec{h})$ contains a complex component or not. Let PWT_k be the set of all hypergraphs in $G^d_{m,n}$ which correspond to proper witness trees for wear k. Using

Theorem 14.1.1, we may bound:

$$\begin{aligned}\Pr\left(\mathcal{E}_{S,k}\right) &\leq \Pr\left(N_S^{\mathsf{PWT}_{k-1}} > 0\right) + \Pr\left(N_S^{\mathsf{MCOG}} > 0\right) \\ &\leq \Pr\left(B_S^{\mathsf{PWT}_{k-1}}\right) + \mathrm{E}^*\left(N_S^{\mathsf{PWT}_{k-1}}\right) + \Pr\left(N_S^{\mathsf{MCOG}} > 0\right). \end{aligned} \tag{14.7}$$

The last summand on the right-hand side of this inequality is handled by Theorem 14.1.1, so we may concentrate on the graph property PWT_{k-1}.

Bounding $\mathrm{E}^*\left(N_S^{\mathsf{PWT}_{k-1}}\right)$. We start by proving that the expected number of proper witness trees in $G(S, \vec{h})$ is $O(1/n)$ for the parameter choices in Theorem 14.2.2. We use a different proof method than Eppstein *et al.* [Epp+14], because we cannot use the statement of [Epp+14, Lemma 1]. We remark here that the following analysis could be extended to obtain bounds of $O(1/n^s)$, for $s \geq 1$. However, this does not help to obtain better bounds, because the last summand of (14.7) is known to be only $O(1/n)$.

Lemma 14.2.8
Let $S \subseteq U$ with $|S| = n$ and $d \geq 3$ be given. For an $\varepsilon > 0$, set $m \geq (1+\varepsilon)(d-1)n$. Then there exists a value $k = \log\log n + \Theta(1)$ such that

$$\mathrm{E}^*\left(N_S^{\mathsf{PWT}_{k-1}}\right) = O\left(\frac{1}{n}\right).$$

Proof. We first obtain a bound on the number of proper witness trees for wear $k-1$. Let T be an unlabeled $(d-1)$-ary tree with k levels. The number v_{k-1} of vertices of such a tree is

$$v_{k-1} = \sum_{i=0}^{k-1}(d-1)^i = \frac{(d-1)^k - 1}{d-2}.$$

For the number e_{k-1} of non-leaf nodes of such a tree, we have

$$e_{k-1} = \sum_{i=0}^{k-2}(d-1)^i = \frac{(d-1)^{k-1} - 1}{d-2}.$$

There are $n \cdot d \cdot m$ ways to label the root of T. There are not more than $n^{d-1} \cdot m^{d-1}$ ways to label the second level of the tree. Labeling the remaining levels in the same way, we see that in total there are fewer than

$$n^{e_{k-1}} \cdot d \cdot m^{v_{k-1}}$$

proper witness trees for wear $k-1$. Fix such a fully labeled witness tree T. Now draw $d \cdot e_{k-1} = v_{k-1} + e_{k-1} - 1$ values randomly from $[m]$ according to the labeling of the nodes in T. The probability that these values realize T is exactly $1/m^{v_{k-1}+e_{k-1}-1}$. We obtain the following bound:

$$\begin{aligned} \mathrm{E}^* \left(N_S^{\mathsf{PWT}_k} \right) &\leq \frac{n^{e_{k-1}} \cdot d \cdot ((1+\varepsilon)\,(d-1)n)^{v_{k-1}}}{((1+\varepsilon)\,(d-1)n)^{v_{k-1}+e_{k-1}-1}} = \frac{n \cdot d}{((1+\varepsilon)(d-1))^{e_{k-1}-1}} \\ &\leq \frac{n \cdot d}{((1+\varepsilon)(d-1))^{(d-1)^{k-2}}}, \end{aligned}$$

which is in $O(1/n)$ for $k = \log\log n + \Theta(1)$. □

Bounding $\Pr\left(B_S^{\mathsf{PTW}_{k-1}}\right)$. We first relax the notion of a witness tree in the following way.

Definition 14.2.9
Let RWT^{k-1} (relaxed witness trees) be the set of all hypergraphs which can be obtained in the following way:

1. Let $T \in \mathsf{PWT}^{k'}$ be an arbitrary proper witness tree for wear k', $k' \leq k-1$. Let ℓ denote the number of nodes on level $k'-1$, i. e., the level prior to the leaf level of T.
2. Arbitrarily choose $\ell' \in \mathbb{N}$ with $\ell' \leq \ell - 1$.
3. Choose $\kappa = \lfloor \ell'/(d-1) \rfloor$ arbitrary distinct non-leaf nodes on level $k'-2$. For each such node, remove all its children together with their $d-1$ children from T. Then remove from a group of $d-1$ siblings on level $k'-1$ the $\ell' - (d-1)\cdot\kappa$ siblings with the largest j-values together with their leaves.

Note that RWT^{k-1} is a peelable graph property, for we can iteratively remove non-leaf nodes that correspond to edges in the hypergraph until the whole leaf level is removed. Removing these nodes as described in the third property makes sure that there exists at most one non-leaf node at level $k'-2$ that has fewer than $d-1$ children. Also, it is clear what the first components in the labeling of the children of this node are. Removing nodes in a more arbitrary fashion would give more labeling choices and thus more trees with property RWT^{k-1}.

Lemma 14.2.10
Let $S \subseteq U$ with $|S| = n$ and $d \geq 3$ be given. For an $\varepsilon > 0$, set $m \geq (1+\varepsilon)(d-1)n$. Let $\ell, c \geq 1$. Choose $\vec{h} \in \mathcal{Z}_{\ell,m}^{c,d}$ at random. Then

$$\Pr\left(B_S^{\mathsf{RWT}^{k-1}}\right) = O\left(\frac{n}{\ell^c}\right).$$

Proof. We apply Lemma 11.3.4, which says that

$$\Pr\left(B_S^{\mathsf{RWT}^{k-1}}\right) \leq \frac{1}{\ell^c} \cdot \sum_{t=2}^{n} t^{2c} \mu_t^{\mathsf{RWT}^{k-1}}.$$

Using the same line of argument as in the bound for $\mathrm{E}^*\left(N_S^{\mathsf{PWT}_k}\right)$, the expected number of witness trees with property RWT^{k-1} with exactly t edges, i. e., exactly t non-leaf nodes, is at most $n \cdot d \cdot / \left((1+\varepsilon)(d-1)\right)^{t-1}$. We calculate:

$$\Pr\left(B_S^{\mathsf{RWT}^{k-1}}\right) = \frac{1}{\ell^c} \sum_{t=1}^{n} t^{2c} \frac{n \cdot d}{\left((1+\varepsilon)(d-1)\right)^{t-1}} = O\left(\frac{n}{\ell^c}\right).$$

□

Putting Everything Together. Using the results from Lemma 14.2.8, Lemma 14.2.10, and Theorem 14.1.1, we conclude that

$$\begin{aligned}\Pr\left(\mathcal{E}_{S,k}\right) &\leq \Pr\left(B_S^{\mathsf{PWT}_k}\right) + \mathrm{E}^*\left(N_S^{\mathsf{PWT}_k}\right) + \Pr\left(N_S^{\mathsf{MCOG}} > 0\right) \\ &\leq O(1/n) + O(n/\ell^c).\end{aligned}$$

Theorem 14.2.2 follows for $\ell = n^\delta$ and $c \geq 2/\delta$.

Remarks and Discussion. In [Kho13], Khosla shows that her algorithm finds an assignment whenever $\vec{h}$ is suitable for the key set S. That means that her algorithm works up to the thresholds for generalized cuckoo hashing. Our result is much weaker and provides only guarantees for a load factor of $1/(d(d-1))$. Moreover, she proves that n insertions take time $O(n)$ with high probability. This is the first result of this kind for insertion procedures for generalized cuckoo hashing. (For the random walk method, this result was conjectured in [FPS13]; for the BFS method, n insertions take time $O(n)$ in expectation.) We did not check if n insertions take time $O(n)$ w.h.p. when hash functions from $\mathcal{Z}$ are used instead of fully random hash functions.

With respect to the algorithm of Eppstein *et al.*, our result shows that n insertions take time $O(n \log\log n)$ with high probability when using hash functions from $\mathcal{Z}$. With an analogous argument to the one given by Khosla in [Kho13], the algorithm of Eppstein *et al.* of course finds an assignment of the keys whenever this is possible. However, the bound of $O(\log\log n)$ on the maximum label is only known for $m \geq (1+\varepsilon)(d-1)n$ and $d \geq 3$, even in the fully random case. Extending the analysis on the maximum label size to more dense hypergraphs is an open question. Furthermore, finding a better bound than $O(n \log\log n)$ (w.h.p.) on the insertion time for n elements is open, too.

14.3. Load Balancing

In this section we apply hash class $\mathcal{Z}$ in the area of load balancing schemes. In the discussion at the end of this section, we will present a link of our results w.r.t. load balancing to the space utilization of generalized cuckoo hashing in which each memory cell can hold $\kappa \geq 1$ items.

Background. In randomized load balancing we want to allocate a set of jobs J to a set of machines M such that a condition, e. g., there exists no machine with "high" load, is satisfied with high probability. To be consistent with the notation used in our framework and previous applications, S will denote the set of jobs, and the machines will be numbered $1, \ldots, m$. In this section we assume $|S| = n = m$, i. e., we allocate n jobs to n machines.

We use the following approach to load balancing: For an integer $d \geq 2$, we split the n machines into groups of size n/d each. For simplicity, we assume that d divides n. Now a job chooses d candidate machines by choosing exactly one machine from each group. This can be modeled by using d hash functions $h_1, \ldots, h_d$ with $h_i \colon S \to [n/d], 1 \leq i \leq d$, such that machine $h_i(j)$ is the candidate machine in group i of job j.

In load balancing schemes, the arrival of jobs has been split into two models: parallel and sequential arrival. We will focus on parallel job arrivals and come back to the sequential case at the end of this section.

In the parallel arrival model, all jobs arrive in parallel, i. e., at the same time. They communicate with the machines in synchronous rounds. In these rounds, decisions on the allocations of jobs to machines are made. The τ-collision protocol is one algorithm to find such an assignment. This protocol was studied in the context of distributed memory machines by Dietzfelbinger and Meyer auf der Heide [DM93]. In the context of load balancing, the allocation algorithm was analyzed by Stemann in [Ste96]. The τ-collision protocol works in the following way: First, each job chooses one candidate machine from each of the $d \geq 2$ groups. Then the following steps are repeated until all jobs are assigned to machines:

1. Synchronously and in parallel, each unassigned job sends an allocation request to each of its candidate machines.

2. Synchronously and in parallel, each machine sends an acknowledgement to all requesting jobs if and only if it got at most τ allocation requests in this round. Otherwise, it does not react.

3. Each job that gets an acknowledgement is assigned to one of the machines that has sent an acknowledgement. Ties are broken arbitrarily.

Note that the number of rounds is not bounded. However, we will show that w.h.p. the τ-collision protocol will terminate after a small number of rounds.

There exist several analysis techniques for load balancing, e. g., layered induction, fluid limit models and witness trees [Raj+01]. We will focus on the witness tree technique to analyze load

balancing schemes. We use the variant studied by Schickinger and Steger in [SS00] in connection with hash class $\mathcal{Z}$. The main contribution of [SS00] is to provide a unified analysis for several load balancing algorithms. That allows us to show that hash class $\mathcal{Z}$ is suitable in all of these situations as well, with only little additional work.

The center of the analysis in [SS00] is the so-called *allocation graph*. In our setting, where each job chooses exactly one candidate machine in each of the d groups, the allocation graph is a bipartite graph $G = ([n], [n], E)$, where the jobs are on the left side of the bipartition, and the machines are on the right side, split into groups of size n/d. Each job vertex is adjacent to its d candidate machines. As already discussed in Section 14.1, the allocation graph is equivalent to the hypergraph $G(S, \vec{h})$. Recall that we refer to the bipartite representation of a hypergraph $G = (V, E)$ with $\mathrm{bi}(V, E)$. We call the vertices on the left side *job vertices* and the vertices on the right side *machine vertices*.

If a machine has high load we can find a subgraph in the allocation graph that shows the chain of events in the allocation process that led to this situation, hence "witnessing" the high load of this machine. (Similarly to the wear of a table cell in the algorithm of Eppstein *et al.* in the previous section.) Such witness trees might differ greatly in structure, depending on the load balancing scheme.

In short, the approach of Schickinger and Steger works as follows.[5]

1. They show that high load leads to the existence of a "witness graph" and describe the properties of such a graph for a given load balancing scheme.

2. For their analysis to succeed they demand that the witness graph from above is a tree in the technical sense. They show that with high probability a witness graph can be turned into a cycle-free witness tree by removing a small number of edges at the root.

3. For such a "real" tree (the "witness tree"), they show that it is unlikely to exist in the allocation graph.

We will give a detailed description of this approach after stating the main result of this section.

Result. The following theorem represents one selected result from [SS00], replacing the full randomness assumption with hash functions from $\mathcal{Z}$ to choose candidate machines for jobs. We simplify the theorem by omitting the exact parameter choices calculated in [SS00]. All the other examples considered in [SS00] can be analyzed in an analogous way, resulting in corresponding theorems. We discuss this claim further in the discussion part of this section.

[5] This approach has much in common with our analysis of insertion algorithms for generalized cuckoo hashing. However, the analysis will be much more complicated here, since the hypergraph $G\left(S, \vec{h}\right)$ has exactly as many vertices as edges.

Theorem 14.3.1
For each constant $\alpha > 0, d \geq 2$, there exist constants $\beta, c > 0$ (depending on α and d), such that for each t with $2 \leq t \leq 1/\beta \ln \ln n, \ell = n^{1/2}$ and $\vec{h} = (h_1, \ldots, h_d) \in \mathcal{Z}_{\ell,n}^{c,d}$, the τ-collision protocol described above with threshold $\tau = O\left(\frac{1}{d-1}((\ln n)/(\ln \ln n))^{1/(t-2)}\right)$ finishes after t rounds with probability at least $1 - O(n^{-\alpha})$.

We remark that Woelfel showed in [Woe06a] that a variant of hash class $\mathcal{Z}$ works well in the case of randomized load balancing as modeled and analyzed by Voecking in [Vöc03], even in a dynamic version where balls may be removed and re-inserted again. Although showing a slightly weaker result, we will use a much wider and more generic approach to the analysis of randomized load balancing schemes and expand the application of hash functions from $\mathcal{Z}$ to other load balancing schemes. Moreover, we hope that our analysis of the failure term of $\mathcal{Z}$ on certain hypergraphs occurring in the analysis will be of independent interest.

We will now analyze the τ-collision protocol using hash functions from class $\mathcal{Z}$. Most importantly, we have to describe the probability of the event that the τ-collision protocol does not terminate after t rounds in the form of a graph property. To achieve this, we start by describing the structure of witness trees.

In the setting of the τ-collision protocol in parallel arrival, a witness tree has the following structure. Using the notation of [SS00], a machine is *active in round* t if there exists at least one job that sends a request to this machine in round t. If no such job exists, the machine is *inactive in round* t. Assume that after round t the collision protocol has not yet terminated. Then there exists a machine y that is active in round t and that received more than τ allocation requests. Arbitrarily choose τ of these requests. These requests were sent by τ unallocated jobs in round t. The vertex that corresponds to machine y is the root of the witness tree, the τ job vertices are its children. In round t, each of the τ unallocated jobs sent allocation requests to $d-1$ other machines. The corresponding machine vertices are the children of each of the τ job vertices in the witness tree. By definition, these machines are also active in round t, and so they were active in round $t-1$ as well. So, there are $\tau \cdot (d-1)$ machines that are active in round $t-1$. We must be aware that among these machines the same machine might appear more than once, because unallocated jobs may have chosen the same candidate machine. So, there may exist vertices in the witness tree that correspond to the same machine. For all these $\tau \cdot (d-1)$ machines the same argument holds in round $t-1$. Proceeding with the construction for rounds $t-2, t-3, \ldots, 1$, we build the *witness tree* T_t *with root* y. It exhibits a regular recursive structure, depicted abstractly in Figure 14.1. Note that all leaves correspond to machine vertices, since no allocation requests are sent in round 0.

As we have seen, such regular witness trees do not need to be subgraphs of the allocation graph since two vertices of a witness tree might be embedded to the same vertex. Hence, the witness tree is "folded together" to a subgraph in the allocation graph. In the embedding of a witness tree as a subgraph of the allocation graph, edges do not occur independently and the analysis becomes difficult, even in the fully random case.

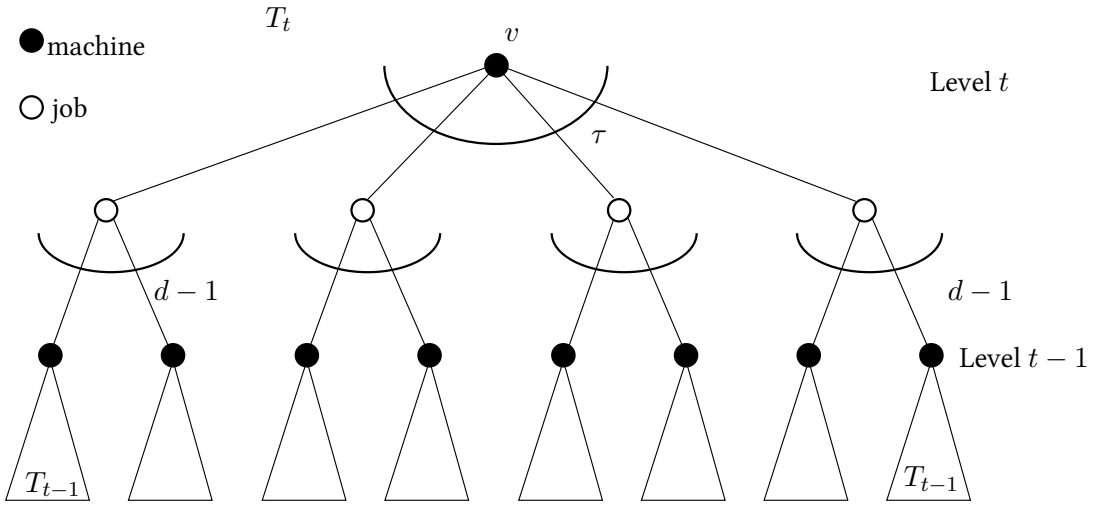

Figure 14.1.: Structure of a witness tree T_t with root v after t rounds if the τ-collision protocol with d candidate machines has not yet terminated.

Schickinger and Steger found the following way to analyze this situation. They introduced the notion of a *multicycle* that describes an "almost tree-like" graph.

Definition 14.3.2
Let $k, t \geq 1$. Let $G = (V, E)$ be an undirected graph. Let $s \in V$ and let $d(v)$ denote the distance between s and v in G, for each $v \in V$. A (k, t)-*multicycle of depth at most* t *at node* s *in* G is a connected subgraph G' of G together with a spanning tree T' of G' with the following properties:

1. G' includes vertex s.
2. G' has cyclomatic number k (*cf.* Section 12).
3. For each vertex v in T', the distance between s and v in T' is $d(v)$.
4. All vertices v in T' have $d(v) \leq t$.
5. Each leaf in T' is incident to an edge in G' that is not in T'.

Multicycles will be used to reason in the following way: When G does not contain a certain (k, t)-multicycle at node s as a subgraph, removing only a few edges in G incident to node s make the neighborhood that includes all vertices of distance at most t of s in the remaining graph acyclic. As we shall see, the proof that this is possible will be quite easy when using a spanning tree that consists only of shortest paths from s to the other vertices. (Such a tree can be obtained by starting a breadth-first search in G from s.)

One easily checks that a (k,t)-multicycle M with m vertices and n edges satisfies $n = m + k - 1$, because the cyclomatic number of a connected graph is exactly $n - m + 1$ [Die05]. Furthermore, it has at most $2kt$ vertices, because there can be at most $2k$ leaves that each have distance at most t from s, and all vertices of the spanning tree lie on the unique paths from s to the leaves. We will later see that for the parameters given in Theorem 14.3.1, a (k,t)-multicycle is with high probability not a subgraph of the allocation graph.

Lemma 14.3.3 ***[SS00, Lemma 2]***
Let $k, t \geq 1$. Assume that a graph $G = (V, E)$ contains no (k', t)-multicycle, for $k' > k$. Furthermore, consider the maximal induced subgraph $H = (V', E')$ of G for a vertex $v \in V$ that contains all vertices $w \in V$ that have distance at most t from v in G. Then we can remove at most $2k$ edges incident to v in H to get a graph H^* such that the connected component of v in H^* is a tree.

Proof. We follow the proof in [SS00]. Start a breadth-first search from v in H and let C be the set of cross edges, i. e., non-tree edges with respect to the bfs tree. Let H' be the subgraph of H that contains the edges from C and the (unique) paths from v to each endpoint of the edges from C. H' is a $(|C|, t)$-multicycle at node v. From the assumption we know that $|C| \leq k$. Let H^* be the subgraph of H that results from removing all edges in H' from H. At most $2k$ edges incident to v are removed in this way. By construction, the connected component that contains v in H^* is acyclic. □

In the light of this lemma, we set $\tau \geq 2k + 1$ and know that if the allocation graph contains a witness tree after t rounds, then it contains a (k,t)-multicycle or a regular witness tree T_{t-1}. This observation motivates us to consider the following graph property:

Definition 14.3.4
Let $k, t \in \mathbb{N}$. Then $\mathsf{MCWT}^{k,t} \subseteq \mathcal{G}^d_{n/d,n}$ is the set of all hypergraphs H such that $\mathrm{bi}(H)$ forms either a (k,t)-multicycle or a witness tree T_{t-1}.

If we use hash class $\mathcal{Z}$ and set $\tau \geq 2k + 1$, for a set S of jobs we have:

$$\Pr(\text{the } \tau\text{-collision protocol does not terminate after } t \text{ rounds}) \leq \Pr\left(N_S^{\mathsf{MCWT}^{k,t}} > 0\right). \quad (14.8)$$

By Lemma 11.2.2 we may bound the probability on the right-hand side of (14.8) by

$$\Pr\left(N_S^{\mathsf{MCWT}^{k,t}} > 0\right) \leq \Pr\left(B_S^{\mathsf{MCWT}^{k,t}}\right) + \mathrm{E}^*\left(N_S^{\mathsf{MCWT}^{k,t}}\right). \quad (14.9)$$

Bounding $\mathrm{E}^*(N_S^{\mathsf{MCWT}^{k,t}})$ We first consider the fully random case. The following lemma is equivalent to Theorem 1 in [SS00]. However, our parameter choices are slightly different because in [SS00] each of the d candidate machines is chosen from the set $[n]$, while in our setting we split the n machines into d groups of size n/d.

Lemma 14.3.5
Let $\alpha \geq 1$ and $d \geq 2$. Set $\beta = 2d(\alpha + 2\ln d + 3/2)$ and $k = \alpha + 2$. Consider t with $2 \leq t \leq (1/\beta)\ln\ln n$. Let

$$\tau = \max\left\{\frac{1}{d-1}\left(\frac{\beta t \ln n}{\ln\ln n}\right)^{\frac{1}{t-2}}, d^{d+1}e^d + 1, 2k+1\right\}.$$

Then

$$\mathrm{E}^*\left(N_S^{\mathsf{MCWT}^{k,t}}\right) = O(n^{-\alpha}).$$

Proof. For the sake of the analysis, we consider $\mathsf{MCWT}^{k,t}$ to be the union of two graph properties $\mathsf{MC}^{k,t}$, hypergraphs that form (k,t)-multicycles, and WT^{t-1}, hypergraphs that form witness trees for the parameter $t-1$. We show the lemma by proving $\mathrm{E}^*\left(N_S^{\mathsf{MC}^{k,t}}\right) = O(n^{-\alpha})$ and $\mathrm{E}^*\left(N_S^{\mathsf{WT}^{t-1}}\right) = O(n^{-\alpha})$. In both cases, we consider the bipartite representation of hypergraphs. Our proofs follow [SS00, Section 4].

We start by bounding $\mathrm{E}^*\left(N_S^{\mathsf{MC}^{k,t}}\right)$. As we have seen, a (k,t)-multicycle is a connected graph that has at most $2kt$ vertices and cyclomatic number k. We start by counting (k,t)-multicycles with exactly s vertices and $s+k-1$ edges, for $s \leq 2kt$. In this case, we have to choose j and u (the number of jobs and machines, resp.) such that $s = j+u$. By Lemma 12.1.2 there are at most $(s+k-1)^{O(k)}$ unlabeled (k,t)-multicycles. Fix such an unlabeled (k,t)-multicycle G. There are two ways to split the vertices of G into the two sides of the bipartition. (G can be assumed to be bipartite since we consider (k,t)-multicycles that are subgraphs of the allocation graph.) Once this bipartition is fixed, we have n^j ways to choose the job vertices and label vertices of G with these jobs. There are d^{s+k-1} ways to label the edges of G with labels from $1,\ldots,d$, which represent the request modeled by an edge between a job vertex and a machine vertex. Once this labeling is fixed, there are $(n/d)^u$ ways to choose machine vertices and label the remaining vertices of G. Fix such a fully labeled graph G'.

For each request r of a job j, choose at random and independently a machine from $[n/d]$. The probability that this machine is the same machine that j had chosen in G' is (d/n). Thus, the probability that G' is realized by the random choices is $(d/n)^{s+k-1}$. By setting $k = \alpha + 2$ and

using the parameter choice $t = O(\ln \ln n)$ we calculate

$$\begin{aligned}
\mathrm{E}^*\left(N_S^{\mathsf{MC}^{k,t}}\right) &\leq \sum_{s=1}^{2kt} \sum_{u+j=s} 2 \cdot n^j \cdot (n/d)^u \cdot d^{s+k-1} \cdot (s+k-1)^{O(k)} \cdot (d/n)^{s+k-1} \\
&\leq n^{1-k} \sum_{s=1}^{2kt} 2s \cdot d^{2(s+k-1)} \cdot (s+k-1)^{O(1)} \\
&\leq n^{1-k} \cdot 2kt \cdot 4kt \cdot d^{2(2kt+k-1)} \cdot (2kt+k-1)^{O(1)} \\
&\leq n^{1-k} \cdot (\ln \ln n)^{O(1)} \cdot (\ln n)^{O(1)} = O(n^{2-k}) = O(n^{-\alpha}).
\end{aligned}$$

Now we consider $\mathrm{E}^*\left(N_S^{\mathsf{WT}^{t-1}}\right)$. By the simple recursive structure of witness trees, a witness tree of depth $t-1$ has $j = \frac{\tau^{t-1}(d-1)^{(t-2)}-\tau}{\tau(d-1)-1}$ job vertices and $u = \frac{\tau^{t-1}(d-1)^{t-1}-1}{\tau(d-1)-1}$ machine vertices. Let T be an unlabeled witness tree of depth $t-1$. T has $r = d \cdot j$ edges. There are not more than n^j ways to choose j jobs from S and label the job vertices of T. There are not more than d^r ways to label the edges with a label from $\{1, \ldots, d\}$. Once this labeling is fixed, there are not more than $(n/d)^u$ ways to choose the machines and label the machine vertices in the witness tree. With these rough estimates, we over-counted the number of witness trees by at least a factor of $(1/\tau!)^{j/\tau} \cdot (1/(d-1)!)^j$. (See Figure 14.1. For each job vertex, there are $(d-1)!$ labelings which result in the same witness tree. Furthermore, for each non-leaf machine vertex, there are $\tau!$ many labelings which yield the same witness tree.) Fix such a fully labeled witness tree T'.

For each request of a job j choose at random a machine from $[n/d]$. The probability that the edge matches the edge in T' is d/n. Thus, the probability that T' is realized by the random choices is $(d/n)^r$. We calculate

$$\begin{aligned}
\mathrm{E}^*\left(N_S^{\mathsf{WT}^{t-1}}\right) &\leq n^j \cdot d^r \cdot (n/d)^u \cdot \left(\frac{1}{\tau!}\right)^{j/\tau} \cdot \left(\frac{1}{(d-1)!}\right)^j \cdot (d/n)^r \\
&\leq n \cdot d^{2r} \cdot \left(\frac{1}{\tau!}\right)^{j/\tau} \cdot \left(\frac{1}{(d-1)!}\right)^j \leq n \left[\frac{e}{\tau} \cdot \left(\frac{e}{d-1}\right)^{d-1} \cdot d^{2d}\right]^j \\
&\leq n \left(\frac{e^d \cdot d^{d+1}}{\tau}\right)^j.
\end{aligned}$$

Observe that

$$j = \frac{\tau^{t-1}(d-1)^{t-2}-1}{\tau(d-1)-1} \geq \frac{\tau^{t-2}(d-1)^{t-2}-1}{d} \geq \frac{(\tau(d-1))^{t-2}}{2d}.$$

For the parameter settings provided in the lemma we get

$$\begin{aligned}
\mathrm{E}^*\left(N_S^{\mathsf{WT}^{t-1}}\right) &\leq n\left(\frac{e^d \cdot d^{d+1}}{\tau}\right)^{\frac{(\tau(d-1))^{t-2}}{2d}} = n\left(\frac{e^d \cdot d^{d+1}}{\tau}\right)^{\frac{\beta t \ln n}{2d \ln\ln n}} \\
&\leq n\left(e^d \cdot d^{d+2} \cdot \left(\frac{\beta \ln\ln n}{t \ln n}\right)^{\frac{1}{t-2}}\right)^{\frac{\beta t \ln n}{2d \ln\ln n}} \\
&\leq n \cdot \left(\left(e^d \cdot d^{d+2}\right)^t \left(\frac{\beta \ln\ln n}{t \ln n}\right)\right)^{\frac{\beta \ln n}{2d \ln\ln n}} \\
&\leq n \cdot \left(\left(e^d \cdot d^{d+2}\right)^{\frac{1}{\beta}\ln\ln n} \cdot \frac{1}{\ln n}\right)^{\frac{\beta \ln n}{2d \ln\ln n}} \\
&\leq n^{3/2+2\ln d-\beta/2d}.
\end{aligned}$$

Setting $\beta = 2d(\alpha + 2\ln d + 3/2)$ suffices to show that $\mathrm{E}^*(N_S^{\mathsf{MCWT}^{k,t}}) = O(n^{-\alpha})$. □

Bounding $\Pr(B_S^{\mathsf{MCWT}^{k,t}})$. To apply Lemma 11.3.4, we need a peelable graph property that contains $\mathsf{MCWT}^{k,t}$. We will first calculate the size of witness trees to see that they are small for the parameter settings given in Theorem 14.3.1.

The Size of Witness Trees. Let T_t be a witness tree after t rounds. Again, the number of job vertices j_t in T_t of the τ-collision protocol is given by

$$j_t = \frac{\tau^t(d-1)^{t-1} - \tau}{\tau(d-1) - 1}.$$

We bound the size of the witness tree when exact parameters are fixed as in Lemma 14.3.5.

Lemma 14.3.6
Let $\alpha > 0$, $d \geq 2$, $\beta = 2d(\alpha + 2\ln d + 3/2), k = \alpha + 2$, and $2 \leq t \leq \frac{1}{\beta}\ln\ln n$. Let $\tau = \max\left\{\frac{1}{d-1}\left(\frac{\beta t \ln n}{\ln\ln n}\right)^{\frac{1}{t-2}}, d^{d+1}e^d + 1, 2k+1\right\}$. Then $j_t < \log n$.

Proof. Observe the following upper bound for the number of jobs in a witness tree after t rounds

$$j_t = \frac{\tau^t(d-1)^{t-1} - \tau}{\tau(d-1) - 1} \leq \frac{\tau(\tau(d-1))^{t-1}}{2\tau - 1} \leq (\tau(d-1))^{t-1}.$$

Now observe that for a constant choice of the value τ it holds

$$(\tau(d-1))^{t-1} \leq (\tau(d-1))^{\frac{1}{\beta}\ln\ln n} \leq (\ln n)^{\frac{\ln\tau+\ln d}{\beta}} \leq \ln n,$$

since $\frac{\ln \tau}{\beta} \leq 1$ for the two constant parameter choices in Lemma 14.3.5. Furthermore, for the non-constant choice of τ it holds

$$((d-1)\tau)^{t-1} \leq \frac{\beta t \ln n}{\ln \ln n} \leq \ln n.$$

It follows $j_t \leq \ln n < \log n$. □

A (k,t)-multicycle has at most $2kt + k - 1$ edges. For $t = O(\log \log n)$ and a constant $k \geq 1$, such multicycles are hence smaller than witness trees.

A Peelable Graph Property. To apply Lemma 11.3.4, we have to find a peelable graph property that contains all subgraphs that have property $\mathsf{MCWT}^{k,t}$ (multicycles or witness trees for $t-1$ rounds). Since we know from above that witness trees and multicycles are contained in small connected subgraphs of the hypergraph $G(S, \vec{h})$, we will use the following graph property.

Definition 14.3.7
Let $K > 0$ and $d \geq 2$ be constants. Let $\mathsf{C}_{\mathit{small}}(K,d)$ contain all connected d-partite hypergraphs $(V,E) \in \mathcal{G}^d_{n/d,n}$ with $|E| \leq K \log n$ disregarding isolated vertices.

The following lemma shows how we can bound the failure term of $\mathcal{Z}$.

Lemma 14.3.8
Let $K > 0$, $c \geq 1$, $\ell \geq 1$, and $d \geq 2$ be constants. Let S be the set of jobs with $|S| = n$. Then

$$\Pr\left(B_S^{\mathsf{MCWT}^{k,t}}\right) \leq \Pr\left(B_S^{\mathsf{C}_{\mathit{small}}(K,d)}\right) = O\left(\frac{n^{K(d+1)\log d+2}}{\ell^c}\right).$$

For proving this bound on the failure probability, we need the following auxiliary graph property.

Definition 14.3.9
Let $K > 0, d \geq 2$, and $\ell \geq 1$ be constants. Let $n \geq 1$ be given. Then $\mathsf{HT}(K,d,\ell)$ ("hypertree") is the set of all d-partite hypergraphs $G = (V,E)$ in $\mathcal{G}^d_{n/d,n}$ with $|E| \leq K \log n$ for which $\mathrm{bi}(G)$ (disregarding isolated vertices) is a tree, has at most ℓ leaf edges and has leaves only on the left (job) side.

We will now establish the following connection between $\mathsf{C}_{\mathit{small}}(K,d)$ and $\mathsf{HT}(K,d,\ell)$.

Lemma 14.3.10
Let $K > 0, d \geq 2$ and $c \geq 1$ be constants. Then $\mathsf{C}_{\mathit{small}}(K,d)$ is weak $\mathsf{HT}(K,d,2c)$-$2c$-reducible, *cf.* Definition 11.3.7.

Proof. Assume $G = (V, E) \in \mathsf{C}_{small}(K, d)$. Arbitrarily choose $E^* \subseteq E$ with $|E^*| \leq 2c$. We have to show that there exists an edge set E' such that $(V, E') \in \mathsf{HT}(K, d, 2c)$, (V, E') is a subgraph of (V, E) and for each edge $e^* \in E^*$ there exists an edge $e' \in E'$ such that $e' \subseteq e^*$ and e' and e^* have the same label.

Identify an arbitrary spanning tree T in $\mathrm{bi}(G)$. Now repeatedly remove leaf vertices with their incident edges, as long as these leaf vertices do not correspond to edges from E^*. Denote the resulting tree by T'. In the hypergraph representation, T' satisfies all properties from above. □

From Lemma 11.3.4 we can use the bound

$$\Pr\left(B_S^{\mathsf{MCWT}^{k,t}}\right) \leq \Pr\left(B_S^{\mathsf{C}_{small}(K,d)}\right) \leq \ell^{-c} \sum_{t=2}^{n} t^{2c} \mu_t^{\mathsf{HT}(K,d,2c)}.$$

Lemma 14.3.11
Let $K > 0, d \geq 2$, and $c \geq 1$ be constants. If $t \leq K \cdot \log n$, then

$$\mu_t^{\mathsf{HT}(K,d,2c)} \leq t^{O(1)} \cdot n \cdot d^{(d+1)t}.$$

For $t > K \log n$ it holds that $\mu_t^{\mathsf{HT}(K,d,2c)} = 0$.

Proof. It is trivial that $\mu_t^{\mathsf{HT}(K,d,2c)} = 0$ for $t > K \log n$, since a hypergraph with more than $K \log n$ edges contains too many edges to have property $\mathsf{HT}(K, d, 2c)$.

Now suppose $t \leq K \log n$. We first count labeled hypergraphs having property $\mathsf{HT}(K, d, 2c)$ consisting of t job vertices and z edges, for some fixed $z \in \{t, \ldots, dt\}$, in the *bipartite representation*.

There are at most $z^{O(2c)} = z^{O(1)}$ unlabeled trees with z edges and at most $2c$ leaf edges (Lemma 12.1.2). Fix one such tree T. There are not more than n^t ways to label the job vertices of T, and there are not more than d^z ways to assign each edge a label from $\{1, \ldots, d\}$. Once these labels are fixed, there are not more than $(n/d)^{z+1-t}$ ways to assign the right vertices to machines. Fix such a fully labeled tree T'.

Now draw z hash values at random from $[n/d]$ and build a graph according to these hash values and the labels of T'. The probability that these random choices realize T' is exactly $1/(n/d)^z$. Thus we may estimate:

$$\begin{aligned}\mu_t^{\mathsf{HT}(K,d,2c)} &\leq \sum_{z=t}^{dt} \frac{(n/d)^{z+1-t} \cdot z^{O(1)} \cdot n^t \cdot d^z}{(n/d)^z} = \sum_{z=t}^{dt} z^{O(1)} \cdot n \cdot d^{z-1+t} \\ &< dt \cdot (dt)^{O(1)} \cdot n \cdot d^{(d+1)t} = t^{O(1)} \cdot n \cdot d^{(d+1)t}.\end{aligned}$$

□

We can proceed with the proof our main lemma.

Proof (of Lemma 14.3.8). By Lemma 11.3.4, we now know that

$$\Pr\left(B_S^{\mathsf{MCWT}^{k,t}}\right) \leq \ell^{-c} \sum_{t=2}^{n} t^{2c} \mu_t^{\mathsf{HT}(K,d,2c)}.$$

Applying the result of Lemma 14.3.11, we calculate

$$\begin{aligned}\Pr\left(B_S^{\mathsf{MCWT}^{k,t}}\right) &\leq \ell^{-c} \sum_{t=2}^{K\log n} t^{2c} t^{O(1)} \cdot n \cdot d^{(d+1)t} = n \cdot \ell^{-c} \cdot (K \log n)^{O(1)} \cdot d^{(d+1)K\log n} \\ &= O(n^2 \cdot \ell^{-c}) \cdot d^{K(d+1)\log n} = O(n^{K(d+1)\log d+2} \cdot \ell^{-c}).\end{aligned}$$

□

Putting Everything Together. Using the previous lemmas allows us to complete the proof of our main theorem.

Proof of Theorem 14.3.1. We plug the results of Lemma 14.3.5 and Lemma 14.3.8 into (14.9) and get

$$\Pr\left(N_S^{\mathsf{MCWT}^{k,t}} > 0\right) \leq O\left(\frac{n^{K(d+1)\log d+2}}{\ell^c}\right) + O\left(\frac{1}{n^\alpha}\right).$$

For the case of parallel arrival with $d \geq 2$ hash functions, we calculated that witness trees do not have more than $\log n$ edges (Lemma 14.3.6). So, we set $K = 1$. Setting $\ell = n^{1/2}$ and $c = 2(2 + \alpha + (d+1)\log d)$ finishes the proof of the theorem. □

Remarks and Discussion. We remark that the graph property $\mathsf{C}_{\mathit{small}}(K, d)$ provides a very general result on the failure probability of $\mathcal{Z}$ on hypergraphs $G(S, \vec{h})$. It can be applied for all results from [SS00]. We will exemplify this statement by discussing what needs to be done to show that $\mathcal{Z}$ works in the setting of Voecking's "Go-Left" sequential allocation algorithm [Vöc03]. By specifying explicitly how to break ties (always allocate the job to the "left-most" machine), Voecking's algorithm decreases the maximum bin load (w.h.p.) in sequential load balancing with $d \geq 2$ hash functions from $\ln\ln n / \ln d + O(1)$ (arbitrary tie-breaking) [Aza+99] to $\ln\ln n/(d \cdot \ln \Phi_d) + O(1)$, which is an exponential improvement in d. Here Φ_d is defined as follows. Let $F_d(j) = 0$ for $j \leq 0$ and $F_d(1) = 1$. For $j \geq 2$, $F_d(j) = \sum_{i=1}^{d} F_d(j-i)$. (This is a generalization of the Fibonacci numbers.) Then $\Phi_d = \lim_{j\to\infty} F_d(j)^{1/j}$. It holds that Φ_d is a constant with $1.61 \leq \Phi_d \leq 2$, see [Vöc03]. (We refer to [SS00, Section 5.2] and [Vöc03] for details about the description of the algorithm.) In the unified witness tree approach of Schickinger and Steger, the main difference between the analysis of parallel arrivals and the sequential algorithm of Voecking is in the definition of the witness tree. Here, the analysis in

[SS00] also assumes that the machines are split into d groups of size n/d. This means that we can just re-use their analysis in the fully random case. For bounding the failure term of hash class $\mathcal{Z}$, we have to show that the witness trees in the case of Voecking's "Go-Left" algorithm (see [SS00, Fig. 6]) have at most $O(\log n)$ jobs, i. e., that they are contained in small connected components. Otherwise, we cannot apply Lemma 14.3.8.

According to [SS00, Page 84], the number of job vertices j_ℓ in a witness tree for a bin with load ℓ is bounded by

$$j_\ell \leq 4h_\ell + 1, \tag{14.10}$$

where h_ℓ is the number of leaves in the witness tree. Following Voecking [Vöc03], Schickinger and Steger show that setting ℓ as large as $\ln\ln n/(d \ln \Phi_d) + O(1)$ is sufficient to bound the expected number of witness trees by $O(n^{-\alpha})$. Such witness trees have only $O(\log n)$ many job nodes.

Lemma 14.3.12
Let $\alpha > 0$ and $\ell = \log_{\Phi_d}(4 \log n^\alpha)/d$. Then $j_\ell \leq 33\alpha \log n$.

Proof. It holds $h_\ell = F_d(d \cdot \ell + 1)$, see [SS00, Page 84], and $F_d(d \cdot \ell + 1) \leq \Phi_d^{d\cdot\ell+1}$, since $F_d(j)^{1/j}$ is monotonically increasing. We obtain the bound

$$\begin{aligned} j_\ell &\leq 4 \cdot h_\ell + 1 \leq 4 \cdot \Phi_d^{d\cdot\ell+1} + 1 \leq 4 \cdot \Phi_d^{\log_{\Phi_d}(4\log n^\alpha)+1} + 1 \\ &= 16 \cdot \Phi_d \cdot \alpha \log n + 1 \leq 33\alpha \log n, \end{aligned}$$

using $\Phi_d \leq 2$ and assuming $\alpha \log n \geq 1$. □

Thus, we know that a witness tree in the setting of Voecking's algorithm is contained in a connected hypergraph with at most $33\alpha \log n$ edges. Thus, we may apply Lemma 14.3.8 in the same way as we did for parallel arrival. The result is that for given $\alpha > 0$ we can choose $(h_1, \ldots, h_d) \in \mathcal{Z}_{\ell,n}^{c,d}$ with $\ell = n^\delta, 0 < \delta < 1$, and $c \geq (33\alpha(d+1)\log d + 2 + \alpha)/\delta$ and know that the maximum load is $\ln\ln n/(d \cdot \ln \Phi_d) + O(1)$ with probability $1 - O(1/n^\alpha)$. So, our general analysis using small connected hypergraphs makes it very easy to show that hash class $\mathcal{Z}$ suffices to run a specific algorithm with load guarantees. However, the parameters for setting up a hash function are rather high when the constant in the big-Oh notation is large.

When we are interested in making the parameters for setting up a hash function as small as possible, one should take care when bounding the constants in the logarithmic bound on the number of edges in the connected hypergraphs. (According to [SS00], (14.10) can be improved by a more careful argumentation.) More promising is a direct approach to witness trees, as we did in the analysis of the algorithm of Eppstein *et al.* in the previous subsection, i. e., directly peeling the witness tree. Using such an approach, Woelfel showed in [Woe06a, Theorem 2.1 and its discussion] that smaller parameters for the hash functions $\mathcal{Z}$ are sufficient to run Voecking's algorithm.

$\kappa+1 \backslash d$	3	4	5	6	7	8
2	0.818	0.772	0.702	0.637	0.582	0.535
3	0.776	0.667	0.579	0.511	0.457	0.414
4	0.725	0.604	0.515	0.450	0.399	0.359
5	0.687	0.562	0.476	0.412	0.364	0.327
6	0.658	0.533	0.448	0.387	0.341	0.305

Table 14.1.: Space utilization thresholds for generalized cuckoo hashing with $d \geq 3$ hash functions and $\kappa+1$ keys per cell, for $\kappa \geq 1$, based on the non-existence of the $(\kappa+1)$-core. Each table cell gives the maximal space utilization achievable for the specific pair $(d, \kappa+1)$. These values have been obtained using Maple® to evaluate the formula from Theorem 1 of [Mol05].

We further remark that the analysis of the τ-collision protocol makes it possible to analyze the space utilization of generalized cuckoo hashing using $d \geq 2$ hash functions and buckets which hold up to $\kappa \geq 2$ keys in each table cell, as proposed by Dietzfelbinger and Weidling in [DW07]. Obviously, a suitable assignment of keys to table cells is equivalent to a κ-orientation of $G(S, \vec{h})$. It is well-known that any graph that has an empty $(\kappa+1)$-core, i. e., that has no subgraph in which all vertices have degree at least $\kappa+1$, has a κ-orientation, see, e. g., [DM09] and the references therein. (The converse, however, does not need to be the case.) The $(\kappa+1)$-core of a graph can be obtained by repeatedly removing vertices with degree at most κ and their incident hyperedges. The precise study of this process is due to Molloy [Mol05]. The τ-collision protocol is the parallel variant of this process, where in each round all vertices with degree at most τ are removed with their incident edges. (In the fully random case, properties of this process were recently studied by Jiang, Mitzenmacher, and Thaler in [JMT14].) In terms of orientability, Theorem 14.3.1 with the exact parameter choices from Lemma 14.3.5 shows that for $\tau = \max\{e^\beta, d^{d+1}e^d + 1, 2k+1\}$ there exists (w.h.p.) an assignment of the n keys to n memory cells when each cell can hold τ keys. (This is equivalent to a hash table load of $1/\tau$.) It is open to find good space bounds for generalized cuckoo hashing using this approach. However, we think that it suffers from the same general problem as the analysis for generalized cuckoo hashing with $d \geq 3$ hash functions and one key per table cell: Since the analysis builds upon a process which requires an empty $(\kappa+1)$-core in the hypergraph to succeed, space utilization seems to decrease for d and κ getting larger. Table 14.1 contains space utilization bounds for static generalized cuckoo hashing with $d \geq 3$ hash functions and κ elements per table cell when the assignment is obtained via a process that requires the $(\kappa+1)$-core to be empty. These calculations clearly support the conjecture that space utilization decreases for larger values of d and κ.

15. A Generalized Version of the Hash Class

In this section we will present a generalized version of our hash class that uses arbitrary κ-wise independent hash classes as building blocks.

15.1. The Generalized Hash Class

Definition 15.1.1
Let $c \geq 1, d \geq 2$, and $\kappa \geq 2$. For integers m, $\ell \geq 1$, and given $f_1, \ldots, f_d \colon U \to [m]$, $g_1, \ldots, g_c \colon U \to [\ell]$, and d two-dimensional tables $z^{(i)}[1..c, 0..\ell - 1]$ with elements from $[m]$ for $i \in \{1, \ldots, d\}$, we let $\vec{h} = (h_1, \ldots, h_d) = (h_1, \ldots, h_d)\langle f_1, \ldots, f_d, g_1, \ldots, g_c, z^{(1)}, \ldots, z^{(d)}\rangle$, where

$$h_i(x) = \Big(f_i(x) + \sum_{1 \leq j \leq c} z^{(i)}[j, g_j(x)]\Big) \bmod m, \text{for } x \in U, i \in \{1, \ldots, d\}.$$

Let $\mathcal{H}_m^\kappa$ [$\mathcal{H}_\ell^\kappa$] be an arbitrary κ-wise independent hash family with functions from U to $[m]$ [from U to $[\ell]$]. Then $\mathcal{Z}_{\ell,m}^{c,d,\kappa}(\mathcal{H}_\ell^\kappa, \mathcal{H}_m^\kappa)$ is the family of all sequences $(h_1, \ldots, h_d)\langle f_1, \ldots, f_d, g_1, \ldots, g_c, z^{(1)}, \ldots, z^{(d)}\rangle$ for $f_i \in \mathcal{H}_m^\kappa$ with $1 \leq i \leq d$ and $g_j \in \mathcal{H}_\ell^\kappa$ with $1 \leq j \leq c$.

In the following, we study $\mathcal{Z}_{\ell,m}^{c,d,2k}\left(\mathcal{H}_\ell^{2k}, \mathcal{H}_m^{2k}\right)$ for some fixed $k \in \mathbb{N}, k \geq 1$. For the parameters $d = 2$ and $c = 1$, this is the hash class used by Dietzfelbinger and Woelfel in [DW03]. We first analyze the properties of this hash class by stating a definition similar to Definition 11.1.2 and a lemma similar to Lemma 11.1.3. We hope that comparing the proofs of Lemma 11.1.3 and Lemma 15.1.3 shows the (relative) simplicity of the original analysis.

Definition 15.1.2
For $T \subseteq U$, define the random variable d_T, the "deficiency" of $\vec{h} = (h_1, \ldots, h_d)$ with respect to T, by $d_T(\vec{h}) = |T| - \max\{k, |g_1(T)|, \ldots, |g_c(T)|\}$. (Note: d_T depends only on the g_j-components of $(h_1, \ldots, h_d)$.) Further, define

(i) bad_T as the event that $d_T > k$;

(ii) good_T as $\overline{\mathit{bad}_T}$, i. e., the event that $d_T \leq k$;

(iii) crit_T as the event that $d_T = k$.

Hash function sequences $(h_1, \ldots, h_d)$ in these events are called *"T-bad"*, *"T-good"*, and *"T-critical"*, resp.

Lemma 15.1.3
Assume $d \geq 2, c \geq 1$, and $k \geq 1$. For $T \subseteq U$, the following holds:

(a) $\Pr(\mathit{bad}_T \cup \mathit{crit}_T) \leq \left(|T|^2/\ell\right)^{ck}$.

(b) Conditioned on good_T (or on crit_T), the hash values $(h_1(x), \ldots, h_d(x))$, $x \in T$, are distributed uniformly and independently in $[r]^d$.

Proof. (a) Assume $|T| \geq 2k$ (otherwise the events bad_T and crit_T cannot occur). Since $g_1, \ldots, g_c$ are independent, it suffices to show that for a function g chosen randomly from $\mathcal{H}_\ell^{2k}$ we have $\Pr(|T| - |g(T)| \geq k) \leq |T|^{2k}/\ell^k$.

We first argue that if $|T| - |g(T)| \geq k$ then there is a subset T' of T with $|T'| = 2k$ and $|g(T')| \leq k$. Initialize T' as T. Repeat the following as long as $|T'| > 2k$: (i) if there exists a key $x \in T'$ such that $g(x) \neq g(y)$ for all $y \in T' \setminus \{x\}$, remove x from T'; (ii) otherwise, remove any key. Clearly, this process terminates with $|T'| = 2k$. It also maintains the invariant $|T'| - |g(T')| \geq k$: In case (i) $|T'| - |g(T')|$ remains unchanged. In case (ii) before the key is removed from T' we have $|g(T')| \leq |T'|/2$ and thus $|T'| - |g(T')| \geq |T'|/2 > k$.

Now fix a subset T' of T of size $2k$ that satisfies $|g(T')| \leq k$. The preimages $g^{-1}(u)$, $u \in g(T')$, partition T' into k' classes, $k' \leq k$, such that g is constant on each class. Since g is chosen from a $2k$-wise independent class, the probability that g is constant on all classes of a given partition of T' into classes $C_1, \ldots, C_{k'}$, with $k' \leq k$, is exactly $\ell^{-(2k-k')} \leq \ell^{-k}$.

Finally, we bound $\Pr(|g(T)| \leq |T| - k)$. There are $\binom{|T|}{2k}$ subsets T' of T of size $2k$. Every partition of such a set T' into $k' \leq k$ classes can be represented by a permutation of T' with k' cycles, where each cycle contains the elements from one class. Hence, there are at most $(2k)!$ such partitions. This yields:

$$\Pr(|T| - |g(T)| \geq k) \leq \binom{|T|}{2k} \cdot (2k)! \cdot \frac{1}{\ell^k} \leq \frac{|T|^{2k}}{\ell^k}. \tag{15.1}$$

(b) If $|T| \leq 2k$, then h_1 and h_2 are fully random on T simply because f_1 and f_2 are $2k$-wise independent. So suppose $|T| > 2k$. Fix an arbitrary g-part of (h_1, h_2) so that good_T occurs, i.e., $\max\{k, |g_1(T)|, \ldots, |g_c(T)|\} \geq |T| - k$. Let $j_0 \in \{1, \ldots, c\}$ be such that $|g_{j_0}(T)| \geq |T| - k$. Arbitrarily fix all values in the tables $z_j^{(i)}$ with $j \neq j_0$ and $i \in \{1, 2\}$. Let T^* be the set of keys in T colliding with other keys in T under g_{j_0}. Then $|T^*| \leq 2k$. Choose the values $z_{j_0}^{(i)}[g_{j_0}(x)]$ for all $x \in T^*$ and $i \in \{1, 2\}$ at random. Furthermore, choose f_1 and f_2 at random from the $2k$-wise independent family $\mathcal{H}_r^{2k}$. This determines $h_1(x)$ and $h_2(x)$, $x \in T^*$, as fully random values. Furthermore, the function g_{j_0} maps the keys $x \in T - T^*$ to distinct entries of the vectors $z_{j_0}^{(i)}$ that were not fixed before. Thus, the hash function values $h_1(x), h_2(x)$, $x \in T - T^*$, are distributed fully randomly as well and are independent of those with $x \in T^*$. □

15.2. Application of the Hash Class

The central lemma to bound the impact of using our hash class in contrast to fully random hash functions was Lemma 11.3.4. One can reprove this lemma in an analogous way for the generalized version of the hash class, using the probability bound from Lemma 15.1.3(a) to get the following result.

Lemma 15.2.1
Let $c \geq 1, k \geq 1$, $S \subseteq U$ with $|S| = n$, and let A be a graph property. Let $\mathsf{B} \supseteq \mathsf{A}$ be a peelable graph property. Let C be a graph property such that B is C-$2ck$-reducible. Then

$$\Pr\left(B_S^{\mathsf{A}}\right) \leq \Pr\left(B_S^{\mathsf{B}}\right) \leq \ell^{-ck} \sum_{t=2k}^{n} t^{2ck} \cdot \mu_t^{\mathsf{C}}.$$

This settles the theoretical background needed to discuss this generalized hash class.

15.3. Discussion

One can now redo all the calculations from Section 13 and Section 14. We discuss the differences. Looking at Lemma 15.2.1, we notice the (t^{2ck})-factor in the sum instead of t^{2c}. Since k is fixed, this factor does not change anything in the calculations that always used $t^{O(1)}$ (see, e.g., the proof of Lemma 12.2.5). The factor $1/\ell^{ck}$ (instead of $1/\ell^c$) leads to lower values for c if $k \geq 2$. E.g., in cuckoo hashing with a stash, we have to set $c \geq (s+2)/(\delta k)$ instead of $c \geq (s+2)/\delta$. This improves the space usage, since we need less tables filled with random values. However, the higher degree of independence needed for the f- and g-components leads to a higher evaluation time of a single function. Hence there is an interesting tradeoff between space usage and evaluation time that we investigated in the following experiments.

16. Experimental Evaluation

In this section we will report on experiments running cuckoo hashing with a stash with our class of hash functions. We will compare it to other well-studied hash families, which will be introduced in Section 16.1. In that section, we will also describe the experimental setup. Subsequently, we will compare these hash functions with respect to the success probability (Section 16.2) and the running time (Section 16.3) for setting up the cuckoo hash table. We will also consider the cache behavior of the hash functions to speculate about running time differences on very large key sets.

16.1. Setup and Considered Hash Families

We start by introducing the setup of our experiments. We restrict all our experiments to hashing 32-bit keys.

Experimental Setup. We consider inputs of size n with $n \in \{2^{10}, \ldots, 2^{23}\}$. Inputs have two types: For general n, we assume that $S = \{1, \ldots, n\}$ and insert the elements of S in random order. For $n = 2^{20}$ we consider the following structured set:

$$\{x_0 + 2^8 \cdot x_1 + 2^{16} \cdot x_2 + 2^{24} \cdot x_3 \mid i \in \{0, 1, 2, 3\} : 0 \leq x_i \leq 31\}.$$

This set is known to be a worst-case input for the simple tabulation scheme that will be introduced below.

For each input of length n, we construct a cuckoo hash table of size $m \in \{1.005n, 1.05n\}$ for each of the two tables, i. e., we consider a load factor of 49.75% and 47.62%, respectively. We let the stash contain at most two keys, which yields a failure probability of $O(1/n^3)$. So, if a key were to be put in a stash that already contains two elements, we invoke a rehash. For each input length, we repeat the experiment 10,000 times, each time with a new seed for the random number generator. Next, we will discuss the considered hash families.

Simple Tabulation. In simple tabulation hashing as analyzed by Pătraşcu and Thorup [PT12], each key is a tuple $(x_1, \ldots, x_c)$ which is mapped to the hash value $\big(T_1(x_1) \oplus \cdots \oplus T_c(x_c)\big) \bmod m$ (where $\oplus$ denote bitwise XOR), by c uniform random hash functions (implemented by lookup tables filled with random values) $T_1, \ldots, T_c$, each with a domain of cardinality $\lceil |U|^{1/c} \rceil$. In our experiments with hashing 32-bit integers, we used two different versions of this scheme. The first

version views a key x to consist of four 8-bit keys x_1, x_2, x_3, x_4. Then, the tabulation scheme uses 4 tables $T_1, \ldots, T_4$ of size 2^8 filled with random 32-bit values. The hash function is then

$$h\left(x = (x_1, x_2, x_3, x_4)\right) = T_1[x_1] \oplus T_2[x_2] \oplus T_3[x_3] \oplus T_4[x_4] \mod m.$$

The second variant views a 32-bit key to consist of two 16-bit keys and uses two random tables of size 2^{16}.

Pătraşcu and Thorup showed in [PT12] that cuckoo hashing fails with probability $O(1/n^{1/3})$ using simple tabulation. According to Thorup (personal communication), a stash does not help to lower the failure probability of cuckoo hashing with simple tabulation hashing.

Polynomials with CW-Trick. Here we consider the standard implementation of an (approximately) k-independent family of hash functions: polynomials of degree $k-1$ over some prime field projected to the hash table range. For setting up such a polynomial, we choose a prime p much larger than $|U|$. (In the experiments, we used $p = 2^{48} - 1$.) Next, we choose k coefficients $a_0, \ldots, a_{k-1} \in [p]$. The hash function $h\colon U \to [m]$ is then defined by

$$h_{a_0,\ldots,a_{k-1}}(x) = \left(\left(\sum_{i=0}^{k-1} a_i x^i\right) \mod p\right) \mod m.$$

Evaluating this polynomial is done using Horner's method. In general, the modulo operation is expensive. When $p = 2^s - 1$ is a Mersenne prime, the "mod p" operation becomes simpler, because the result of $x \bmod p$ can be calculated as follows (this is the so-called "CW-trick" of Carter and Wegman suggested in [CW79]):

```
1: i ← x & p                          ▷ & is bit-wise and
2: i ← i + (x ≫ s)                    ▷ x ≫ s is a right-shift of x by s bits
3: if i ≥ p then
4:     return i − p
5: else
6:     return i
```

Murmur3. To compare hash functions used in practice with hash functions discussed in theory, we have picked one candidate used in practice: Murmur3. It is the third generation of a class of non-cryptographic hash functions invented by Austin Appleby with the goal of providing a fast and well-distributing hash function. A detailed overview over the algorithm can be found at [App]. To get two different hash functions, we use two different seeds when setting up a Murmur3-based hash function. Due to time constraints, we did not test other popular hash functions such as xxhash [Col], SipHash [AB12], or CityHash [PA].

Hash Family $\mathcal{Z}$. In view of the main results regarding our hash class, a hash function from $\mathcal{Z}_{\ell,m}^{c,2}$ guarantees the same failure probability in cuckoo hashing with a stash as a fully random hash function when we choose $\ell = n^\delta$ and $c \geq (s+2)/\delta$ (*cf.* Theorem 13.1.4). We have two main parameters:

1. The size $\ell = n^\delta$ of the random tables.
2. The number c of random tables per hash function and components g_j.

We aim at a failure probability of $O(1/n^3)$ and consequently use a stash size of 2. For $\ell = n^\delta$, the parameter c must then satisfy $c \geq 4/\delta$. As buildings blocks for our hash functions, we use the 2-universal "multiplication-shift" scheme from Dietzfelbinger *et al.* [Die+97] for hashing the 32-bit keys to ℓ_{out}-bit numbers, which reads for random odd $a \in [2^{32}]$:

$$h_a(x) = (ax \bmod 2^{32}) \operatorname{div} 2^{32-\ell_{\text{out}}}$$

In 32-bit arithmetic, this can be implemented as

$$h_a(x) = (ax) \gg (32 - \ell_{\text{out}}).$$

The 2-independent hash family that is used to initialize the f functions of a hash function from $\mathcal{Z}$, uses a variant of the "multiplication-shift" scheme from [Die96] for hashing 32-bit integers to ℓ_{out}-bit integers. There, we choose $a, b \in [2^{64}]$ and use the hash function

$$h_{a,b}(x) = \left((ax+b) \bmod 2^{64}\right) \operatorname{div} 2^{64-\ell_{\text{out}}}.$$

On a 64-bit architecture, we can implement this hashing scheme by

$$h_a(x) = (ax+b) \gg (64 - \ell_{\text{out}}).$$

We evaluate the following three constructions:

1. **Low Space Usage, Many Functions**. We let $\ell = n^{1/4}$ and must set $c = 16$. For $n = 2^{20}$ our hash function pair then consists of sixteen 2-universal functions $g_1, \ldots, g_{16}$, two 2-independent functions f_1, f_2 and thirtytwo tables of size 32 filled with random values. (Altogether, the tables can be stored in an integer array of size 1024.)
2. **Moderate Space Usage, Fewer Functions**. We let $\ell = n^{1/2}$ and get $c = 8$. For $n = 2^{20}$ our hash function pair then consists of eight 2-universal functions $g_1, \ldots, g_8$, two 2-independent functions f_1, f_2 and sixteen tables of size 2^{10} filled with random values. (Altogether, the tables can be stored in an integer array of size 16384.)
3. **Low Space Usage, High-Degree Polynomials**. In light of the generalization of our hash class discussed in Section 15, we also study the (extreme) case that for stash size s we use

Identifier	Construction
`simp-tab-8`	Simple Tabulation with four 8-bit keys from [PT12]
`simp-tab-16`	Simple Tabulation with two 16-bit keys from [PT12]
`Murmur3`	Murmur3 hash functions of [App]
`k-ind (3)`	3-wise independent hashing with polynomials of degree 2 with CW-Trick
$\mathcal{Z}_1$	$\mathcal{Z}$ construction "low space usage, many functions"
$\mathcal{Z}_2$	$\mathcal{Z}$ construction "moderate space usage, fewer functions"
$\mathcal{Z}_3$	$\mathcal{Z}$ construction "low space usage, high-degree polynomials"

Table 16.1.: Hash Families considered in our experiments. The "identifier" is used to refer to the constructions in the charts and in the text.

one g-function, two z tables of size $n^{1/2}$ and two f-functions. For $n = 2^{20}$, we use three 16-wise independent functions and two tables of size 2^{10}. (The tables can be stored in an integer array of size 2048.)

We observe that apart from the description lengths of these constructions, the difference in evaluation time is not clear. An array of size 1024 with 4 byte integers easily fits into L1 cache of the Intel i7 used in our experiments. This is not the case for an array of size 2^{14}. However, the second constructions needs only (roughly) half of the arithmetic operations. The third construction uses the fewest tables, but involves the evaluation of high-degree polynomials.

All constructions considered in the following are summarized with their identifiers in Table 16.1.

Our implementation uses C++. For compiling C++ code, we use *gcc* in version 4.8. Random values are obtained using `boost::random`.[1] The experiments were carried out on the machine specified in Section 1. The code can be found at `http://eiche.theoinf.tu-ilmenau.de/maumueller-diss/`.

16.2. Success Probability

We first report on our results for structured inputs. For $n = 2^{20}$ and $m = 1.05n$, rehashes occurred rarely. For all hash functions and all trials, a stash of size 1 would have sufficed. In 9 out of 10,000 runs, `simp-tab-16` used a stash of size 1. This was the maximum number of rehashes over all constructions. The results for $m = 1.005n$ looked very differently. Details can be found in Tab. 16.2. Without a stash the construction failed in about 8% of the cases. Using a stash of size 2 already decreased the likelyhood of a rehash to at most 0.74%. (With a stash of size 3 it would have been decreased to 0.21% of the cases.) We note that from our experiments

[1] `http://boost.org`

method	$s = 0$	$s = 1$	$s = 2$	rehash
`simp-tab-8`	9,223	590	131	54
`simp-tab-16`	9,194	595	139	72
`Murmur3`	9,232	576	139	53
`k-ind (3)`	9,137	632	159	72
$\mathcal{Z}_1$	9,127	656	159	58
$\mathcal{Z}_2$	9,201	595	150	54
$\mathcal{Z}_3$	9,177	615	134	74

Table 16.2.: Maximum stash size s for structured inputs of $n = 2^{20}$ elements with $m = 1.005n$.

we cannot report of a difference in the success probability between simple tabulation hashing and hash class $\mathcal{Z}$.

When considering key sets $\{1, \ldots, n\}$ for $n \in \{2^{10}, \ldots, 2^{23}\}$ for $m = 1.005n$, again there was no big difference between the different hash function constructions. As expected, the failure probability rapidly decreased for n getting larger. For $n = 2^{15}$, about 14% of the trials put a key into the stash and about 1.5% of the runs caused a rehash with a stash of size 2, for all constructions. For $n = 2^{19}$, these frequencies decreased to 10% and 0.8%, respectively. For $n = 2^{23}$, the frequencies were 6.5% and 0.7%, respectively.

16.3. Running Times

Table 16.3 shows the measurements of our running time experiments to construct a hash table with keys from $\{1, \ldots, n\}$ for dense tables with $m = 1.005n$. We observe that the simple tabulation scheme is the fastest implementation; it is faster than the deterministic `Murmur3`-based hash functions. Among the constructions based on hash class $\mathcal{Z}$, the second construction—moderate space usage and fewer hash functions—is faster than the other constructions. It is about a factor 1.8 slower than the fastest hash class, while providing theoretical guarantees on the failure probability comparable to a fully random hash function.[2] We also point out that there is a big difference in running time between `s-tab-8` and `s-tab-16`. This is because our inputs consisted of integers smaller than 2^{22}. In this situation, the ten most significant bits are unused. So, `s-tab-8` will always use $T_1[0]$ for the eight most significant bits. For comparison, we also include the results from the experiment on inputs from the hypercube $[32]^4$ in the last row of Table 16.3. In that case, construction `simp-tab-16` was a little bit faster than construction `simp-tab-8`. With respect to class $\mathcal{Z}$, it seems that from our experiments the parameter settings $\ell = \sqrt{n}$ (or the next power of 2 being at least as large as $\sqrt{n}$) and $c = 2(s + 2)$ provide the best performance. Using more hash functions but smaller tables is a little bit slower on our test

[2] When we aim for a failure probability of $O(1/\sqrt{n})$ with hash class $\mathcal{Z}$, we can use 6 tables of size $\sqrt{n}$, instead of 16 tables for construction $\mathcal{Z}_2$. This hash class was a factor of 1.26 slower than `simp-tab-8`, and was thus even faster than the 3-independent hash class.

n	`s-tab-8`	`s-tab-16`	`Murmur3`	`k-ind (3)`	$\mathcal{Z}_1$	$\mathcal{Z}_2$	$\mathcal{Z}_3$
1024	0.02 ms	0.03 ms	0.03 ms	0.04 ms	0.06 ms	0.04 ms	0.21 ms
4096	0.10 ms	0.10 ms	0.11 ms	0.14 ms	0.23 ms	0.17 ms	0.80 ms
16384	0.41 ms	0.48 ms	0.48 ms	0.55 ms	0.93 ms	0.66 ms	3.13 ms
65536	1.88 ms	2.41 ms	2.15 ms	2.57 ms	4.12 ms	3.09 ms	12.93 ms
262144	8.59 ms	10.62 ms	9.82 ms	12.05 ms	18.93 ms	14.75 ms	55.62 ms
1048576	39.84 ms	46.50 ms	45.10 ms	54.00 ms	80.85 ms	69.15 ms	214.36 ms
4194304	286.32 ms	293.20 ms	312.13 ms	375.00 ms	554.14 ms	510.35 ms	1081.58 ms
1048576	55.24 ms	54.67 ms	60.43 ms	75.46 ms	117.37 ms	101.02 ms	268.58 ms

Table 16.3.: Different running times for the construction of a cuckoo hash table for $m = 1.005n$. Each entry is the average over 10,000 trials. The last row contains the measurements for the structured input.

n	`s-tab-8`	`s-tab-16`	`Murmur3`	`k-ind (3)`	$\mathcal{Z}_1$	$\mathcal{Z}_2$	$\mathcal{Z}_3$
1024	0.02 ms	0.02 ms	0.03 ms	0.03 ms	0.06 ms	0.04 ms	0.19 ms
4096	0.09 ms	0.1 ms	0.11 ms	0.13 ms	0.22 ms	0.15 ms	0.74 ms
16384	0.38 ms	0.45 ms	0.44 ms	0.51 ms	0.87 ms	0.62 ms	2.93 ms
65536	1.75 ms	2.24 ms	2.0 ms	2.43 ms	3.91 ms	2.93 ms	12.12 ms
262144	8.08 ms	9.89 ms	9.19 ms	11.42 ms	18.16 ms	14.2 ms	52.57 ms
1048576	38.07 ms	43.95 ms	41.95 ms	51.64 ms	77.66 ms	65.07 ms	202.65 ms
4194304	265.03 ms	268.66 ms	280.21 ms	350.47 ms	531.16 ms	497.7 ms	1020.77 ms
1048576	52.26 ms	51.25 ms	57.04 ms	72.19 ms	113.8 ms	98.33 ms	254.60 ms

Table 16.4.: Different running times for the construction of a cuckoo hash table for $m = 1.05n$. Each entry is the average over 10,000 trials. The last row contains the measurements for the structured input type.

setup. Furthermore, using only one table but a high degree of independence is not competetive. Table 16.4 shows the results for $m = 1.05n$. The construction time is a little bit lower compared with the denser case. The relations with respect to running time stay the same.

We also measured the cache behavior of the hash functions. Figure 16.1 and Figure 16.2 show the measurements we got with respect to L1 cache misses and L2 cache misses, respectively. For reference to cache misses that happen because of access to the cuckoo hash table, we include the results for `k-ind (3)`. We see that among the tabulation-based constructions, `simp-tab-8` and $\mathcal{Z}_1$ show the best behavior with respect to cache misses. The data structure for the hash function is in L1 cache practically all the time. For `simp-tab-16`, we have about two additional cache misses per insertion. (Note that the four tables for `simp-tab-16` need 1 MB of data, and thus do not fit into L1 or L2 cache on our setup.) The variant $\mathcal{Z}_2$ of hash class $\mathcal{Z}$, using sixteen tables of size $\sqrt{n}$, performs badly when the key set is large. So, while $\mathcal{Z}_2$ is the fastest construction among the considered variants of hash class $\mathcal{Z}$ for $n \leq 2^{23}$, its performance should decrease for larger key sets. Then, construction $\mathcal{Z}_1$ with tables of size $n^{1/4}$ should be faster.

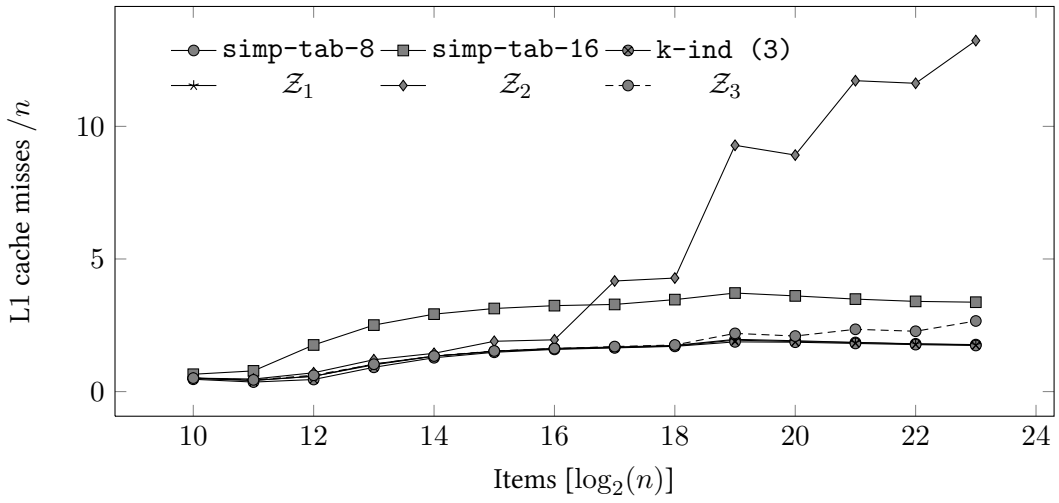

Figure 16.1.: Level 1 cache misses for building a cuckoo hash table from inputs $\{1, \ldots, n\}$ with a particular hash function. Each data point is the average over 10,000 trials. Cache Misses are scaled by n.

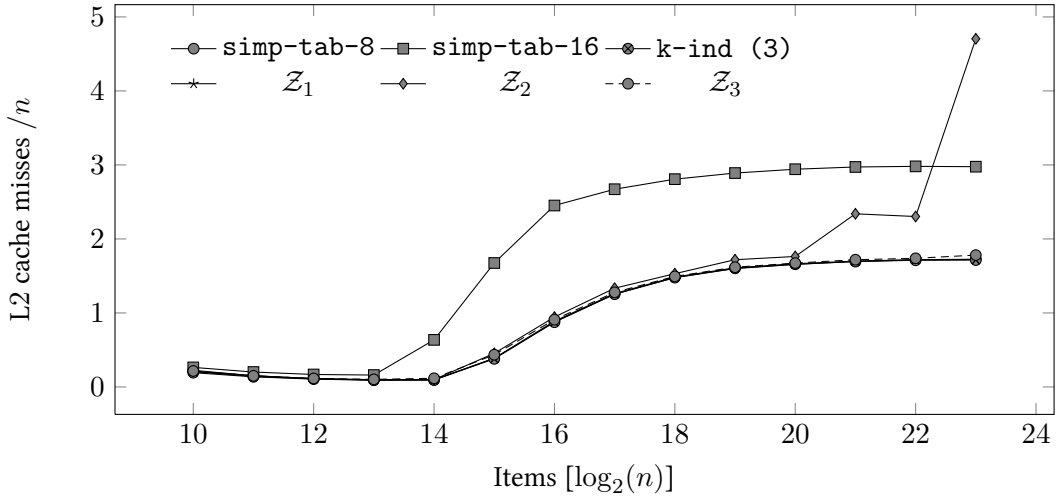

Figure 16.2.: Level 2 cache misses for building a cuckoo hash table from inputs $\{1, \ldots, n\}$ with a particular hash function. Each data point is the average over 10,000 trials. Cache Misses are scaled by n.

17. Conclusion and Open Questions

In this part of the thesis, we described a general framework to analyze hashing-based algorithms and data structures whose analysis depends on properties of the random graph $G(S, \vec{h})$ when $\vec{h}$ comes from a certain class of simple hash functions. This class combined lookups in small random tables with the evaluation of simple 2-universal or 2-independent hash functions.

We showed that these hash functions can be used in such diverse applications as cuckoo hashing (with a stash), generalized cuckoo hashing, the simulation of uniform hash functions, the construction of a perfect hash function, and load balancing. The framework allowed us to analyze these applications without exposing details of the hash family, only using a first moment approach for random graphs. Particular choices for the parameters to set up hash functions from $\mathcal{Z}$ provide hash functions that can be evaluated efficiently.

We already proposed directions for future work in the respective sections of this part of the thesis, and collect here the points we find most interesting.

1. Our method is tightly connected to the first moment method. Unfortunately, some properties of random graphs cannot be proven using this method. For example, the classical proof that the connected components of the random graph $G(S, h_1, h_2)$ for $m = (1 + \varepsilon)|S|$, for $\varepsilon > 0$, with fully random hash functions have size $O(\log n)$ uses a Galton-Watson process (see, e. g., [Bol85]). From previous work [DM90; DM92] we know that hash class $\mathcal{Z}$ has some classical properties regarding the balls-into-bins game. In the hypergraph setting this translates to a degree distribution of the vertices close to the fully random case. We are currently investigating whether this approach yields good enough bounds for the process mentioned above or not.

2. The analysis of generalized cuckoo hashing could succeed (asymptotically) using hash functions from $\mathcal{Z}$. For this, one has to extend the analysis of the behavior of $\mathcal{Z}$ on small connected hypergraphs to connected hypergraphs with super-logarithmically many edges.

3. Witness trees are another approach to tackle the analysis of generalized cuckoo hashing. We presented initial results in Section 14.3. It is open whether this approach yields good bounds on the space utilization of generalized cuckoo hashing.

4. In light of the new construction of Thorup [Tho13], it should be demonstrated in experiments whether or not $\log n$-wise and n^{δ}-wise independent hash classes with constant evaluation time are efficient.

5. It should be tested whether hash class $\mathcal{Z}$ allows running linear probing robustly or not. Furthermore, it would be interesting to see if it is ε-minwise independent (for good enough values ε).

Bibliography

[AB12] Jean-Philippe Aumasson and Daniel J. Bernstein. "SipHash: A Fast Short-Input PRF". In: *Proc. of the 13th International Conference on Cryptology in India (INDOCRYPT'12)*. Springer, 2012, pp. 489–508. DOI: 10.1007/978-3-642-34931-7_28 (cited on pp. 101, 182).

[AD13] Martin Aumüller and Martin Dietzfelbinger. "Optimal Partitioning for Dual Pivot Quicksort - (Extended Abstract)". In: *Proc. of the 40th International Colloquium on Automata, Languages and Programming (ICALP'13)*. Springer, 2013, pp. 33–44. DOI: 10.1007/978-3-642-39206-1_4 (cited on pp. 1, 4, 18).

[ADW12] Martin Aumüller, Martin Dietzfelbinger, and Philipp Woelfel. "Explicit and Efficient Hash Families Suffice for Cuckoo Hashing with a Stash". In: *Proc. of the 20th annual European symposium on Algorithms (ESA'12)*. Springer, 2012, pp. 108–120. DOI: 10.1007/978-3-642-33090-2_11 (cited on pp. 4, 137, 139).

[ADW14] Martin Aumüller, Martin Dietzfelbinger, and Philipp Woelfel. "Explicit and Efficient Hash Families Suffice for Cuckoo Hashing with a Stash". In: *Algorithmica* 70.3 (2014), pp. 428–456. DOI: 10.1007/s00453-013-9840-x (cited on pp. 4, 106, 113, 119, 125, 131, 132, 134, 135).

[Aga96] Ramesh C. Agarwal. "A Super Scalar Sort Algorithm for RISC Processors". In: *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data*. ACM, 1996, pp. 240–246. DOI: 10.1145/233269.233336 (cited on p. 68).

[AMS99] Noga Alon, Yossi Matias, and Mario Szegedy. "The Space Complexity of Approximating the Frequency Moments". In: *J. Comput. Syst. Sci.* 58.1 (1999), pp. 137–147. DOI: 10.1006/jcss.1997.1545 (cited on pp. 3, 102).

[ANS09] Yuriy Arbitman, Moni Naor, and Gil Segev. "De-amortized Cuckoo Hashing: Provable Worst-Case Performance and Experimental Results". In: *Proc. of the 36th International Colloquium on Automata, Languages and Programming (ICALP'09)*. Springer, 2009, pp. 107–118. DOI: 10.1007/978-3-642-02927-1_11 (cited on p. 145).

[App] Austin Appleby. *MurmurHash3*. https://code.google.com/p/smhasher/wiki/MurmurHash3 (cited on pp. 101, 106, 182, 184).

[Aum10] Martin Aumüller. "An alternative Analysis of Cuckoo Hashing with a Stash and Realistic Hash Functions". Diplomarbeit. Technische Universität Ilmenau, 2010, p. 98 (cited on pp. 119, 121, 125, 131–134).

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. "The Input/Output Complexity of Sorting and Related Problems". In: *Commun. ACM* 31.9 (1988), pp. 1116–1127. DOI: 10.1145/48529.48535 (cited on p. 66).

[Aza+99] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. "Balanced Allocations". In: *SIAM J. Comput.* 29.1 (1999), pp. 180–200. DOI: 10.1137/S0097539795288490 (cited on p. 174).

[BBD09] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. "Hash, Displace, and Compress". In: *Proc. of the 17th Annual European Symposium on Algorithms (ESA'09)*. Springer, 2009, pp. 682–693. DOI: 10.1007/978-3-642-04128-0_61 (cited on p. 140).

[Ben86] Jon Louis Bentley. *Programming pearls*. Addison-Wesley, 1986 (cited on p. 205).

[Big+08] Paul Biggar, Nicholas Nash, Kevin Williams, and David Gregg. "An experimental study of sorting and branch prediction". In: *ACM Journal of Experimental Algorithmics* 12 (2008). DOI: 10.1145/1227161.1370599 (cited on pp. 68, 93).

[BKZ05] Fabiano C. Botelho, Yoshiharu Kohayakawa, and Nivio Ziviani. "A Practical Minimal Perfect Hashing Method". In: *Proc. of the 4th International Workshop on Experimental and Efficient Algorithms (WEA'05)*. Springer, 2005, pp. 488–500. DOI: 10.1007/11427186_42 (cited on p. 140).

[BM05] Gerth Stølting Brodal and Gabriel Moruz. "Tradeoffs Between Branch Mispredictions and Comparisons for Sorting Algorithms". In: *Proc. of the 9th International Workshop on Algorithms and Data Structures (WADS'05)*. Springer, 2005, pp. 385–395. DOI: 10.1007/11534273_34 (cited on p. 68).

[BM93] Jon Louis Bentley and M. Douglas McIlroy. "Engineering a Sort Function". In: *Softw., Pract. Exper.* 23.11 (1993), pp. 1249–1265. DOI: 10.1002/spe.4380231105 (cited on pp. 10, 12).

[Bol85] Béla Bollobás. *Random Graphs*. Academic Press, London, 1985 (cited on pp. 105, 189).

[BPZ07] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. "Simple and Space-Efficient Minimal Perfect Hash Functions". In: *Proc. of the 10th International Workshop on Algorithms and Data Structures (WADS'07)*. Springer, 2007, pp. 139–150. DOI: 10.1007/978-3-540-73951-7_13 (cited on p. 140).

[BPZ13] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. "Practical perfect hashing in nearly optimal space". In: *Inf. Syst.* 38.1 (2013), pp. 108–131. DOI: 10.1016/j.is.2012.06.002 (cited on pp. 4, 103, 104, 131, 139–143, 148).

[Cal97] Neil J. Calkin. "Dependent Sets of Constant Weight Binary Vectors". In: *Combinatorics, Probability and Computing* 6.3 (1997), pp. 263–271 (cited on p. 137).

[Cel+13] L. Elisa Celis, Omer Reingold, Gil Segev, and Udi Wieder. "Balls and Bins: Smaller Hash Families and Faster Evaluation". In: *SIAM J. Comput.* 42.3 (2013), pp. 1030–1050. DOI: 10.1137/120871626 (cited on pp. 3, 103, 108).

[Cha+04] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. "The Bloomier filter: an efficient data structure for static support lookup tables". In: *Proc. of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*. SIAM, 2004, pp. 30–39. DOI: 10.1145/2f982792.982797 (cited on pp. 140, 155).

[CHM92] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. "An Optimal Algorithm for Generating Minimal Perfect Hash Functions". In: *Inf. Process. Lett.* 43.5 (1992), pp. 257–264. DOI: 10.1016/0020-0190(92)90220-P (cited on p. 140).

[CHM97] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. "Perfect Hashing". In: *Theor. Comput. Sci.* 182.1-2 (1997), pp. 1–143. DOI: 10.1016/S0304-3975(96)00146-6 (cited on pp. 139, 148).

[Col] Yann Collet. *xxhash*. http://code.google.com/p/xxhash/ (cited on p. 182).

[Col+98a] Richard Cole, Alan M. Frieze, Bruce M. Maggs, Michael Mitzenmacher, Andréa W. Richa, Ramesh K. Sitaraman, and Eli Upfal. "On Balls and Bins with Deletions". In: *Proc. of the 2nd International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM'98)*. Springer, 1998, pp. 145–158. DOI: 10.1007/3-540-49543-6_12 (cited on p. 159).

[Col+98b] Richard Cole, Bruce M. Maggs, Friedhelm Meyer auf der Heide, Michael Mitzenmacher, Andréa W. Richa, Klaus Schröder, Ramesh K. Sitaraman, and Berthold Vöcking. "Randomized Protocols for Low Congestion Circuit Routing in Multistage Interconnection Networks". In: *Proc. of the 3th Annual ACM Symposium on the Theory of Computing (STOC'98)*. ACM, 1998, pp. 378–388. DOI: 10.1145/276698.276790 (cited on p. 159).

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)* MIT Press, 2009, pp. I–XIX, 1–1292 (cited on pp. 1, 9).

[CW03] Scott A. Crosby and Dan S. Wallach. "Denial of Service via Algorithmic Complexity Attacks". In: *Proc. of the 12th Conference on USENIX Security Symposium - Volume 12*. SSYM'03. Washington, DC: USENIX Association, 2003, pp. 3–3 (cited on p. 101).

[CW77] J. Lawrence Carter and Mark N. Wegman. "Universal classes of hash functions (Extended Abstract)". In: *Proc. of the 9th Annual ACM Symposium on Theory of Computing (STOC'77)*. ACM, 1977, pp. 106–112. DOI: 10.1145/800105.803400 (cited on p. 107).

[CW79] Larry Carter and Mark N. Wegman. "Universal Classes of Hash Functions". In: *J. Comput. Syst. Sci.* 18.2 (1979), pp. 143–154. DOI: 10.1016/0022-0000(79)90044-8 (cited on pp. 3, 101, 102, 107, 182).

[Dah+14] Søren Dahlgaard, Mathias Bæk Tejs Knudsen, Eva Rotenberg, and Mikkel Thorup. "The Power of Two Choices with Simple Tabulation". In: *CoRR* abs/1407.6846 (2014) (cited on pp. 3, 103).

[DH01] Martin Dietzfelbinger and Torben Hagerup. "Simple Minimal Perfect Hashing in Less Space". In: *Proc. of the 9th Annual European Symposium on Algorithms (ESA'01)*. Springer, 2001, pp. 109–120. DOI: 10.1007/3-540-44676-1_9 (cited on p. 140).

[Die+10] Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink. "Tight Thresholds for Cuckoo Hashing via XOR-SAT". In: *Proc. of the 37th International Colloquium on Automata, Languages and Programming (ICALP'10)*. Springer, 2010, pp. 213–225. DOI: 10.1007/978-3-642-14165-2_19 (cited on pp. 149, 154).

[Die+97] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. "A Reliable Randomized Algorithm for the Closest-Pair Problem". In: *J. Algorithms* 25.1 (1997), pp. 19–51. DOI: 10.1006/jagm.1997.0873 (cited on pp. 107, 183).

[Die05] Reinhard Diestel. *Graph Theory*. Springer, 2005 (cited on pp. 111, 125, 128, 147, 154, 168).

[Die07] Martin Dietzfelbinger. "Design Strategies for Minimal Perfect Hash Functions". In: *4th International Symposium on Stochastic Algorithms: Foundations and Applications (SAGA'07)*. Springer, 2007, pp. 2–17. DOI: 10.1007/978-3-540-74871-7_2 (cited on pp. 103, 139, 140, 148).

[Die12] Martin Dietzfelbinger. "On Randomness in Hash Functions (Invited Talk)". In: *29th International Symposium on Theoretical Aspects of Computer Science (STACS'12)*. Springer, 2012, pp. 25–28. DOI: 10.4230/LIPIcs.STACS.2012.25 (cited on p. 103).

[Die96] Martin Dietzfelbinger. "Universal Hashing and k-Wise Independent Random Variables via Integer Arithmetic without Primes." In: *Proc. of the 13th Annual Symposium on Theoretical Aspects of Computer Science (STACS'96)*. Springer, 1996, pp. 569–580. DOI: 10.1007/3-540-60922-9_46 (cited on p. 183).

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976 (cited on p. 69).

[DM03] Luc Devroye and Pat Morin. "Cuckoo hashing: Further analysis". In: *Inf. Process. Lett.* 86.4 (2003), pp. 215–219. DOI: 10.1016/S0020-0190(02)00500-8 (cited on pp. 120, 121, 135).

[DM09] Luc Devroye and Ebrahim Malalla. "On the k-orientability of random graphs". In: *Discrete Mathematics* 309.6 (2009), pp. 1476–1490. DOI: 10.1016/j.disc.2008.02.023 (cited on p. 176).

[DM90] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. "A New Universal Class of Hash Functions and Dynamic Hashing in Real Time". In: *Proc. of the 17th International Colloquium on Automata, Languages and Programming (ICALP'90)*. Springer, 1990, pp. 6–19. DOI: 10.1007/BFb0032018 (cited on pp. 3, 103, 104, 108, 109, 189).

[DM92] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. "Dynamic Hashing in Real Time". In: *Informatik, Festschrift zum 60. Geburtstag von Günter Hotz*. Teubner, 1992, pp. 95–119 (cited on pp. 108, 189).

[DM93] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. "Simple, Efficient Shared Memory Simulations". In: *Proc. of the 5th ACM Symposium on Parallelism in Algorithms and Architectures, (SPAA'93)*. ACM, 1993, pp. 110–119. DOI: 10.1145/165231.165246 (cited on p. 164).

[DP09] Devdatt P. Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, 2009, pp. I–XIV, 1–196 (cited on p. 21).

[DR09] Martin Dietzfelbinger and Michael Rink. "Applications of a Splitting Trick". In: *Proc. of the 36th International Colloquium on Automata, Languages and Programming (ICALP'09)*. Springer, 2009, pp. 354–365. DOI: 10.1007/978-3-642-02927-1_30 (cited on pp. 103, 137, 140).

[DS09a] Martin Dietzfelbinger and Ulf Schellbach. "On risks of using cuckoo hashing with simple universal hash classes". In: *Proc. of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'09)*. SIAM, 2009, pp. 795–804. DOI: 10.1145/1496770.1496857 (cited on p. 102).

[DS09b] Martin Dietzfelbinger and Ulf Schellbach. "Weaknesses of Cuckoo Hashing with a Simple Universal Hash Class: The Case of Large Universes". In: *Proc. of the 35th Conference on Current Trends in Theory and Practice of Informatics (SOFSEM'09)*. 2009, pp. 217–228. DOI: 10.1007/978-3-540-95891-8_22 (cited on p. 102).

[DT14] Søren Dahlgaard and Mikkel Thorup. "Approximately Minwise Independence with Twisted Tabulation". In: *Proc. of the 14th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT'14)*. Springer, 2014, pp. 134–145. DOI: 10.1007/978-3-319-08404-6_12 (cited on pp. 3, 103).

[DW03] Martin Dietzfelbinger and Philipp Woelfel. "Almost random graphs with simple hash functions". In: *Proc. of the 35th annual ACM Symposium on Theory of computing (STOC'03)*. ACM, 2003, pp. 629–638. DOI: 10.1145/780542.780634 (cited on pp. 3, 4, 103–105, 108, 119, 125, 135, 136, 177).

[DW07] Martin Dietzfelbinger and Christoph Weidling. "Balanced allocation and dictionaries with tightly packed constant size bins". In: *Theor. Comput. Sci.* 380.1-2 (2007), pp. 47–68. DOI: 10.1016/j.tcs.2007.02.054 (cited on p. 176).

[Emd70] M. H. van Emden. "Increasing the efficiency of quicksort". In: *Commun. ACM* 13.9 (Sept. 1970), pp. 563–567. DOI: 10.1145/362736.362753 (cited on pp. 12, 61).

[Epp+14] David Eppstein, Michael T. Goodrich, Michael Mitzenmacher, and Pawel Pszona. "Wear Minimization for Cuckoo Hashing: How Not to Throw a Lot of Eggs into One Basket". In: *Proc. of the 13th International Symposium of Experimental Algorithms, (SEA'14)*. Springer, 2014, pp. 162–173. DOI: 10.1007/978-3-319-07959-2_14 (cited on pp. 4, 104, 147, 155, 156, 159–161).

[ER60] P Erdös and A Rényi. "On the evolution of random graphs". In: *Publ. Math. Inst. Hung. Acad. Sci* 5 (1960), pp. 17–61 (cited on p. 149).

[FK84] Michael L Fredman and János Komlós. "On the size of separating systems and families of perfect hash functions". In: *SIAM Journal on Algebraic Discrete Methods* 5.1 (1984), pp. 61–68 (cited on p. 139).

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. "Storing a Sparse Table with 0(1) Worst Case Access Time". In: *J. ACM* 31.3 (1984), pp. 538–544. DOI: 10.1145/828.1884 (cited on pp. 108, 139).

[FM12] Alan M. Frieze and Pàll Melsted. "Maximum matchings in random bipartite graphs and the space utilization of Cuckoo Hash tables". In: *Random Struct. Algorithms* 41.3 (2012), pp. 334–364. DOI: 10.1002/rsa.20427 (cited on p. 149).

[FMM11] Alan M. Frieze, Pàll Melsted, and Michael Mitzenmacher. "An Analysis of Random-Walk Cuckoo Hashing". In: *SIAM J. Comput.* 40.2 (2011), pp. 291–308. DOI: 10.1137/090770928 (cited on p. 149).

[Fot+05] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. "Space Efficient Hash Tables with Worst Case Constant Access Time". In: *Theory Comput. Syst.* 38.2 (2005), pp. 229–248. DOI: 10.1007/s00224-004-1195-x (cited on pp. 4, 103, 104, 147–149, 154, 155).

[FP10] Nikolaos Fountoulakis and Konstantinos Panagiotou. "Orientability of Random Hypergraphs and the Power of Multiple Choices". In: *Proc. of the 37th International Colloquium on Automata, Languages and Programming (ICALP'10)*. Springer, 2010, pp. 348–359. DOI: 10.1007/978-3-642-14165-2_30 (cited on p. 149).

[FPS13] Nikolaos Fountoulakis, Konstantinos Panagiotou, and Angelika Steger. "On the Insertion Time of Cuckoo Hashing". In: *SIAM J. Comput.* 42.6 (2013), pp. 2156–2181. DOI: 10.1137/100797503 (cited on pp. 148, 149, 163).

[Fri+12] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. "Cache-Oblivious Algorithms". In: *ACM Transactions on Algorithms* 8.1 (2012), p. 4. DOI: 10.1145/2071379.2071383 (cited on p. 67).

[GM11] Michael T. Goodrich and Michael Mitzenmacher. "Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation". In: *Proc. of the 38th International Colloquium on Automata, Languages and Programming (ICALP'11)*. Springer, 2011, pp. 576–587. DOI: 10.1007/978-3-642-22012-8_46 (cited on p. 135).

[Gou72] Henry W. Gould. *Combinatorial Identities*. 1972 (cited on p. 32).

[Hen91] Pascal Hennequin. "Analyse en moyenne d'algorithmes: tri rapide et arbres de recherche". available at http://www-lor.int-evry.fr/~pascal/. PhD thesis. Ecole Politechnique, Palaiseau, 1991 (cited on pp. 9, 10, 13, 18, 39, 40, 46, 61, 62).

[Hoa62] C. A. R. Hoare. "Quicksort". In: *Comput. J.* 5.1 (1962), pp. 10–15 (cited on pp. 1, 9, 12, 40, 67).

[HP12] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)* Morgan Kaufmann, 2012 (cited on pp. 13, 66, 215).

[HSS96] Friedhelm Meyer auf der Heide, Christian Scheideler, and Volker Stemann. "Exploiting Storage Redundancy to Speed up Randomized Shared Memory Simulations". In: *Theor. Comput. Sci.* 162.2 (1996), pp. 245–281. DOI: 10.1016/0304-3975(96)00032-1 (cited on p. 159).

[HT01] Torben Hagerup and Torsten Tholey. "Efficient Minimal Perfect Hashing in Nearly Minimal Space". In: *Proc. of the 18th Annual Symposium on Theoretical Aspects of Computer Science (STACS'01)*. 2001, pp. 317–326. DOI: 10.1007/3-540-44693-1_28 (cited on p. 139).

[Ili14] Vasileios Iliopoulos. "A note on multipivot Quicksort". In: *CoRR* abs/1407.7459 (2014) (cited on pp. 46, 62).

[Ind01] Piotr Indyk. "A Small Approximately Min-Wise Independent Family of Hash Functions". In: *J. Algorithms* 38.1 (2001), pp. 84–90. DOI: 10.1006/jagm.2000.1131 (cited on p. 102).

[JEB86] C. T. M. Jacobs and Peter van Emde Boas. "Two Results on Tables". In: *Inf. Process. Lett.* 22.1 (1986), pp. 43–48. DOI: 10.1016/0020-0190(86)90041-4 (cited on p. 139).

[JM13] Tomasz Jurkiewicz and Kurt Mehlhorn. "The cost of address translation". In: *Proc. of the 15th Meeting on Algorithm Engineering and Experiments, (ALENEX'13)*. SIAM, 2013, pp. 148–162. DOI: 10.1137/1.9781611972931.13 (cited on p. 68).

[JMT14] Jiayang Jiang, Michael Mitzenmacher, and Justin Thaler. "Parallel peeling algorithms". In: *Proc. of the 26th ACM Symposium on Parallelism in Algorithms and Architectures, (SPAA '14)*. ACM, 2014, pp. 319–330. DOI: 10.1145/2612669.2612674 (cited on p. 176).

[JNL02] Daniel Jiménez-González, Juan J. Navarro, and Josep-Lluis Larriba-Pey. "The Effect of Local Sort on Parallel Sorting Algorithms". In: *10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP'02)*. IEEE Computer Society, 2002, pp. 360–367. DOI: 10.1109/EMPDP.2002.994310 (cited on p. 68).

[Kho13] Megha Khosla. "Balls into Bins Made Faster". In: *Proc. of the 21st European Symposium on Algorithms (ESA'13)*. Springer, 2013, pp. 601–612. DOI: 10.1007/978-3-642-40450-4_51 (cited on pp. 4, 104, 147, 148, 155–157, 163).

[Kla14] Pascal Klaue. "Optimal Partitionierungsverfahren für Multi-Pivot-Quicksort". in German. MA thesis. TU Ilmenau, 2014 (cited on pp. 45, 55, 62).

[KLM96] Richard M. Karp, Michael Luby, and Friedhelm Meyer auf der Heide. "Efficient PRAM Simulation on a Distributed Memory Machine". In: *Algorithmica* 16.4/5 (1996), pp. 517–542. DOI: 10.1007/BF01940878 (cited on pp. 3, 143–145).

[KMW08] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. "More Robust Hashing: Cuckoo Hashing with a Stash". In: *Proc. of the 16th annual European symposium on Algorithms (ESA'08)*. Springer, 2008, pp. 611–622. DOI: 10.1007/978-3-540-87744-8_51 (cited on pp. 131, 134).

[KMW09] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. "More Robust Hashing: Cuckoo Hashing with a Stash". In: *SIAM J. Comput.* 39.4 (2009), pp. 1543–1561. DOI: 10.1137/080728743 (cited on pp. 4, 104, 131, 132).

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973 (cited on pp. 9, 56, 62).

[KS06] Kanela Kaligosi and Peter Sanders. "How Branch Mispredictions Affect Quicksort". In: *Proc. of the 14th Annual European Symposium on Algorithms (ESA'06)*. Springer, 2006, pp. 780–791. DOI: 10.1007/11841036_69 (cited on p. 68).

[Kus+14] Shrinu Kushagra, Alejandro López-Ortiz, Aurick Qiao, and J Ian Munro. "Multi-Pivot Quicksort: Theory and Experiments". In: *Proc. of the 16th Meeting on Algorithms Engineering and Experiments (ALENEX'14)*. SIAM, 2014, pp. 47–60. DOI: 10.1137/1.9781611973198.6 (cited on pp. 1, 2, 13, 46, 53–55, 65, 67, 70, 77, 79, 82, 89, 93, 94, 97, 212, 214, 227).

[KW12] Toryn Qwyllyn Klassen and Philipp Woelfel. "Independence of Tabulation-Based Hash Classes". In: *Proc. Theoretical Informatics - 10th Latin American Symposium (LATIN'12)*. Springer, 2012, pp. 506–517. DOI: 10.1007/978-3-642-29344-3_43 (cited on p. 108).

[KŁ02] Michał Karoński and Tomasz Łuczak. "The phase transition in a random hypergraph". In: *Journal of Computational and Applied Mathematics* 142.1 (2002), pp. 125–135. DOI: 10.1016/S0377-0427(01)00464-2 (cited on pp. 147, 149–151).

[Lev09] David Levinthal. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors*. `https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`. 2009 (cited on pp. 66, 67, 215).

[LL99] Anthony LaMarca and Richard E. Ladner. "The Influence of Caches on the Performance of Sorting". In: *J. Algorithms* 31.1 (1999), pp. 66–104. DOI: `10.1006/jagm.1998.0985` (cited on pp. 13, 67, 82).

[LO14] Alejandro Lopez-Ortiz. *Multi-Pivot Quicksort: Theory and Experiments*. Talk given at Dagstuhl Seminar 14091: "Data Structures and Advanced Models of Computation on Big Data". 2014 (cited on p. 2).

[Maj+96] Bohdan S. Majewski, Nicholas C. Wormald, George Havas, and Zbigniew J. Czech. "A Family of Perfect Hashing Methods". In: *Comput. J.* 39.6 (1996), pp. 547–554. DOI: `10.1093/comjnl/39.6.547` (cited on pp. 140, 154).

[MBM93] Peter M. McIlroy, Keith Bostic, and M. Douglas McIlroy. "Engineering Radix Sort". In: *Computing Systems* 6.1 (1993), pp. 5–27 (cited on pp. 67, 69, 70).

[McM78] Colin L. McMaster. "An Analysis of Algorithms for the Dutch National Flag Problem". In: *Commun. ACM* 21.10 (1978), pp. 842–846. DOI: `10.1145/359619.359629` (cited on p. 69).

[Meh84] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Vol. 1. EATCS Monographs on Theoretical Computer Science. Springer, 1984 (cited on p. 139).

[Mey78] S. J. Meyer. "A failure of structured programming". In: *Zilog Corp., Software Dept. Technical Rep. No. 5, Cupertino, CA* (1978) (cited on p. 69).

[MM09] Marc Mezard and Andrea Montanari. *Information, physics, and computation*. Oxford University Press, 2009 (cited on p. 154).

[MNW15] Conrado Martínez, Markus E. Nebel, and Sebastian Wild. "Analysis of Branch Misses in Quicksort". In: *Proc. of the 12th Meeting on Analytic Algorithmics and Combinatorics (ANALCO' 15)*. To appear. 2015 (cited on pp. 1, 13, 68).

[Mol05] Michael Molloy. "Cores in random hypergraphs and Boolean formulas". In: *Random Struct. Algorithms* 27.1 (2005), pp. 124–135. DOI: `10.1002/rsa.20061` (cited on pp. 148, 150, 176).

[MR01] Conrado Martínez and Salvador Roura. "Optimal Sampling Strategies in Quicksort and Quickselect". In: *SIAM J. Comput.* 31.3 (2001), pp. 683–705. DOI: `10.1137/S0097539700382108` (cited on pp. 12, 39, 41, 61).

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995 (cited on p. 4).

[MS03] Kurt Mehlhorn and Peter Sanders. "Scanning Multiple Sequences Via Cache Memory". In: *Algorithmica* 35.1 (2003), pp. 75–93. DOI: 10.1007/s00453-002-0993-2 (cited on pp. 66, 82).

[MU05] Michael Mitzenmacher and Eli Upfal. *Probability and Computing : Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005 (cited on p. 4).

[Mus97] David R. Musser. "Introspective Sorting and Selection Algorithms". In: *Softw., Pract. Exper.* 27.8 (1997), pp. 983–993 (cited on p. 92).

[MV08] Michael Mitzenmacher and Salil P. Vadhan. "Why simple hash functions work: exploiting the entropy in a data stream". In: *Proc. of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'08)*. SIAM, 2008, pp. 746–755. DOI: 10.1145/1347082.1347164 (cited on p. 103).

[NW14] Markus E. Nebel and Sebastian Wild. "Pivot Sampling in Java 7's Dual-Pivot Quicksort – Exploiting Asymmetries in Yaroslavskiy's Partitioning Scheme". In: *Analysis of Algorithms (AofA'14)*. 2014 (cited on pp. 1, 39, 41).

[Ott48] Richard Otter. "The number of trees". In: *Annals of Mathematics* (1948), pp. 583–599 (cited on p. 145).

[PA] Geoff Pike and Jyrki Alakuijala. *CityHash*. http://code.google.com/p/cityhash/ (cited on p. 182).

[Pag14] Rasmus Pagh. *Basics of Hashing: k-independence and applications*. Talk given at Summer School on Hashing'14 in Copenhagen. 2014 (cited on p. 101).

[Pag99] Rasmus Pagh. "Hash and Displace: Efficient Evaluation of Minimal Perfect Hash Functions". In: *Proc. of the 6th International Workshop on Algorithms and Data Structures (WADS'99)*. Springer, 1999, pp. 49–54. DOI: 10.1007/3-540-48447-7_5 (cited on p. 140).

[PP08] Anna Pagh and Rasmus Pagh. "Uniform Hashing in Constant Time and Optimal Space". In: *SIAM J. Comput.* 38.1 (2008), pp. 85–96. DOI: 10.1137/060658400 (cited on pp. 4, 103, 104, 131, 137–139).

[PPR09] Anna Pagh, Rasmus Pagh, and Milan Ruzic. "Linear Probing with Constant Independence". In: *SIAM J. Comput.* 39.3 (2009), pp. 1107–1120. DOI: 10.1137/070702278 (cited on pp. 3, 102).

[PR04] Rasmus Pagh and Flemming Friche Rodler. "Cuckoo hashing". In: *J. Algorithms* 51.2 (2004), pp. 122–144. DOI: 10.1016/j.jalgor.2003.12.002 (cited on pp. 102, 104, 119, 121, 135).

[PT10] Mihai Patrascu and Mikkel Thorup. "On the k-Independence Required by Linear Probing and Minwise Independence". In: *Proc. of the 37th International Colloquium on Automata, Languages and Programming (ICALP'10)*. Springer, 2010, pp. 715–726. DOI: 10.1007/978-3-642-14165-2_60 (cited on p. 102).

[PT11] Mihai Pătraşcu and Mikkel Thorup. "The Power of Simple Tabulation Hashing". In: *Proc. of the 43rd Annual ACM Symposium on Theory of Computing (STOC'11)*. 2011, pp. 1–10. DOI: 10.1145/1993636.1993638 (cited on pp. 3, 4, 103).

[PT12] Mihai Patrascu and Mikkel Thorup. "The Power of Simple Tabulation Hashing". In: *J. ACM* 59.3 (2012), p. 14. DOI: 10.1145/2220357.2220361 (cited on pp. 103, 106, 108, 181, 182, 184).

[PT13] Mihai Patrascu and Mikkel Thorup. "Twisted Tabulation Hashing." In: *Proc. of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'13)*. SIAM, 2013, pp. 209–228. DOI: 10.1137/1.9781611973105.16 (cited on pp. 3, 103, 108).

[Rah02] Naila Rahman. "Algorithms for Hardware Caches and TLB". In: *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]*. 2002, pp. 171–192. DOI: 10.1007/3-540-36574-5_8 (cited on pp. 66, 67, 80).

[Raj+01] Sanguthevar Rajasekaran, Panos M. Pardalos, John H. Reif, and Rosé Rolim, eds. *Handbook of Randomized Computing, Vol. 1*. Kluwer Academic Publishers, 2001 (cited on p. 164).

[Rin14] Michael Rink. "Thresholds for Matchings in Random Bipartite Graphs with Applications to Hashing-Based Data Structures". to appear. PhD thesis. Technische Universität Ilmenau, 2014 (cited on pp. 136, 137).

[Rou01] Salvador Roura. "Improved master theorems for divide-and-conquer recurrences". In: *J. ACM* 48.2 (2001), pp. 170–205. DOI: 10.1145/375827.375837 (cited on pp. 16, 17, 42, 46, 47).

[RRW14] Omer Reingold, Ron D. Rothblum, and Udi Wieder. "Pseudorandom Graphs in Data Structures". In: *Proc. of the 41st International Colloquium on Automata, Languages and Programming (ICALP'14)*. Springer, 2014, pp. 943–954. DOI: 10.1007/978-3-662-43948-7_78 (cited on pp. 3, 103).

[SB] Robert Sedgewick and Jon Bentley. *Quicksort is optimal*. http://www.cs.princeton.edu/~rs/talks/QuicksortIsOptimal.pdf (cited on p. 69).

[Sed75] Robert Sedgewick. "Quicksort". PhD thesis. Standford University, 1975 (cited on pp. 1, 9, 28, 38, 208, 209).

[Sed77] Robert Sedgewick. "Quicksort with Equal Keys". In: *SIAM J. Comput.* 6.2 (1977), pp. 240–268. DOI: 10.1137/0206018 (cited on p. 9).

[Sed78] Robert Sedgewick. "Implementing Quicksort programs". In: *Commun. ACM* 21.10 (1978), pp. 847–857. DOI: 10.1145/359619.359631 (cited on p. 205).

[SF96] Robert Sedgewick and Philippe Flajolet. *An introduction to the analysis of algorithms*. Addison-Wesley-Longman, 1996, pp. I–XV, 1–492 (cited on p. 9).

[Sie04] Alan Siegel. "On Universal Classes of Extremely Random Constant-Time Hash Functions". In: *SIAM J. Comput.* 33.3 (2004), pp. 505–543. DOI: 10.1137/S0097539701386216 (cited on pp. 102, 107, 135, 137).

[Sof12] Anthony Sofo. "New classes of harmonic number identities". In: *J. Integer Seq* 15 (2012) (cited on p. 48).

[SPS85] Jeanette Schmidt-Pruzan and Eli Shamir. "Component structure in the evolution of random hypergraphs". English. In: *Combinatorica* 5.1 (1985), pp. 81–94. DOI: 10.1007/BF02579445 (cited on p. 147).

[SS00] Thomas Schickinger and Angelika Steger. "Simplified Witness Tree Arguments". In: *Proc. of the 27th Conference on Current Trends in Theory and Practice of Informatics (SOFSEM'00)*. Springer, 2000, pp. 71–87. DOI: 10.1007/3-540-44411-4_6 (cited on pp. 4, 104, 147, 159, 165, 166, 168, 169, 174, 175).

[SS63] John C. Shepherdson and Howard E. Sturgis. "Computability of Recursive Functions". In: *J. ACM* 10.2 (1963), pp. 217–255. DOI: 10.1145/321160.321170 (cited on p. 66).

[SS90] Jeanette P. Schmidt and Alan Siegel. "The Spatial Complexity of Oblivious k-Probe Hash Functions". In: *SIAM J. Comput.* 19.5 (1990), pp. 775–786. DOI: 10.1137/0219054 (cited on p. 139).

[Ste96] Volker Stemann. "Parallel Balanced Allocations". In: *Proc. of the 8th ACM Symposium on Parallelism in Algorithms and Architectures, (SPAA'96)*. ACM, 1996, pp. 261–269. DOI: 10.1145/237502.237565 (cited on pp. 159, 164).

[SW04] Peter Sanders and Sebastian Winkel. "Super Scalar Sample Sort". In: *Proc. of the 12th Annual European Symposium on Algorithms (ESA'04)*. Springer, 2004, pp. 784–796. DOI: 10.1007/978-3-540-30140-0_69 (cited on pp. 13, 67, 68, 70, 91–93).

[Tan93] Kok-Hooi Tan. "An asymptotic analysis of the number of comparisons in multipartition quicksort". PhD thesis. Carnegie Mellon University, 1993 (cited on p. 13).

[Tho13] Mikkel Thorup. "Simple Tabulation, Fast Expanders, Double Tabulation, and High Independence". In: *Proc. 54th Annual Symposium on Foundations of Computer Science (FOCS)*. ACM, 2013, pp. 90–99. DOI: 10.1109/FOCS.2013.18 (cited on pp. 102, 135, 137, 139, 189).

[TZ04] Mikkel Thorup and Yin Zhang. "Tabulation based 4-universal hashing with applications to second moment estimation". In: *Proc. of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*. SIAM, 2004, pp. 615–624. DOI: 10.1145/982792.982884 (cited on p. 102).

[TZ12] Mikkel Thorup and Yin Zhang. "Tabulation-Based 5-Independent Hashing with Applications to Linear Probing and Second Moment Estimation". In: *SIAM J. Comput.* 41.2 (2012), pp. 293–331. DOI: 10.1137/100800774 (cited on pp. 102, 108).

[Vöc03] Berthold Vöcking. "How asymmetry helps load balancing". In: *J. ACM* 50.4 (2003), pp. 568–589. DOI: 10.1145/792538.792546 (cited on pp. 103, 104, 109, 159, 166, 174, 175).

[WC79] Mark N. Wegman and Larry Carter. "New Classes and Applications of Hash Functions". In: *Proc. 20th Annual Symposium on Foundations of Computer Science (FOCS'79)*. IEEE Computer Society, 1979, pp. 175–182. DOI: 10.1109/SFCS.1979.26 (cited on pp. 107, 108).

[WC81] Mark N. Wegman and Larry Carter. "New Hash Functions and Their Use in Authentication and Set Equality". In: *J. Comput. Syst. Sci.* 22.3 (1981), pp. 265–279. DOI: 10.1016/0022-0000(81)90033-7 (cited on p. 107).

[Wil+13] Sebastian Wild, Markus E. Nebel, Raphael Reitzig, and Ulrich Laube. "Engineering Java 7's Dual Pivot Quicksort Using MaLiJAn". In: *Proc. of the 15th Meeting on Algorithms Engineering and Experiments (ALENEX'13)*. 2013, pp. 55–69. DOI: 10.1137/1.9781611972931.5 (cited on pp. 1, 40).

[Wil13] Sebastian Wild. "Java 7's Dual Pivot Quicksort". MA thesis. University of Kaiserslautern, 2013, p. 171 (cited on p. 29).

[WN12] Sebastian Wild and Markus E. Nebel. "Average Case Analysis of Java 7's Dual Pivot Quicksort". In: *Proc. of the 20th European Symposium on Algorithms (ESA'12)*. 2012, pp. 825–836. DOI: 10.1007/978-3-642-33090-2_71 (cited on pp. 1, 9–12, 28, 38, 46, 67, 70, 79, 206).

[WNM13] Sebastian Wild, Markus E. Nebel, and Hosam Mahmoud. "Analysis of Quickselect under Yaroslavskiy's Dual-Pivoting Algorithm". In: *CoRR* abs/1306.3819 (2013) (cited on p. 1).

[WNN13] Sebastian Wild, Markus E. Nebel, and Ralph Neininger. "Average case and distributional analysis of Java 7's dual pivot quicksort". In: *CoRR* abs/1304.0988 (2013). Accepted for publication in ACM Transactions on Algorithms (cited on pp. 1, 9, 69, 70, 94).

[Woe05] Philipp Woelfel. *The Power of Two Random Choices with Simple Hash Functions*. Available at http://eiche.theoinf.tu-ilmenau.de/maumueller-diss/. 2005 (cited on p. 4).

[Woe06a] Philipp Woelfel. "Asymmetric balanced allocation with simple hash functions". In: *Proc. of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'06)*. ACM, 2006, pp. 424–433. DOI: 10.1145/1109557.1109605 (cited on pp. 3, 103–105, 109, 166, 175).

[Woe06b] Philipp Woelfel. "Maintaining External Memory Efficient Hash Tables". In: *Proc. of the 10th International Workshop on Randomization and Computation (RANDOM'06)*. Springer, 2006, pp. 508–519. DOI: 10.1007/11830924_46 (cited on p. 140).

[Woe99] Philipp Woelfel. "Efficient Strongly Universal and Optimally Universal Hashing". In: *24th International Symposium on Mathematical Foundations of Computer Science (MFCS'99)*. Springer, 1999, pp. 262–272. DOI: 10.1007/3-540-48340-3_24 (cited on p. 107).

[Yar09] Vladimir Yaroslavskiy. *Dual-pivot quicksort*. 2009 (cited on pp. 1, 97).

[Zob70] Albert Lindsey Zobrist. *A new hashing method with application for game playing*. Tech. rep. University of Wisconsin, 1970 (cited on p. 103).

[ÖP03] Anna Östlin and Rasmus Pagh. "Uniform hashing in constant time and linear space". In: *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC'03)*. ACM, 2003, pp. 622–628. DOI: 10.1145/780542.780633 (cited on pp. 136, 137).

A. Quicksort: Algorithms in Detail

A.1. Quicksort

The following algorithm is an implementation of classical quicksort, slightly modified from [Sed78]. Note that there is a different, somewhat simpler, partitioning procedure which does not use two indices that move towards each other. Pseudocode can be found in, e. g., [Ben86, p. 110]. (Also read footnote † on that page.) This "simpler" algorithm is obtained from our Algorithm 3 ("Exchange$_1$") by setting $k' \leftarrow 1$ on Line 2 and simplifying unnecessary code for this special case.

Algorithm 4 The quicksort algorithm

procedure *Quicksort*($A[1..n]$)
Requires: Sentinel $A[0] = -\infty$;

```
 1: if n ≤ 1 then return;
 2: p ← A[n];
 3: i ← 0; j ← n;
 4: while true do
 5:     do i ← i + 1 while A[i] < p end while
 6:     do j ← j − 1 while A[j] > p end while
 7:     if j > i then
 8:         Exchange A[i] and A[j];
 9:     else
10:         break;
11: Exchange A[i] and A[n];
12: Quicksort(A[1..i − 1]);
13: Quicksort(A[i + 1..n]);
```

A.2. Dual-Pivot Quicksort

A.2.1. General Setup

The general outline of a dual-pivot quicksort algorithm is presented as Algorithm 5.

Algorithm 5 Dual-Pivot-Quicksort (outline)

```
procedure Dual-Pivot-Quicksort(A, left, right)
1  if right − left ≤ THRESHOLD then
2      InsertionSort(A, left, right);
3      return ;
4  if A[right] > A[left] then
5      swap A[left] and A[right];
6  p ← A[left];
7  q ← A[right];
8  partition(A, p, q, left, right, pos_p, pos_q);
9  Dual-Pivot-Quicksort(A, left, pos_p - 1);
10 Dual-Pivot-Quicksort(A, pos_p + 1, pos_q - 1);
11 Dual-Pivot-Quicksort(A, pos_q + 1, right);
```

To get an actual algorithm we have to implement a *partition* function that partitions the input as depicted in Figure 2.1. A partition procedure in this thesis has two output variables pos_p and pos_q that are used to return the positions of the two pivots in the partitioned array.

For moving elements around, we make use of the following two operations.

```
procedure rotate3(a, b, c)
1  tmp ← a;
2  a ← b;
3  b ← c;
4  c ← tmp;
```

```
procedure rotate4(a, b, c, d)
1  tmp ← a;
2  a ← b;
3  b ← c;
4  c ← d;
5  d ← tmp;
```

A.2.2. Yaroslavskiy's Partitioning Method

As mentioned in Section 4.1, Yaroslavskiy's algorithm makes sure that for ℓ large elements in the input it will compare ℓ or $\ell - 1$ elements to the larger pivot first. How does it accomplish this? By default, it compares to the smaller pivot first, but for each large elements that it sees, it will compare the next element to the larger pivot first.

Algorithm 6 shows the partition step of (a slightly modified version of) Yaroslavskiy's algorithm. In contrast to the algorithm studied in [WN12], it saves an unnecessary index check at Line 8, and uses a `rotate3` operation to save assignments. (In our experiments this makes Yaroslavskiy's algorithm about 4% faster.)

Algorithm 6 Yaroslavskiy's Partitioning Method

procedure *Y-Partition*(A, p, q, *left*, *right*, pos_p, pos_q)

```
1  l ← left + 1; g ← right − 1; k ← l;
2  while k ≤ g do
3      if A[k] < p then
4          swap A[k] and A[l];
5          l ← l + 1;
6      else
7          if A[k] > q then
8              while A[g] > q do
9                  g ← g − 1;
10             if k < g then
11                 if A[g] < p then
12                     rotate3(A[g], A[k], A[l]);
13                     l ← l + 1;
14                 else
15                     swap A[k] and A[g];
16                 g ← g − 1;
17     k ← k + 1;
18 swap A[left] and A[l − 1];
19 swap A[right] and A[g + 1];
20 pos_p ← l − 1; pos_q ← g + 1;
```

A.2.3. Algorithm Using "Always Compare to the Larger Pivot First"

Algorithm 7 presents an implementation of the strategy *"Always compare to the larger pivot first."* Like Yaroslavskiy's algorithm, it uses three pointers into the array. One pointer is used to scan the array from left to right until a large element has been found (moving small elements to a correct position using the second pointer on the way). Subsequently, it scans the array from right to left using the third pointer until a non-large element has been found. These two elements are then placed into a correct position. This is repeated until the two pointers have crossed. The design goal is to check as rarely as possible if these two pointers have met, since this event occurs infrequently. (In contrast, Yaroslavskiy's algorithm checks this for each element scanned by index `k` in Algorithm 6.)

This strategy makes $2n \ln n$ comparisons and $1.6n \ln n$ assignments on average.

Algorithm 7 Always Compare To Larger Pivot First Partitioning

```
procedure Q-Partition(A, p, q, left, right, pos_p, pos_q)
 1  i ← left + 1; k ← right − 1; j ← i;
 2  while j ≤ k do
 3      while q < A[k] do
 4          k ← k − 1;
 5      while A[j] < q do
 6          if A[j] < p then
 7              swap A[i] and A[j];
 8              i ← i + 1;
 9          j ← j + 1;
10      if j < k then
11          if A[k] > p then
12              rotate3(A[k], A[j], A[i]);
13              i ← i + 1;
14          else
15              swap A[j] and A[k];
16          k ← k − 1;
17      j ← j + 1;
18  swap A[left] and A[i − 1];
19  swap A[right] and A[k + 1];
20  pos_p ← i − 1; pos_q ← k + 1;
```

A.2.4. Partitioning Methods Based on Sedgewick's Algorithm

Algorithm 8 shows Sedgewick's partitioning method as studied in [Sed75].

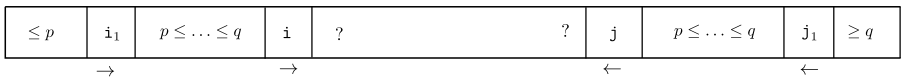

Figure A.1.: An intermediate partitioning step in Sedgewick's algorithm.

Sedgewick's partitioning method uses two pointers `i` and `j` to scan through the input. It does not swap entries in the strict sense, but rather has two "holes" at positions $\mathtt{i}_1$ resp. $\mathtt{j}_1$ that can be filled with small resp. large elements. "Moving a hole" is not a swap operation in the strict sense (three elements are involved), but requires the same amount of work as a swap operation (in which we have to save the content of a variable into a temporary variable [Sed75]). An intermediate step in the partitioning algorithm is depicted in Figure A.1.

The algorithm works as follows: Using `i` it scans the input from left to right until it has found a large element, always comparing to the larger pivot first. Small elements found in this way are moved to a correct final position using the hole at array position $\mathtt{i}_1$. Subsequently, using `j` it scans the input from right to left until it has found a small element, always comparing to the smaller pivot first. Large elements found in this way are moved to a correct final position using the hole at array position $\mathtt{j}_1$. Now it exchanges the two elements at positions `i` resp. `j` and continues until `i` and `j` have met.

Algorithm 9 shows an implementation of the modified partitioning strategy from Section 4.1. In the same way as Algorithm 8 it scans the input from left to right until it has found a large element. However, it uses the smaller pivot for the first comparison in this part. Subsequently, it scans the input from right to left until it has found a small element. Here, it uses the larger pivot for the first comparison.

A.2.5. Algorithms for the Sampling Partitioning Method

The sampling method $\mathcal{SP}$ from Section 4.2 uses a mix of two classification algorithms: *"Always compare to the smaller pivot first"*, and *"Always compare to the larger pivot first"*. The actual partitioning method uses Algorithm 7 for the first $sz = n/1024$ classifications and then decides which pivot should be used for the first comparison in the remaining input. (This is done by comparing the two variables `i` and `k` in Algorithm 7.) If there are more large elements than small elements in the sample it continues using Algorithm 7, otherwise it uses Algorithm 10 below.

A.2.6. Algorithm for the Counting Strategy

Algorithm 11 is an implementation of the counting strategy from Section 6.4. It uses a variable `d` which stores the difference of the number of small elements and the number of large elements which have been classified so far. On average this algorithm makes $1.8n \ln n + O(n)$ comparisons and $1.6n \ln n$ assignments.

Algorithm 8 Sedgewick's Partitioning Method

procedure *S-Partition*(A, p, q, *left*, *right*, pos_p, pos_q)

```
1  i ← i_1 ← left; j ← j_1 := right;
2  while true do
3      i ← i + 1;
4      while A[i] ≤ q do
5          if i ≥ j then
6              break outer while
7          if A[i] < p then
8              A[i_1] ← A[i]; i_1 ← i_1 + 1; A[i] ← A[i_1];
9          i ← i + 1;
10     j ← j − 1;
11     while A[j] ≥ p do
12         if A[j] > q then
13             A[j_1] ← A[j]; j_1 ← j_1 − 1; A[j] ← A[j_1];
14         if i ≥ j then
15             break outer while
16         j ← j − 1;
17     A[i_1] ← A[j]; A[j_1] ← A[i];
18     i_1 ← i_1 + 1; j_1 ← j_1 − 1;
19     A[i] ← A[i_1]; A[j] ← A[j_1];
20 A[i_1] ← p; A[j_1] ← q;
21 pos_p ← i_1; pos_q ← j_1;
```

Algorithm 9 Sedgewick's Partitioning Method, modified

procedure *S2-Partition*(A, p, q, *left*, *right*, pos_p, pos_q)

```
1  i ← i_1 ← left; j ← j_1 := right;
2  while true do
3      i ← i + 1;
4      while true do
5          if i ≥ j then
6              break outer while
7          if A[i] < p then
8              A[i_1] ← A[i]; i_1 ← i_1 + 1; A[i] ← A[i_1];
9          else if A[i] > q then
10             break inner while
11         i ← i + 1;
12     j ← j − 1;
13     while true do
14         if A[j] > q then
15             A[j_1] ← A[j]; j_1 ← j_1 − 1; A[j] ← A[j_1];
16         else if A[j] < p then
17             break inner while
18         if i ≥ j then
19             break outer while
20         j ← j − 1;
21     A[i_1] ← A[j]; A[j_1] ← A[i];
22     i_1 ← i_1 + 1; j_1 ← j_1 − 1;
23     A[i] ← A[i_1]; A[j] ← A[j_1];
24 A[i_1] ← p; A[j_1] ← q;
25 pos_p ← i_1; pos_q ← j_1;
```

Algorithm 10 Simple Partitioning Method (smaller pivot first)

procedure *SimplePartitionSmall*(A, p, q, *left*, *right*, pos_p, pos_q)

```
1  l ← left + 1; g ← right − 1; k ← l;
2  while k ≤ g do
3      if A[k] < p then
4          swap A[k] and A[l];
5          l ← l + 1;
6          k ← k + 1;
7      else
8          if A[k] < q then
9              k ← k + 1;
10         else
11             swap A[k] and A[g]
12             g ← g − 1;
13 swap A[left] and A[l − 1];
14 swap A[right] and A[g + 1];
15 pos_p ← l − 1; pos_q ← g + 1;
```

A.3. A Fast Three-Pivot Algorithm

We give here the complete pseudocode for the three-pivot algorithm described in [Kus+14]. In contrast to the pseudocode given in [Kus+14, Algorithm A.1.1], we removed two unnecessary bound checks (Line 5 and Line 10 in our code) and we move misplaced elements in Lines 15–29 using less assignments. (This is used in the implementation of [Kus+14], as well.[1]) On average, this algorithm makes $1.846n \ln n + O(n)$ comparisons and $1.57n \ln n + O(n)$ assignments.

[1] Personal communication with Alejandro López-Ortiz.

Algorithm 11 Counting Strategy $\mathcal{C}$

procedure *Counting-Partition*(A, p, q, *left*, *right*, pos_p, pos_q)

```
1  i ← left + 1; k ← right − 1; j ← i;
2  d ← 0;                          ▷ d holds the difference of the number of small and large elements.
3  while j ≤ k do
4      if d > 0 then
5          if A[j] < p then
6              swap A[i] and A[j];
7              i ← i + 1; j ← j + 1; d ← d + 1;
8          else
9              if A[j] < q then
10                 j ← j + 1;
11             else
12                 swap A[j] and A[k];
13                 k ← k − 1; d ← d − 1;
14     else
15         while A[k] > q do
16             k ← k − 1; d ← d − 1;
17         if j ≤ k then
18             if A[k] < p then
19                 rotate3(A[k], A[j], A[i]);
20                 i ← i + 1; d ← d + 1;
21             else
22                 swap A[j] and A[k];
23             j ← j + 1;
24 swap A[left] and A[i − 1];
25 swap A[right] and A[k + 1];
26 pos_p ← i − 1; pos_q ← k + 1;
```

Algorithm 12 Symmetric Three-Pivot Algorithm ([Kus+14])

procedure *3-Pivot*(*A*, *left*, *right*)

Require: $right - left \geq 2$, $A[left] \leq A[left+1] \leq A[right]$

```
1  p_1 ← A[left]; p_2 ← A[left + 1]; p_3 ← A[right];
2  i ← left + 2; j ← i;
3  k ← right − 1; l ← k;
4  while j ≤ k do
5      while A[j] < p_2 do
6          if A[j] < p_1 then
7              swap A[i] and A[j];
8              i ← i + 1;
9          j ← j + 1;
10     while A[k] > p_2 do
11         if A[k] > p_3 then
12             swap A[k] and A[l];
13             l ← l − 1;
14         k ← k − 1;
15     if j ≤ k then
16         if A[j] > p_3 then
17             if A[k] < p_1 then
18                 rotate4(A[j], A[i], A[k], A[l]);
19                 i ← i + 1;
20             else
21                 rotate3(A[j], A[k], A[l]);
22             l ← l − 1;
23         else
24             if A[k] < p_1 then
25                 rotate3(A[j], A[i], A[k]);
26                 i ← i + 1;
27             else
28                 swap A[j] and A[k];
29         j ← j + 1; k ← k − 1;
30 rotate3(A[left + 1], A[i − 1], A[j − 1]);
31 swap A[left] and A[i − 2];
32 swap A[right] and A[l − 1];
33 3-Pivot(A, left, i − 3);
34 3-Pivot(A, i − 1, j − 2);
35 3-Pivot(A, j, l);
36 3-Pivot(A, l + 2, right);
```

B. Details of k-pivot Quicksort Experiments

This section contains detailed measurements obtained in the running time experiments for multi-pivot quicksort.

B.1. Guesses On The Running Time Influence of Memory Accesses

With the results from Section 7 in connection with the penalties for cache misses from [Lev09] and TBL misses from [HP12], we can calculate the *average number of cycles used for accessing memory* (*AVGCAM*). Let L1-P, L2-P, and L3-P denote the penalty (in clock cycles) for a miss in the L1, L2, and L3 cache, respectively. Let TLB-P be the penalty for a miss in the TLB. Let HT be the hit time of the L1 cache. For a multi-pivot quicksort algorithm which makes on average $\mathrm{E}(\mathrm{MA}_n)$ memory accesses to sort an input of n elements, we may then calculate:

$$\begin{aligned}\text{AVGCAM} = \text{HT} \cdot \mathrm{E}\left(\mathrm{MA}_n\right) + \text{avg. \#L1-Misses} \cdot \text{L1-P} + \text{avg. \#L2-Misses} \cdot \text{L2-P} \\ + \text{avg. \#L3-Misses} \cdot \text{L3-P} + \text{avg. \#TLB-Misses} \cdot \text{TLB-P}. \end{aligned} \tag{B.1}$$

For the Intel i7 used in the experiments, the characteristic numbers are:

- HT $= 4$ cycles [Lev09]
- L1-P $= 7$ cycles , L2-P $= 14$ cycles , L3-P $= 120$ cycles [Lev09]
- TBL-P $= 10$ cycles [HP12]. (We could not find exact numbers for the Intel i7.)

Using the detailed values from our experiments (see the tables on the following pages), Table B.1 shows the results with respect to this cost measure. Note that all values expect for the average number of memory accesses are based on experimental measurements.

B.2. Detailed Figures From Experiments

The following pages contain exact measurements from the multi-pivot quicksort running time experiments. Table B.2 contains figures for the cache misses of the considered algorithms. Table B.3 shows the TLB misses. Table B.4 consists of measurements with respect to branch mispredictions, executed instructions, and required CPU clock cycles.

Algorithm	AVGCAM
Permute_{127}	$4.58n \ln n$ (15.9%)
$\text{Permute}'_{127}$	$7.18n \ln n$ (57.3%)
Permute_{15}	$4.54n \ln n$ (14.7%)
$\text{Permute}'_{15}$	$8.15n \ln n$ (54.9%)
Permute_{256}	$4.31n \ln n$ (14.3%)
$\text{Permute}'_{256}$	$6.7n \ln n$ (48.2%)
Permute_{31}	$3.69n \ln n$ (13.3%)
$\text{Permute}'_{31}$	$8.95n \ln n$ (61.9%)
Permute_{512}	$4.2n \ln n$ (12.2%)
$\text{Permute}'_{512}$	$7.07n \ln n$ (43.1%)
Permute_{7}	$6.47n \ln n$ (21.6%)
$\text{Permute}'_{7}$	$10.98n \ln n$ (64.5%)
Exchange_1	$10.63n \ln n$ (64.3%)
$\mathcal{Y}$	$8.26n \ln n$ (53.8%)
$\mathcal{L}$	$8.28n \ln n$ (54.2%)
Exchange_3	$7.16n \ln n$ (47.1%)
Exchange_5	$6.96n \ln n$ (42.1%)
Exchange_7	$7.28n \ln n$ (41.4%)
Exchange_9	$7.75n \ln n$ (42.8%)
Copy'_{127}	$9.91n \ln n$ (91.9%)
Copy'_{255}	$9.26n \ln n$ (77.3%)
Copy'_{511}	$8.74n \ln n$ (65.1%)

Table B.1.: Average number of cycles used for accessing memory for the algorithms considered in our experiments. The value in parentheses shows the ratio of the average number of cycles needed for memory accesses and the average number of cycles needed for sorting.

Algorithm	avg. number of L1 misses	avg. number of L2 misses	avg. number of L3 misses
Permute_{127}	$0.0496n \ln n$ (0.0%)	$0.0165n \ln n$ (521.4%)	$0.0128n \ln n$ (764.2%)
$\text{Permute}'_{127}$	$0.0666n \ln n$ (34.4%)	$0.0175n \ln n$ (561.5%)	$0.0143n \ln n$ (861.8%)
Permute_{15}	$0.0728n \ln n$ (46.9%)	$0.0033n \ln n$ (22.8%)	$0.0018n \ln n$ (21.6%)
$\text{Permute}'_{15}$	$0.0867n \ln n$ (74.8%)	$0.0059n \ln n$ (121.7%)	$0.0022n \ln n$ (45.5%)
Permute_{256}	$0.0578n \ln n$ (16.5%)	$0.0152n \ln n$ (473.2%)	$0.0113n \ln n$ (662.1%)
$\text{Permute}'_{256}$	$0.0934n \ln n$ (88.4%)	$0.017n \ln n$ (542.1%)	$0.0126n \ln n$ (751.1%)
Permute_{31}	$0.0594n \ln n$ (19.9%)	$0.0027n \ln n$ (0.0%)	$0.0015n \ln n$ (0.0%)
$\text{Permute}'_{31}$	$0.0707n \ln n$ (42.8%)	$0.0216n \ln n$ (714.7%)	$0.0159n \ln n$ (969.4%)
Permute_{512}	$0.0874n \ln n$ (76.4%)	$0.017n \ln n$ (540.8%)	$0.0097n \ln n$ (553.0%)
$\text{Permute}'_{512}$	$0.1691n \ln n$ (241.2%)	$0.0206n \ln n$ (676.1%)	$0.0128n \ln n$ (760.6%)
Permute_{7}	$0.0969n \ln n$ (95.5%)	$0.0039n \ln n$ (45.7%)	$0.0024n \ln n$ (60.2%)
$\text{Permute}'_{7}$	$0.1153n \ln n$ (132.5%)	$0.0098n \ln n$ (270.7%)	$0.003n \ln n$ (104.0%)
Exchange_{1}	$0.1472n \ln n$ (197.0%)	$0.02.8n \ln n$ (869.9%)	$0.0076n \ln n$ (409.4%)
$\mathcal{Y}$	$0.1166n \ln n$ (135.3%)	$0.0124n \ln n$ (366.8%)	$0.0041n \ln n$ (178.6%)
$\mathcal{C}$	$0.1166n \ln n$ (135.2%)	$0.0115n \ln n$ (334.6%)	$0.0036n \ln n$ (144.6%)
$\mathcal{SP}$	$0.1166n \ln n$ (135.2%)	$0.0125n \ln n$ (372.9%)	$0.0042n \ln n$ (181.8%)
$\mathcal{L}$	$0.1166n \ln n$ (135.2%)	$0.0133n \ln n$ (400.4%)	$0.0042n \ln n$ (182.3%)
Exchange_{3}	$0.101n \ln n$ (103.7%)	$0.0093n \ln n$ (250.4%)	$0.0033n \ln n$ (119.4%)
Exchange_{5}	$0.0995n \ln n$ (100.6%)	$0.0073n \ln n$ (177.2%)	$0.0021n \ln n$ (44.5%)
Exchange_{7}	$0.1041n \ln n$ (110.1%)	$0.0075n \ln n$ (184.6%)	$0.002n \ln n$ (32.5%)
Exchange_{9}	$0.1104n \ln n$ (122.8%)	$0.0083n \ln n$ (213.2%)	$0.0021n \ln n$ (39.9%)
Copy'_{127}	$0.1183n \ln n$ (138.6%)	$0.0507n \ln n$ (1811.4%)	$0.0226n \ln n$ (1424.5%)
Copy'_{255}	$0.1221n \ln n$ (146.4%)	$0.0443n \ln n$ (1572.0%)	$0.0236n \ln n$ (1490.2%)
Copy'_{511}	$0.1347n \ln n$ (171.7%)	$0.0465n \ln n$ (1653.7%)	$0.0233n \ln n$ (1469.7%)
ssortv1	$0.0638n \ln n$ (28.7%)	$0.0179n \ln n$ (574.2%)	$0.0145n \ln n$ (879.6%)
stdsort	$0.1355n \ln n$ (173.4%)	$0.0197n \ln n$ (644.3%)	$0.0064n \ln n$ (331.7%)

Table B.2.: Measurements of L1, L2, and L3 cache misses of the algorithms considered in the experiments for $n = 2^{27}$ items. All values are scaled by $n \ln n$ and averaged over 500 trials. The value in parentheses shows the ratio of the specific cost and the lowest value in the respective cost measure.

Algorithm	avg. TLB misses
Permute$_{127}$	$0.0716n \ln n$ (76.0%)
Permute$'_{127}$	$0.1203n \ln n$ (195.7%)
Permute$_{15}$	$0.0416n \ln n$ (2.2%)
Permute$'_{15}$	$0.0498n \ln n$ (22.4%)
Permute$_{256}$	$0.0819n \ln n$ (101.4%)
Permute$'_{256}$	$0.1226n \ln n$ (201.3%)
Permute$_{31}$	$0.0462n \ln n$ (13.5%)
Permute$'_{31}$	$0.0712n \ln n$ (75.0%)
Permute$_{512}$	$0.0873n \ln n$ (114.5%)
Permute$'_{512}$	$0.1401n \ln n$ (244.4%)
Permute$_{7}$	$0.082n \ln n$ (101.6%)
Permute$'_{7}$	$0.0421n \ln n$ (3.5%)
Exchange$_1$	$0.0407n \ln n$ (0.0%)
$\mathcal{Y}$	$0.0413n \ln n$ (1.4%)
$\mathcal{L}$	$0.041n \ln n$ (0.9%)
Exchange$_3$	$0.041n \ln n$ (0.9%)
Exchange$_5$	$0.0412n \ln n$ (1.3%)
Exchange$_7$	$0.0409n \ln n$ (0.6%)
Exchange$_9$	$0.0412n \ln n$ (1.2%)
Copy$'_{127}$	$0.0412n \ln n$ (1.2%)
Copy$'_{255}$	$0.0411n \ln n$ (1.1%)
Copy$'_{511}$	$0.0411n \ln n$ (0.9%)

Table B.3.: Average number of load misses in the translation-lookaside buffer for $n = 2^{27}$ scaled by $n \ln n$. Numbers are averaged over 100 trials.

Algorithm	avg. branch mispred. count	avg. instruction count	avg. cycle count
Permute_{127}	$0.1298n \ln n$ (5.7%)	$22.817n \ln n$ (143.1%)	$28.7709n \ln n$ (166.9%)
$\text{Permute}'_{127}$	$0.1569n \ln n$ (27.9%)	$11.2769n \ln n$ (20.1%)	$12.5178n \ln n$ (16.1%)
Permute_{15}	$0.163n \ln n$ (32.8%)	$30.5548n \ln n$ (225.5%)	$30.7798n \ln n$ (185.5%)
$\text{Permute}'_{15}$	$0.1665n \ln n$ (35.7%)	$19.1643n \ln n$ (104.2%)	$14.852n \ln n$ (37.8%)
Permute_{256}	$0.1383n \ln n$ (12.7%)	$22.7312n \ln n$ (142.2%)	$30.1065n \ln n$ (179.2%)
$\text{Permute}'_{256}$	$0.1285n \ln n$ (4.7%)	$12.2017n \ln n$ (30.0%)	$13.9035n \ln n$ (29.0%)
Permute_{31}	$0.1345n \ln n$ (9.6%)	$27.0696n \ln n$ (188.4%)	$27.646n \ln n$ (156.4%)
$\text{Permute}'_{31}$	$0.133n \ln n$ (8.4%)	$16.2657n \ln n$ (73.3%)	$14.4689n \ln n$ (34.2%)
Permute_{512}	$0.1688n \ln n$ (37.5%)	$25.5953n \ln n$ (172.7%)	$34.4346n \ln n$ (219.4%)
$\text{Permute}'_{512}$	$0.1457n \ln n$ (18.7%)	$12.4341n \ln n$ (32.5%)	$16.3872n \ln n$ (52.0%)
Permute_{63}	$0.128n \ln n$ (4.3%)	$24.5402n \ln n$ (161.4%)	$30.138n \ln n$ (179.5%)
Permute_{7}	$0.3516n \ln n$ (186.5%)	$27.4562n \ln n$ (192.5%)	$29.9299n \ln n$ (177.6%)
$\text{Permute}'_{7}$	$0.3574n \ln n$ (191.2%)	$17.5678n \ln n$ (87.2%)	$17.0291n \ln n$ (58.0%)
qsort1	$0.5985n \ln n$ (387.8%)	$11.1845n \ln n$ (19.2%)	$17.4641n \ln n$ (62.0%)
Exchange_{1}	$0.5695n \ln n$ (364.1%)	$10.6444n \ln n$ (13.4%)	$16.5432n \ln n$ (53.4%)
qsort1sentinels	$0.6156n \ln n$ (401.7%)	$11.1655n \ln n$ (19.0%)	$17.7468n \ln n$ (64.6%)
qsort1sn	$0.5698n \ln n$ (364.4%)	$10.1945n \ln n$ (8.6%)	$16.4503n \ln n$ (52.6%)
$\mathcal{Y}$	$0.5722n \ln n$ (366.4%)	$10.4313n \ln n$ (11.1%)	$15.3437n \ln n$ (42.3%)
$\mathcal{C}$	$0.5997n \ln n$ (388.7%)	$14.0916n \ln n$ (50.1%)	$17.4701n \ln n$ (62.0%)
qsort2v3	$0.5911n \ln n$ (381.7%)	$11.4412n \ln n$ (21.9%)	$16.737n \ln n$ (55.2%)
$\mathcal{SP}$	$0.5881n \ln n$ (379.3%)	$10.0664n \ln n$ (7.2%)	$16.0601n \ln n$ (49.0%)
$\mathcal{L}$	$0.5685n \ln n$ (363.3%)	$9.5129n \ln n$ (1.3%)	$15.2785n \ln n$ (41.7%)
Exchange_{3}	$0.5832n \ln n$ (375.3%)	$9.3864n \ln n$ (0.0%)	$15.1875n \ln n$ (40.9%)
Exchange_{5}	$0.6527n \ln n$ (431.9%)	$10.5238n \ln n$ (12.1%)	$16.5401n \ln n$ (53.4%)
Exchange_{7}	$0.6631n \ln n$ (440.4%)	$10.5193n \ln n$ (12.1%)	$17.5827n \ln n$ (63.1%)
Exchange_{9}	$0.6573n \ln n$ (435.7%)	$11.201n \ln n$ (19.3%)	$18.0916n \ln n$ (67.8%)
Copy'_{127}	$0.1528n \ln n$ (24.5%)	$10.4937n \ln n$ (11.8%)	$10.7813n \ln n$ (0.0%)
Copy'_{255}	$0.1227n \ln n$ (0.0%)	$11.4709n \ln n$ (22.2%)	$11.9893n \ln n$ (11.2%)
Copy'_{511}	$0.1891n \ln n$ (54.1%)	$11.9471n \ln n$ (27.3%)	$13.4204n \ln n$ (24.5%)
ssortv1	$0.1635n \ln n$ (33.2%)	$10.8589n \ln n$ (15.7%)	$13.084n \ln n$ (21.4%)
stdsort	$0.5883n \ln n$ (379.5%)	$10.2882n \ln n$ (9.6%)	$16.6929n \ln n$ (54.8%)

Table B.4.: Measurements for inputs containing $n = 2^{27}$ items of the average number of branch mispredictions, the average number of executed instructions, and the average number of CPU cycles required by the algorithms considered in the experiments. All values are scaled by $n \ln n$ and averaged over 500 trials. The value in parentheses shows the ratio to the minimum cost with respect to the cost measure over all algorithms.

List of Figures

2.1 Result of the partition step in dual-pivot quicksort schemes using two pivots p, q with $p \leq q$. Elements left of p are smaller than or equal to p, elements right of q are larger than or equal to q. The elements between p and q are at least as large as p and at most as large as q. 10

3.1 An example for a decision tree to classify three elements a_2, a_3, and a_4 according to the pivots a_1 and a_5. Five out of the 27 leaves are explicitly drawn, showing the classification of the elements and the costs c_i of the specific paths. 20

4.1 Visualization of the decision process when inspecting an element using strategy $\mathcal{O}$ (left) and $\mathcal{C}$ (right). Applying strategy $\mathcal{O}$ from left to right uses that of the remaining elements three are small and one is large, so it decides that the element should be compared with p first. Applying strategy $\mathcal{C}$ from right to left uses that of the inspected elements two were small and only one was large, so it decides to compare the element with p first, too. Note that the strategies would differ if, e. g., the right-most element would be a medium element. 36

4.2 Average comparison count (scaled by $n \ln n$) needed to sort a random input of up to $n = 2^{29}$ integers. We compare classical quicksort ($\mathcal{QS}$), Yaroslavskiy's algorithm ($\mathcal{Y}$), the optimal sampling algorithm ($\mathcal{N}$), the optimal counting algorithm ($\mathcal{C}$), the modified version of Sedgewick's algorithm ($\mathcal{S}'$), and the simple strategy *"always compare to the larger pivot first"* ($\mathcal{L}$). Each data point is the average over 400 trials. 38

6.1 Result of the partition step in k-pivot quicksort using pivots $p_1, \ldots, p_k$. 46

6.2 A comparison tree for 5 pivots. 50

6.3 The different comparison trees for 3-pivot quicksort with their comparison cost (dotted boxes, only displaying the numerator). 54

7.1 General memory layout of Algorithm 1 for $k = 2$. 71

7.2 Top: Example for the cyclic rotations occurring in one round of Algorithm 1 starting from the example given in Figure 7.1. First, the algorithm finds an A_2-element, which is then moved into the A_2-segment (1.), replacing an A_1-element which is moved into the A_1-segment (2.). It replaces an A_2-element that is moved to replace the next misplaced element in the A_2-segment, an A_0 element (3.). This element is then moved to the A_0-segment (4.), overwriting the misplaced A_2-element, which ends the round. Bottom: Memory layout and offset indices after moving the elements from the example. 71

7.3 General memory layout of Algorithm 3 for $k = 6$. Two pointers i and j are used to scan the array from left-to-right and right-to-left, respectively. Pointers $g_1, \ldots, g_{k-1}$ are used to point to the start of segments. 72

7.4 The `rotate` operation in Line 8 of Algorithm 3. An element that belongs to group A_1 is moved into its respective segment. Pointers i, g_2, g_3 are increased by 1 afterwards. 72

7.5 The `rotate` operation in Line 13 of Algorithm 3. An element that belongs to group A_6 is moved into its respective segment. Pointers j, g_4, g_5 are decreased by 1 afterwards. 72

7.6 Example for the rotate operation in Line 17 of Algorithm 3. The element found at position i is moved into its specific segment. Subsequently, the element found at position j is moved into its specific segment. 72

7.7 The average number of assignments for sorting an input consisting of n elements using Algorithm 1 ("Permute$_k$") and Algorithm 3 ("Exchange$_k$") for certain values of k. Each data point is the average over 600 trials. 79

7.8 TLB misses for Algorithms 1–3. Each data point is averaged over 500 trials, TLB load misses are scaled by $n \ln n$. 86

8.1 Running time experiments for dual-pivot quicksort algorithms. Each data point is the average over 500 trials. Times are scaled by $n \ln n$. 89

8.2 The comparison trees used for the 5-,7-, and 9-pivot algorithms. 90

8.3 Running time experiments for k-pivot quicksort algorithms based on the "Exchange$_k$" partitioning strategy. Each data point is the average over 500 trials. Times are scaled by $n \ln n$. 90

8.4 Running time experiments for k-pivot quicksort algorithms based on the "Permute$_k$" partitioning algorithm. Each data point is the average over 500 trials. Times are scaled by $n \ln n$. 92

8.5 Final Running time experiments for k-pivot quicksort algorithms based in C++. Each data point is the average over 500 trials. Times are scaled by $n \ln n$. 93

11.1 The minimal obstruction graphs for cuckoo hashing. 120

13.1 An example of a graph that contains an excess-3 core graph (bold edges). This subgraph certifies that a stash of size at most 2 does not suffice to accommodate the key set. This figure can also be found in [Aum10]. 133
13.2 Example for the construction of a perfect hash function 141
13.3 Comparison of the probability of a random graph being acyclic and the theoretical bound following from a first moment approach for values $\varepsilon \in [0.08, 4]$. . . . 142

14.1 Structure of a witness tree T_t with root v after t rounds if the τ-collision protocol with d candidate machines has not yet terminated. 167

16.1 Level 1 cache misses for building a cuckoo hash table from inputs $\{1, \ldots, n\}$ with a particular hash function. Each data point is the average over 10,000 trials. Cache Misses are scaled by n. 187
16.2 Level 2 cache misses for building a cuckoo hash table from inputs $\{1, \ldots, n\}$ with a particular hash function. Each data point is the average over 10,000 trials. Cache Misses are scaled by n. 187

A.1 An intermediate partitioning step in Sedgewick's algorithm. 209

List of Tables

5.1 Comparison of the leading term of the average cost of classical quicksort and dual-pivot quicksort for specific sample sizes. Note that for real-life input sizes, however, the linear term can make a big difference. 41

6.1 Optimal average comparison count for k-pivot quicksort for $k \in \{2, \dots, 9\}$. For $k \geq 4$ these numbers are based on experiments). For odd k, we also include the average comparison count of quicksort with the median-of-k strategy. (The numbers for the median-of-k variant can be found in [Emd70] or [Hen91].) . . . 61

7.1 Average number of assignments for partitioning ($\mathrm{E}(\mathrm{PAS}_n)$) and average number of assignments for sorting ($\mathrm{E}(\mathrm{AS}_n)$) an array of length n disregarding lower order terms using Algorithm 1, Algorithm 2, and Algorithm 3. We did not calculate the average comparison count for Algorithm 3 for $k \in \{15, 31, 63, 127\}$. For comparison, note that classical quicksort ("$\mathrm{Exchange}_1$") makes $n \ln n$ assignments involving array accesses on average. 78

7.2 Average number of memory accesses for partitioning ($\mathrm{E}(\mathrm{PMA}_n)$) and average number of memory accesses for sorting an array ($\mathrm{E}(\mathrm{MA}_n)$) of length n disregarding lower order terms. Note that classical quicksort makes $2n \ln n$ memory accesses on average. 81

7.3 Cache misses incured by Algorithm 1 ("Perm_k") and Algorithm 3 ("Ex_k") in a single partitioning step. All values are averaged over 600 trials. 83

7.4 Average number of L1 cache misses compared to the average number of memory accesses. Measurements have been obtained for $n = 2^{27}$. Cache misses are scaled by $n \ln n$. In parentheses, we show the ratio to the best algorithmic variant of Algorithm 3 w.r.t. memory/cache behavior ($k = 5$), calculated from the non-truncated experimental data. 84

7.5 Average number of TLB misses for random inputs with 2^{27} items over 100 trials. Load misses are scaled by $n \ln n$. The number in parentheses shows the relative difference to algorithm $\mathrm{Exchange}_1$. 85

8.1 Overview of the dual-pivot quicksort algorithms considered in the experiments. 88

8.2 Comparison of the actual running times of the algorithms on 500 different inputs of size 2^{27}. A table cell in a row labeled "A" and a column labeled "B" contains a string "$x\%/y\%/z\%$" and is read as follows: "In about 95%, 50%, and 5% of the cases algorithm A was more than x, y, and z percent faster than algorithm B, respectively." 91

14.1 Space utilization thresholds for generalized cuckoo hashing with $d \geq 3$ hash functions and $\kappa + 1$ keys per cell, for $\kappa \geq 1$, based on the non-existence of the $(\kappa + 1)$-core. Each table cell gives the maximal space utilization achievable for the specific pair $(d, \kappa + 1)$. These values have been obtained using Maple® to evaluate the formula from Theorem 1 of [Mol05]. 176

16.1 Hash Families considered in our experiments. The "identifier" is used to refer to the constructions in the charts and in the text. 184
16.2 Maximum stash size s for structured inputs of $n = 2^{20}$ elements with $m = 1.005n$. 185
16.3 Different running times for the construction of a cuckoo hash table for $m = 1.005n$. Each entry is the average over 10,000 trials. The last row contains the measurements for the structured input. 186
16.4 Different running times for the construction of a cuckoo hash table for $m = 1.05n$. Each entry is the average over 10,000 trials. The last row contains the measurements for the structured input type. 186

B.1 Average number of cycles used for accessing memory for the algorithms considered in our experiments. The value in parentheses shows the ratio of the average number of cycles needed for memory accesses and the average number of cycles needed for sorting. 216
B.2 Measurements of L1, L2, and L3 cache misses of the algorithms considered in the experiments for $n = 2^{27}$ items. All values are scaled by $n \ln n$ and averaged over 500 trials. The value in parentheses shows the ratio of the specific cost and the lowest value in the respective cost measure. 217
B.3 Average number of load misses in the translation-lookaside buffer for $n = 2^{27}$ scaled by $n \ln n$. Numbers are averaged over 100 trials. 218
B.4 Measurements for inputs containing $n = 2^{27}$ items of the average number of branch mispredictions, the average number of executed instructions, and the average number of CPU cycles required by the algorithms considered in the experiments. All values are scaled by $n \ln n$ and averaged over 500 trials. The value in parentheses shows the ratio to the minimum cost with respect to the cost measure over all algorithms. 219

List of Algorithms

1 Permute elements to produce a partition 73
2 Copy elements to produce a partition 74
3 Move elements by rotations to produce a partition 75

4 The quicksort algorithm 205
5 Dual-Pivot-Quicksort (outline) 206
6 Yaroslavskiy's Partitioning Method 207
7 Always Compare To Larger Pivot First Partitioning 208
8 Sedgewick's Partitioning Method 210
9 Sedgewick's Partitioning Method, modified 211
10 Simple Partitioning Method (smaller pivot first) 212
11 Counting Strategy $\mathcal{C}$ 213
12 Symmetric Three-Pivot Algorithm ([Kus+14]) 214

Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberaterinnen oder anderen Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer Prüfungsbehörde vorgelegt.

Ich bin darauf hingewiesen worden, dass die Unrichtigkeit der vorstehenden Erklärung als Täuschungsversuch bewertet wird und gemäß §7 Absatz 10 der Promotionsordnung den Abbruch des Promotionsverfahrens zur Folge hat.

Ilmenau, den 1. Dezember 2014

Martin Aumüller