Karlsruhe Institute of Technology

Constructing Cuckoo hash tables independent of their load-factor

Bachelor's Thesis of

Henning Schulze

at the Department of Informatics
Institute of Theoretical Informatics, Algorithmics II

Supervisors: Prof. Dr. rer. nat. Peter Sanders
M.Sc. Tobias Maier

11. September 2017 – 09. November 2017

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, 09.11.2017

. . . .

(Henning Schulze)

Abstract

A bucketized Cuckoo hash table is a hash table which is divided into disjunctive buckets containing k elements each. Each element is stored in a bucket dictated by one of two hash functions. Cuckoo hash tables guarantee a constant worst-case access-time. Bucketized Cuckoo hash tables additionally support very high load-factors. This thesis presents multiple algorithms to construct bucketized Cuckoo hash tables, based on the Selfless(k)-algorithm [CSW07]. Our algorithms work directly on the table. Thereby, we achieve algorithms using sublinear memory and whose running time is independent of the load-factor. We also present highly efficient parallel versions of our algorithms. Additionally, we combine the d-ary Cuckoo hashing [Fot+03] approach with bucket Cuckoo hashing by adapting the Selfless(k)-algorithm [CSW07] to be used with more than two hash functions. We show the performance in a number of benchmarks and compare our algorithms to a growing and non-growing state of the art iterated insertion algorithm.

Zusammenfassung

Eine bucketized Cuckoo Hashtabelle ist eine Hashtabelle, welche in disjunkte Zellgruppen unterteilt ist, die jeweils k Elemente enthalten. Die Position eines jeden Elementes wird durch eine von zwei Hashfunktionen bestimmt. Cuckoo Hashtabellen garantieren eine konstante worst-case Zugriffszeit. Bucketized Cuckoo Hashtabellen ermöglichen zusätzlich sehr hohe Füllgrade. Diese Abschlussarbeit präsentiert verschiedene Algorithmen, abgeleitet vom Selfless(k)-Algorithmus [CSW07], um bucketized Cuckoo Hashtabellen zu konstruieren. Unsere Algorithmen operieren direkt auf der Tabelle. Dadurch erzielen wir Algorithmen mit einen sublinearen Speicherplatzverbrauch und einer Laufzeit, welche unabhängig vom Füllgrad ist. Ebenso präsentieren wir hocheffiziente parallele Versionen unserer Algorithmen. Zudem kombinieren wir den d-ary Cuckoo hashing [Fot+03] Ansatz mit bucket Cuckoo hashing, indem wir den Selfless(k)-Algorithmus [CSW07] anpassen, so dass dieser mit mehr als zwei Hashfunktionen verwendet werden kann. Wir zeigen die Performance unserer Algorithmen unter verschiedenen Aspekten und vergleichen sie mit einem aktuellen wachsenden und nicht-wachsenden iterated insertion Algorithmus.

Contents

Abstract i

Zusammenfassung iii

1 Introduction 1
- 1.1 Problem . . . 1
- 1.2 Motivation . . . 1
- 1.3 General Introduction . . . 2
- 1.4 Related work . . . 2
- 1.5 Overview . . . 3

2 Preliminaries 5
- 2.1 Definitions . . . 5
- 2.2 The Selfless(k) algorithm . . . 6
- 2.3 Incremented bucketized Cuckoo hash table construction . . . 7

3 Algorithms 9
- 3.1 Definitions . . . 9
- 3.2 Base algorithm . . . 10
- 3.3 Priority effects . . . 11
- 3.4 Variants . . . 12
  - 3.4.1 Index . . . 12
  - 3.4.2 Hybrid . . . 12
  - 3.4.3 Inplace . . . 13
- 3.5 Multi Hash Functions . . . 16
- 3.6 Two level . . . 17

4 Implementation 19
- 4.1 Common structures . . . 19
- 4.2 Variants . . . 19
  - 4.2.1 Index . . . 19
  - 4.2.2 Hybrid . . . 24
  - 4.2.3 Inplace . . . 28
- 4.3 Multi Hash Functions . . . 30
- 4.4 Two level . . . 31
- 4.5 Parallelization . . . 32

5 Evaluation 33
- 5.1 Legend . . . 33

5.2 Common parameters/hardware specs 34
5.3 Running times 34
5.4 Load-factor independence 34
5.5 Memory consumption 37
5.6 Efficiency 39
5.7 Multi hash functions 39

6 Conclusion **43**
6.1 Overview 43
6.2 Future work 43

Bibliography **45**

List of Figures

3.1 Control flow of the inplace variant 13

4.1 Fixing by index 23
4.2 Fixed and prehashed arrays 24
4.3 Fixing the first prehashed element 26
4.4 Fixing a prehashed element by index 27

5.1 Legend 33
5.2 Variant running times 35
5.3 Level variant running times 35
5.4 Load-factor behaviour 36
5.5 Load-factor independence of our variants 36
5.6 Load-factor independence of our level variants 36
5.7 Memory consumption with the table 38
5.8 Memory consumption without the table 38
5.9 Efficiency 38

List of Tables

5.1 Error measurements of our different versions for $k = 8$ and $h = 3$ 40
5.2 Error measurements of our different versions for $k = 4$ and $h = 3$ 41

1 Introduction

1.1 Problem

A bucketized Cuckoo hash table is a hash table which is divided into disjunctive buckets containing k elements each. Each element is stored in a bucket dictated by one of two hash functions. We consider the *offline* case where all elements are available at the beginning. In our setup the elements are in a table without any order. Given two independent hash functions we want to construct a bucketized Cuckoo hash table out of the given table.

1.2 Motivation

Hash tables are ubiquitous data structures in the field of computer science. Basic hash table approaches such as separate chaining or open addressing have the problem that for higher load-factors the access-time does not remain constant and is even linear in the worst-case [Knu98]. Bucketized Cuckoo hash tables guarantee a constant worst-case access-time. Each element has to be in a bucket containing k elements dictated by one of two hash functions. Therefore, we have to check at most two entire buckets for each access resulting in a worst-case access-time of $O(2 \cdot k) = O(1)$.

Additionally, bucketized Cuckoo hash tables support very high load-factors, for $k = 2$ approximately 80%, for $k = 4$ approx. 97%, and for $k = 8$ approx. 99.7% [CSW07] [MS17]. Therefore, they are very space-efficient without deteriorating the access-time.

In Section 1.1 we presumed that our elements are already unordered in a table. We chose this setup because it has real-world applications: imagine a bucket Cuckoo hashing algorithm that fails. If it fails the table is nearly filled with elements. The common approach would be to allocate a new table, choose two new hash functions, copy the elements to the new table, deallocate the old table, and then run the algorithm again.

However, instead one could simply run our algorithm with two new hash functions on the table saving oneself the deallocation and allocation (and getting all elements in a form where they are ready to be inserted). The setup is exactly like ours: a table filled with elements without any order. For load-factors close to the respective threshold this scenario is likely.

Bucket Cuckoo hashing has practical usage in many different applications. Once constructed, a bucketized Cuckoo hash table guarantees a constant worst-case access-time. Therefore, it can be used in real-time systems where running time guarantees have to

be met. Additionally, they support very high load-factors which makes them relevant for devices with limited memory. Another application is parallel disk servers, we refer to [SEK03].

1.3 General Introduction

There are already existing implementations for bucket Cuckoo Hashing, for example our comparison algorithm introduced in Section 2.3. What separates us from other bucket Cuckoo Hashing algorithms is that we construct our hash table where as the other algorithms insert the elements one by one. Our approach has the characteristic that it moves elements, but once they are assigned they will not be moved again. Additionally, our algorithms are load-factor independent (and almost work inplace).

Constructing algorithms for bucket Cuckoo Hashing already exist but so far only in theory. We provide practical implementations. Our algorithms either apply the Selfless(k)-algorithm or are derived from it. The Selfless(k)-algorithm was introduced by Cain, Sanders, and Wormald in [CSW07] as a graph algorithm. We transformed it into a hashing algorithm.

We developed multiple variants that adapted the Selfless(k)-algorithm in different ways. Our first algorithm uses a full graph representation. Our next algorithm keeps the graph partly implicit using a prehashing that tries to predict in which buckets elements will be hashed into and then writes them in advance. We also have an algorithm keeping the graph fully implicit. It requires different iteration techniques for elements and slightly varying from the Selfless(k)-algorithm. However, it still adheres to the Selfless(k)-algorithm as much as possible.

A further variant we developed is an algorithm which uses a new hash function, dividing the table into subtables. It distributes all elements as equally as possible among those subtables. The subtables are then independent from each other. Therefore, we can construct them independently from each other achieving sublinear memory consumption. We also present a highly efficient parallelization of this part exploiting the independence of the subtables.

1.4 Related work

The basic approach to use two hash functions instead of one originates from the balls into bins game introduced in [Aza+99] where each ball is placed in the least full bin. Building on this approach Pagh and Rodler introduced Cuckoo hashing [PR01], which is the underlying principle of our hash problem. In a Cuckoo hash table the position of each element is dictated by one out of two hash functions. Elements are inserted into a position dictated by one out of two hash functions. If both positions are occupied a displacement strategy is needed. The easiest is to randomly pick one of the occupied positions, remove the element there, write the element to the freed-up position and reinsert the removed element.

Cuckoo hashing can generally be divided into two different settings, the *offline* case and the *online* case. In the *online* case a number of elements are inserted and deleted over a period of time. The insert or delete requests arrive separately. This thesis focuses on the *offline* case where all elements are available at the beginning (see Section 1.1 for our setup). A number of papers approach the *online* case, we refer to [DW05] [Fot+03] [FMM09].

The approach, to build space efficient hash tables by assigning elements to one out of two buckets of size k was introduced by Dietzfelbinger and Weidling in [DW05]. Building on this approach Cain, Sanders, and Wormald introduced the Selfless(k)-algorithm [CSW07], which was originally introduced in [San04], our algorithms and implementations are based on. However, the Selfless(k)-algorithm was described as a graph and disk scheduling algorithm solving the k-orientability problem [Kar98]. This thesis focuses on the hash table problem. Therefore, our implementation approach is different. The k-orientability problem is equivalent to bucket Cuckoo hashing (see Section 2.1). There have been other approaches to solve the k-orientability, for example the DEM algorithm [FR07] from Fernholz and Ramachandran.

Another approach developing Cuckoo hashing is d-ary Cuckoo hashing [Fot+03] from Fotakis, Pagh, Sanders, and Spirakis. Instead of introducing buckets it uses more than two hash functions. Similar to bucket Cuckoo hashing it supports very high load-factors and guarantees a constant worst-case access-time. One of our algorithms combines this approach with bucket Cuckoo hashing (see Section 3.5). There are also a lot of other approaches to construct hash tables or dictionaries with a constant worst-case access-time, we refer to [Die+94] [FKS82].

1.5 Overview

We begin this thesis with chapter 2 which covers important definitions, the Selfless(k)-algorithm our algorithms are based on and our comparison algorithms. We continue with chapter 3 where we present our notation of the Selfless(k)-algorithm. We then present various variations of the Selfless(k)-algorithm as well as derived algorithms that are very memory-efficient or enable the usage of multiple hash functions. We also offer a modification that can be used with any of the other variants using sublinear memory. chapter 4 contains implementation details of our algorithms, for example implicit data structures, behaviour exploits and different element assignment approaches as well as parallelizations. chapter 5 evaluates our algorithms regarding different benchmarks such as running time, memory consumption, load-factor independence, and efficiency. Lastly, chapter 6 draws a conclusion on our results and outlines future work, for instance external algorithms.

2 Preliminaries

In Section 2.1 we cover the definitions of the most important terms that are used throughout the thesis. Section 2.2 then presents the Selfless(k)-algorithm taken from [CSW07]. Lastly, Section 2.3 presents the iterated insertion algorithm we use for comparison in its growing and non-growing version.

2.1 Definitions

Throughout this thesis h_0 and h_1 are two independent hash functions. $\boldsymbol{G} := (V, E)$ represents an undirected graph where V is a set of vertices and E is a set of undirected edges between vertices in V. The corresponding graph $\boldsymbol{G'} := (V', E')$ represents a directed graph constructed from G where $V' = V$ and E' is constructed by directing a subset of edges in E. An undirected edge between $v, u \in V$ is represented via $\{v, u\}$ where as a directed edge from v to u for $v, u \in V'$ is represented via (v, u). For $v \in V$ the **degree** of v is defined as $\boldsymbol{deg(v)} := |\{\{u, v\} | \{u, v\} \in E\}|$, for $v \in V'$ the **in-degree** of v is defined as $\boldsymbol{deg_{in}(v)} := |\{(u, v) | (u, v) \in E'\}|$. In this thesis $deg(v)$ will **always refer** to $deg(v)$ in G and $deg_{in}(v)$ will **always refer** to $deg_{in}(v)$ in G'.

We use $\boldsymbol{m}$ as table-size and $\boldsymbol{n}$ as the number of elements throughout this thesis. The **load-factor** is the number of elements divided through the table-size ($\frac{n}{m}$). The load-factor has sharp thresholds. Above these thresholds, it is very likely that a bucket Cuckoo hashing is impossible because of unresolvable collisions (regardless of the used algorithm). Below these thresholds, the probability of unresolvable collisions is negligible. For $k = 4$ that threshold is approx. 97% and for $k = 8$ it is approx. 99.7% [CSW07] [MS17].

The **balanced allocation paradigm** [Aza+99]: Assign m balls to n bins where each ball can be assigned to one out of two random bins such that the maximum occupancy is minimized. An equivalent problem is the **Edge orientability problem**: Given a graph G, construct a graph G' by orienting all edges in G so that the maximum in-degree in V' is minimal. A graph is **k-orientable** [Kar98] if given a graph G we can construct a graph G' by orienting all edges in G so that for every v in V' $deg_{in}(v) \leq k$. Therefore, a graph is k-orientable if the edge orientability problem can be solved so that for each vertex v in the result graph G' $deg_{in}(v) \leq k$ is. [CSW07]

A Cuckoo hash table is a hash table where the position of each element e is dictated by $h_0(e)$ or $h_1(e)$. We can construct such a hash table with **Cuckoo Hashing**: Given a set of elements and an empty table we start to pick one of the elements e from the set. If the position $h_0(e)$ or $h_1(e)$ is empty, then we write e to that position. If both positions already

have elements a displacement strategy is needed. The easiest is to randomly pick h_0 or h_1 and then reinsert the element at $h_0(e)$ or $h_1(e)$ into the set of elements and write e to the freed-up position. [PR01]

A bucketized Cuckoo hash table is a hash table which is divided into disjunctive buckets containing k elements each. Each element is stored in a bucket dictated by one of two hash functions. We consider the *offline* case where all elements are available at the beginning. In our setup the elements are in a table without any order. Given two independent hash functions we want to construct a bucketized Cuckoo hash table out of the given table.

It is equivalent to the k-orientability problem (with the same k). Each instance of the bucketized Cuckoo hash table construction implicitly defines a graph in the following way: each bucket creates a vertex. Each element e forms an undirected edge between the buckets $h_0(e)$ and $h_1(e)$. Note that $h_0(e)$ and $h_1(e)$ can be the same bucket. Directing an edge $\{b1, b2\}$ to $(b1, b2)$ means to assign an element which can either be hashed into $b1$ or into $b2$ to $b2$. If this graph is k-orientable, then we can direct all edges so that for every v in V' $deg_{in}(v) \leq k$ is. Therefore, each bucket has $\leq k$ assigned elements and because we directed all edges, we assigned all elements. So, we have a valid bucketized Cuckoo hash table.

2.2 The Selfless(k) algorithm

The Selfless(k) algorithm solves the k-orientability problem by either giving a solution or proving that there cannot be one [CSW07]. It does so by assigning a load-degree to each node. Then it directs edges to nodes with minimal load-degree. The basic idea is that directing an edge to a node with minimal load-degree is never wrong.

Given an undirected graph G the Selfless(k) algorithm first initializes a directed graph G' with $V' = V$ and $E' = \{\}$. Then it defines a load-degree for every vertex $v \in V$ as $deg_{ld}(v) := deg(v) + 2 \cdot deg_{in}(v)$. If there is a vertex v with $deg_{ld}(v) - deg_{in}(v) \leq k$ and $deg(v) > 0$, then it takes all edges $\{v, w\} \in E$ and moves them to E' as (w, v). If not it chooses a random vertex v out of all vertices with minimal load-degree out of G, chooses a random edge $\{v, w\} \in E$ and moves it to E' as (w, v). This is repeated until $E = \{\}$ or the minimal load-degree becomes $> 2k$. If there are edges left in E after it stops, then the k-orientability problem for the given graph and k cannot be solved. Otherwise, it yields a valid solution. [CSW07]

Algorithm 1 Selfless(k) algorithm

```
1: procedure SELFLESS(K) ALGORITHM
2:     buildLoadDegrees()
3:     while ∃e ∈ E && min. deg_ld ≤ 2k do
4:         if ∃v : deg_ld(v) − deg_in(v) ≤ k && deg(v) > 0 then
5:             while ∃{v, w} ∈ E do
6:                 move it to G′ with (w, v)
7:         else
8:             choose v with min. deg_ld
9:             choose edge {v, w} and move it to G′ with (w, v)
```

2.3 Incremented bucketized Cuckoo hash table construction

We compare our algorithms with an iterated insertion algorithm using two hash functions. The table is divided into disjunctive buckets with space for k elements. The main idea of the algorithm is to directly insert each element into one of its two buckets. If that is not possible, then a displacement strategy is needed to free-up a space in one of the two buckets.

The algorithm exists in two different versions, a growing and a non-growing. We mainly use the non-growing version for comparison. Our setup was given a table filled with random elements and two hash functions to construct a bucketized Cuckoo hash table. To do so, the non-growing version creates a new empty table of the same size in which the elements are inserted. The growing version also creates a new table, but with a fixed size.

The table periodically grows by a fixed size. The advantage is that already inserted elements can be freed in the original table. The disadvantage is that the table is always near capacity. Therefore, the insertion time can be very high for high load-factors. Note that in comparison to our algorithms the growing version does not work inplace, it creates the new table at once. Because the growing version can periodically deallocate inserted elements, it almost works inplace. Therefore, it is closer to our algorithms.

The displacement strategy for both versions is a bound breath-first search (abv.: BFS). Let e be an element we want to insert and $b_0 = h_0(e)$ and $b_1 = h_1(e)$ its two buckets. If both b_0 and b_1 are full, then the algorithm cannot simply insert e and starts the displacement strategy, for example a bound BFS.

The BFS iterates over all elements in b_0 and b_1 and checks whether the other bucket for each element has a free space. If that is the case, then the appropriate element is moved to its other bucket and we can insert e into the freed-up position. If not we iterate over all elements in the other buckets we just checked and so on. If a certain depth is reached the hashing fails and breaks off, apart from a negligible probability, for the non-growing version, caused by a too high load-factor.

3 Algorithms

This chapter presents our algorithms and the ways we adapt the Selfless(k)-algorithm. In Section 3.1 we describe variables and algorithm-specific definitions. Section 3.2 presents the base algorithm which all other variants either use or are derived from. Before we get to the variants themselves, Section 3.3 explains how our priorities behave when applying the base algorithm. This will later become important. After that Subsection 3.4.1 and Subsection 3.4.2 present variants of the base algorithm. The third variant in Subsection 3.4.3 presents a deviation of the base algorithm to work with much less memory. Additionally, in Section 3.5 we present a deviation of the base algorithm that enables the usage of multiple hash functions. Lastly, in Section 3.6 we present an algorithm that uses the previous variants to provide an easily parallelizable and cache-efficient algorithm using sublinear memory.

3.1 Definitions

Variables

We define $\boldsymbol{pri_{min}}$ as the minimal priority. The **minimal priority** is the lowest out of all priorities of buckets with a valid priority. We define $\boldsymbol{h}$ as the number of hash functions, this definition is only relevant for the multi hash functions algorithm.

To **fix an element** e means to write e into one if its two buckets $h_0(e)$ or $h_1(e)$. Each bucket has to know its fixed elements. Unless explicitly stated otherwise fixed elements are never moved again. To **prehash an element** e means to write e into one if its two buckets $h_0(e)$ or $h_1(e)$. In contrast to fixed elements prehashed elements can still be moved around until they are fixed.

An **active bucket** is a bucket that could be chosen next by the respective algorithm. Active buckets always have the same priority which is usually pri_{min}. We define the priority of the active buckets as $\boldsymbol{pri_{active}}$. **Activating** a bucket b means to turn b into an active bucket. **Deactivating** an active bucket means to turn an active bucket into a non-active bucket. Both activating and deactivating buckets do not affect their priorities. A **possible element** e for a bucket b is an element that could be hashed into the bucket b, meaning either $h_0(e) = b$ and/or $h_1(e) = b$. An **index** for an element e is the position of e in the table. Whenever e is moved the indexes for e have to be updated to its new table position.

3.2 Base algorithm

Algorithm 2 Base algorithm

```
1:  procedure Base algorithm
2:      buildPriorityQueueWithLoadDegrees()
3:      while ∃ b with priority 0 do
4:          safeBucket(b)
5:      while ∃ b with pri_min && pri_min ≤ k do
6:          if deg_in(b) == k then
7:              remove b from PQ
8:              continue
9:          if ∃non-hashed elment e with h_0(e) == b || h_1(e) == b  then
10:             fixElementIntoBucket(b, e)
11:             priority(b)++
12:             priority(getNeighbourBucket(b, e))−−
13:             if priority(getNeighbourBucket(b, e)) == 0 then
14:                 safeBucket(getNeighbourBucket(b, e))
15:         else
16:             remove b from PQ
17:             continue
18: procedure safeBucket(bucket b)
19:     while ∃non-hashed elment e with h_0(e) == b || h_1(e) == b  do
20:         fixElementIntoBucket(b, e)
21:         priority(getNeighbourBucket(b, e))−−
22:         if priority(getNeighbourBucket(b, e)) == 0 then
23:             safeBucket(getNeighbourBucket(b, e))
24:     remove b from PQ
```

The Selfless(k)-algorithm [CSW07] is a graph algorithm. We use a different notation of it as a hashing algorithm. We use an equivalent priority function subsuming all priorities $\leq k$ under priority 0. Therefore, the general idea is the same: if we have a bucket with less or equal possible elements than space, then we fix all those possible elements. Otherwise, we choose a bucket with space and a minimum of possible elements $+2\cdot$ fixed elements. Note that we only choose buckets with priority 0 or pri_{min}. Because priority 0 is the lowest valid priority we always choose buckets with pri_{min}. This results in active buckets always having the priority pri_{min}, meaning $pri_{active} = pri_{min}$.

Each instance of the bucketized Cuckoo hash table construction implicitly defines a graph in the following way: each bucket creates a vertex. Each element e forms an undirected edge between the buckets $h_0(e)$ and $h_1(e)$. Note that $h_0(e)$ and $h_1(e)$ can be the same bucket. Directing an edge $\{b1, b2\}$ to $(b1, b2)$ means to assign an element which can either be hashed into $b1$ or into $b2$ to $b2$. If this graph is k-orientable, then we can direct all edges so that for every v in V' $deg_{in}(v) \leq k$ is. Therefore, each bucket has $\leq k$ assigned elements

and because we directed all edges we assigned all elements. So, we have a valid bucketized Cuckoo hash table. We optimize the load-degree function of the Selfless(k) algorithm and define our load-degree function as

$$deg_{ld}(b) = \begin{cases} 0 & \text{if } deg_{Self\,Ld}(b) - k \leq 0 \\ deg_{Self\,Ld}(b) - k & \text{else} \end{cases}$$

where $deg_{Self\,Ld}$ is the load-degree function of the Selfless(k) algorithm (see Section 2.2). When choosing a vertex v with $deg_{ld}(v) = 0$ we direct all edges to it as this is equivalent to $deg_{Self\,Ld}(v) - deg_{in}(v) \leq k$.
$0 \geq deg_{ld}(v) = deg_{Self\,Ld}(v) - k$ and therefore $deg_{Self\,Ld}(v) \leq k$. Because $deg_{in}(v) \geq 0$ we can conclude: $deg_{Self\,Ld}(v) - deg_{in}(v) \leq k$.
We need a priority queue (abv. PQ) to support the priorization of the deg_{ld}. What PQ we use, how we fix elements, and how we represent G depends on the respective variants and/or implementations.

3.3 Priority effects

Most of the algorithms and implementations use some of the here presented effects in order to work. We explain how the pri_{active} behaves when applying the base algorithm. Note that the pri_{active} is **equal** to the pri_{min} in the case of the base algorithm. We use pri_{active} for clarification in this section because the following effects apply even when slightly deviating from the base algorithm. While the base algorithm is running $pri_{active} < k + 1$ because the base algorithm stops when a pri_{active} of $k + 1$ is reached.

We analyse how our pri_{active} behaves when fixing a single element into an active bucket. Let e be an element with $b_0 = h_0(e)$ as active bucket and $b_1 = h_1(e)$ as other bucket. When fixing e into b_0 we increment the priority of b_0 by one and decrement the priority of b_1 by one. If b_1 was an active bucket before fixing e , then it gets the priority $pri_{active} - 1$. This causes the base algorithm to deactivate all active buckets except b_1 and to decrement the pri_{active} by one. Therefore, the next time the base algorithm chooses an active bucket it only has a single choice, b_1. If b_0 was the only active bucket and b_1 had a priority higher than $pri_{active} + 1$ before fixing e, then there are no more buckets with priority pri_{active}. So, the base algorithm increments pri_{active} by one and activates all buckets with the new pri_{active}. Otherwise, the pri_{active} stays the same because either b_1 gets pri_{active} as priority or other buckets still have pri_{active} as priority.

Here we analyse the *expected priority* for each bucket. The *expected priority* for each bucket is $2 \cdot \frac{n}{m} \cdot k - k = 2 \cdot \textit{load-factor} \cdot k - k$. In words the *expected priority* is the total amount of possible hashes divided through the total amount of space. The minus k comes from our priority/load-degree function. Because each element produces two possible hashes the total amount of possible hashes is $2 \cdot n$. For high load-factors the *expected priority* is nearly k.

Lastly, we analyse the overall behaviour of pri_{active}. The base algorithm always chooses an active bucket and increments its priority. The other bucket is random, so most likely, its priority is higher than pri_{active}, especially at the start where the priorities are more diverse. Therefore, the number of active buckets decreases until there are no more buckets with pri_{active}. This causes the base algorithm to increment the pri_{active} and activate all buckets with the new pri_{active}. This behaviour continues until the pri_{active} gets close to the *expected priority*. Buckets with a high priority are more likely to get their priority decreased. Because they have more possible elements, it is more likely that one of their possible elements gets fixed into its other bucket. Therefore, all priorities converge to the *expected priority*.

3.4 Variants

3.4.1 Index

The index variant applies the base algorithm. Therefore, an active bucket always has pri_{min} as priority. The general idea of the index variant is to represent the entire graph G. We achieve that by letting each bucket store the indexes for all its possible elements, hence the name of the variant. Now we represent G because we represent V by knowing all buckets. And we represent E because each bucket can get its possible elements over its indexes and calculate the other buckets. We do not need information about our neighbours in G', therefore, we only have to store the deg_{in} for each bucket.

3.4.2 Hybrid

The hybrid variant applies the base algorithm. Therefore, an active bucket always has pri_{min} as priority. The main idea of this variant is to try and prehash (see Section 3.1) every element at the start. We do that because for each prehashed element e we only create one index. Let us assume that e was prehashed into $b = h_0(e)$. Then we only create an index for the bucket $h_1(e)$. Bucket b and all other buckets have to store which of their elements are prehashed because knowing that provides the same information as an index. If we were to create an index for b, then that index would point to a position in b because we prehashed e into b. Therefore, creating an index for b would be redundant.

So, each bucket has to store its indexes and which elements are prehashed. We do our prehashing after building the priorities from the load-degrees because our prehashing takes into account the number of prehashed elements the buckets already have as well as their priorities. How a possible element for an active bucket is chosen changes. We always prefer prehashed elements. Only if there are no prehashed elements the indexes are used. This is done to avoid updating indexes. Same as for the index variant each bucket has to store the deg_{in}.

3.4.3 Inplace

The main idea of this variant is to remove the index data structure by relying more on prehashing. Therefore, we never have an explicit representation of G. This variant deviates from the base algorithm, since the graph cannot be traversed in the same way. A major change is that an active bucket does **not** always have pri_{min}.

The problem that occurs when trying to strictly apply the base algorithm is that we run into the situation where all active buckets have no prehashed elements but still have possible elements. We could resolve this, by scanning the table and prehashing all elements every time that occurs. That would require a lot of table scans making this variant unfeasible for any implementation. What we are doing instead is a mixture of repeated prehashings, smart selection of active buckets and allowing non-minimal bucket selection. To yield equivalent results to those of the base algorithm we apply a correction at the end.

The inplace variant is segregated into five parts. The first part is the initialization which only happens once. The second part is prehashing all elements with additional functionality. The third part applies the base algorithm but it changes how we choose one of the active buckets. The fourth part does the same but introduces a new restriction. Until part four decides to call the last part we always call part two, then three and then four in a loop. The last part is only called once and tries to fix all leftover elements. Then it applies a correction, which fixes any elements that may still be leftover by moving around fixed elements. Figure 3.1 illustrates that control flow.

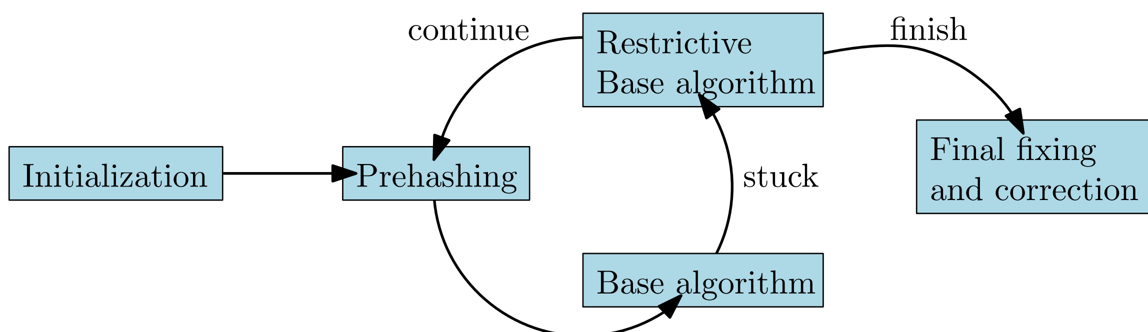


Figure 3.1: Control flow of the inplace variant

Initialization (1.) The initialization first calculates the *expected priority* and then tries to fix elements. It does so by calculating how many possible elements each bucket has. Then we repeatedly iterate over all elements and check whether an element could be hashed into a bucket with number possible elements $< k$. If it does we fix that element into that bucket and decrease the number of possible elements for its other bucket. This is repeated until the number of fixed elements per iteration becomes too small (specified in Subsection 4.2.3). If all elements are fixed the algorithm stops. How many elements

are fixed in this part depends on the load-factor. Therefore, the number of elements left for the rest of the algorithm does too, and thus the running time of the algorithm as well. After that the PQ is initialized.

Prehashing of elements (2.) Despite of its name this part not only prehashes elements but also fixes some, although a minority. In this part we iterate over all elements and, similar to the hybrid variant, try to prehash them taking into account the number of prehashed elements and the priorities of their respective buckets. There are two exceptions where we fix an element instead of prehashing it. It can occur that a bucket with a priority smaller than pri_{active} still has space and possible elements. If we come across a possible element for such a bucket we fix it directly into that bucket. This situation is not possible in the hybrid variant.

It is important that we fix this element and do not prehash it. If we were to prehash it and its other bucket has pri_{active} as priority and only this element as possible element we could get stuck. The other situation where we fix an element is if one of its buckets is full. In that case we fix the element into the other bucket.

Application of the base algorithm (3.) When choosing an active bucket we check whether it has prehased elements or not. We only choose active buckets with prehashed elements. If an active bucket does not have prehashed elements we skip it. If it also has no more possible elements, then we remove it from the PQ. Although we have no indexes we are able to check if a bucket has no more possible elements by calculating backwards from its priority and deg_{in}.

If all active buckets have no prehashed elements, then we continue with the fourth part. If we have very few (specified in Subsection 4.2.3) active buckets, then we increase the pri_{active} by one, activate all buckets with the new pri_{active}, and call part four. Therefore, active buckets may have a higher priority than pri_{min}. This is a deviation of the base algorithm, but to continue in this part would require an expensive table scan (prehashing), which we are able to avoid or postpone making it easier, faster, and better.

Restrictive choosing of buckets (4.) The idea behind this part is to follow the base algorithm but to choose from the active buckets in a way that guarantees that we do not decrease the pri_{active}. Note that pri_{active} may not be pri_{min}, see last part. To achieve that we only fix prehashed elements that could be hashed into an other bucket with a **higher** priority than pri_{active}. Every active bucket with no such prehashed elements is skipped. Therefore, we can never create a bucket with $pri_{active} - 1$ and no prehashed elements which caused the stop in part three. If pri_{active} is one smaller than the *expected priority*, then we repeat that for a sufficient (specified in Subsection 4.2.3) degree of precision or time and then call the last part, the final fixing and correction. Otherwise, we repeat it to a certain precision (specified in Subsection 4.2.3) and then do the second and then the third part again.

Progress in part three and four It is important that part three and/or part four fix a substantial amount of elements before part two is called again. If they would not do that, then the time spent on the table scan (prehashing) in part two would make this algorithm unfeasible for any implementation. To ensure the progress in part three and/or four we take a look at progress-hindering situations for each part and analyse how the other part behaves in the situation.

The only progress-hindering situation that can occur for part four is if the possible elements for the active buckets do not or only have very few other buckets with a higher priority. We can now conclude that those possible elements have to have buckets with pri_{active} as their other bucket. After this part part two does a prehashing and prehashes those possible elements to one of their active buckets. If an active bucket has a possible element with a bucket with a higher priority than pri_{active} as its other bucket, then part two will prehash it to the active bucket.

This creates active buckets of the following nature: they either have a prehashed element or they have no prehashed elements and their only possible element is prehashed into an other active bucket. Note that buckets with multiple possible elements and no prehased elements are possible but very unlikely because the prehashing algorithm would have to prehash all those multiple possible elements to their other buckets who have the same priority.

This means when we fix an element into an active bucket we most likely create a bucket with $pri_{active} - 1$. But the important thing is that we do not run into the stopping situation of part three because that bucket with $pri_{active} - 1$ will either have a prehased element or its only possible element was fixed into the other bucket leaving it with no possible elements and we can remove it from the PQ.

In every other situation part four guarantees our progress. Interference from buckets with a lower priority than pri_{active} is minimal because there can only be very few of those buckets. Additionally, part two fixes elements into those buckets whenever it is called guaranteeing progress for them and raising their priority.

Final hashing and correction (5.) This part gets called from the fourth part with non-fixed elements leftover. If the fourth part ran with a very high degree of precision, then we have the following situation: Apart from a negligible number of buckets every bucket has either pri_{active} or $pri_{active} + 1$. Note that as before pri_{active} might not be pri_{min}. Every leftover element, apart from a negligible number, can either be hashed into two buckets with pri_{active} or two buckets with $pri_{active} + 1$. Otherwise, the fourth part could have fixed it and would not have changed to part five.

We now call our prehashing algorithm from part two except that we fix the elements instead of prehashing them. Because we deviated from the base algorithm and our prehashing algorithm, although precise on two separate priorities, is not perfect either, we

may have elements that we cannot fix. To fix those we apply a correction that scans the table and finds these non-fixed elements. We then remove them from the table and store them. Afterwards we insert them using a bound BFS. If the BFS fails, then our hashing failed, most likely because of a too high load-factor.

3.5 Multi Hash Functions

The base algorithm is not designed to be used with more than two hash functions. We offer a variation that can be used with multiple hash functions. A trivial solution would be to apply the base algorithm (described in *base1*), but that yields a result with elements leftover that could not be fixed. We can fix those using a BFS with multiple hash functions.

Our variation decreases the number of leftover elements significantly, see our experimental results in Section 5.7. To clarify the development of our algorithm we labelled its different versions. When running the base algorithm we see an interesting effect with the priorities. As explained in Section 3.3 the priorities grow towards the *expected priority* the further the algorithm progresses and if the pri_{active} decreases it only does so by one with only one bucket having the new pri_{active}.

base1: h is the number of hash functions we use. Therefore, the expected number of possible elements for each bucket is $h \cdot k$ resulting in an *expected priority* of $h \cdot k \cdot \textit{load-factor} - k$. We can run the base algorithm if we do everything we did for the single other bucket of the fixed element for all other buckets of the fixed element. When doing so, our pri_{active} first rises to the *expected priority* but then starts getting smaller and smaller again.

base2: If we take a close look at the base algorithm we can see why. When fixing a possible element into an active bucket we increase that buckets priority by one. Because the element could have been hashed into $h - 1$ other buckets, we decrease the priority of $h - 1$ buckets by one, assuming they are still in the PQ. So, fixing an element decreases the sum of all priorities by $h - 2$. For $h \geq 3$ that results in lowering the *expected priority* and thereby causing the decrease of the pri_{active} at the end. To counteract that effect, we increase the priority of an active bucket that we fix into by $h - 1$ instead of 1. This keeps the sum of all priorities constant and yields a better result.

base3: Our second modification changes how the possible element that is fixed into an active bucket is chosen. The idea is to prefer possible elements that can be hashed into fewer other buckets because some of their other buckets are full. To do so, we prioritize the possible elements of each bucket by giving each element the priority of how many of its other buckets are still in the PQ. At the start that is always $h - 1$. As later on buckets that are full and have possible elements get removed from the PQ this changes. There are h priorities for the possible elements. When choosing a possible element for an active bucket now we always choose that with the lowest priority.

final: Building on the last modification we let the priorities of the elements also affect

the priorities of the buckets. Decrementing the priority of a bucket when decrementing the priority of a possible element has worked badly in practice. Therefore, we consider only one of the priorities for a possible element as extra special, the priority 0. When a possible element for a bucket has priority 0 it means that it can only be fixed into that bucket because all its other buckets are full. What we do now is when we decrement a priority for an element for a bucket to 0 we also decrement the priority of that bucket by $h - 1$. That together with all the other modifications of the base algorithm leads to a good heuristic with very few leftover elements before the correction. The leftover elements are fixed using a BFS with multiple hash functions.

3.6 Two level

The two level variant offers a modification which allows it to be used with any of the above variants. The idea behind it is to run the variants from above on smaller subtables using only sublinear memory. Ideally, we also try to profit from cache-effects. Additionally, this makes our algorithm easily parallelizable, since, the subtables can be constructed independently.

It starts by dividing the given table into $\sqrt{m}$ subtables. Then it uses an additional hash function to assign each element to a subtable. After this is done, it assigns the free space on a percentage basis for each subtable as the number of elements of the subtable divided through the number of all elements. On each of those subtables we run one of the variants from above.

Because we calculate our subtables sequentially, we only consume the memory of the respective variant on that subtable. Our memory for each of the variants from above is linear in its table size. Our subtable size is $\sqrt{m}$, therefore we have a memory consumptions which is linear in $\sqrt{m}$. So, our memory consumption is sublinear.

We can find any element by first finding the subtable of the element using our additional hash function with which we assigned the element to a subtable. We can then determine the two buckets of the element in its subtable using h_0 and h_1. To support that, we also need to store the beginning and ending of each subtable. This also only uses sublinear memory because we only have a sublinear number of subtables ($\sqrt{m}$).

4 Implementation

This chapter presents the different implementations of the algorithms presented in chapter 3. Section 4.1 focuses on structures all implementations have in common. Subsection 4.2.1 presents the implementation of the index variant. It covers a lot of data structures and concepts that the other implementations reuse. Subsection 4.2.2 provides the implementation details of the hybrid variant. Subsection 4.2.3 presents the implementation for each part of the inplace variant. In Section 4.3 we present our implementation of the multi hash functions variant and explain why we cannot use previously used structures. Lastly, we present the implementation of our two level variant in Section 4.4 using only sublinear memory. In Section 4.5 follow some notes on its parallelization using OpenMP.

4.1 Common structures

To store the deg_{in} we use an array of size m/k, which stores for each bucket how many fixed elements it has. We store the fixed elements in the beginning of each bucket b. Therefore, the fixed elements of b are in the interval $[b, b + deg_{in}(b))$ (see Figure 4.2). Thus, they do not have to be explicitly marked.

Since k is usually 4 or 8 and at most 16 some values are bound. We can exploit this. Priorities and the deg_{in} are both bound by k. Therefore, we can use the smallest possible uint type for them, which would be unit8_t in C++. Other values like indexes and buckets are bound by m or by m/k. To save memory as well as increasing the cache-performance, we want to choose the smallest uint or int that can hold every value. We achieve that by letting each variant have template parameters for uint types and/or int types with set default-types. The best types are automatically initialized for each variant.

4.2 Variants

4.2.1 Index

The index variant uses indexes for each bucket, therefore we need to store them. The easiest way to do so is to use a two-dimensional vector and to let each bucket store its indexes in a vector. Each index access would then require two array accesses. We always choose an arbitrary active bucket, therefore the access-pattern of indexes is random. This means the cache access will produce the consecutive cache misses that cannot be pipelined.

Our solution is to use a one-dimensional array where each bucket stores a fixed number of indexes. The main advantage is that each index access only needs one instead of two

array accesses and therefore approximately half the time. Let $c \cdot k$ be the number of indexes we store for each bucket. The idea is to store indexes for a bucket b in the interval $[b \cdot c \cdot k, (b + 1) \cdot c \cdot k)$. So, all indexes for b are at $b + x$ with $x < c \cdot k$ in our index array. We define that x as a **relative index** for b. Two problems can now occur: Bucket b may not have $c \cdot k$ indexes or it has more than $c \cdot k$ indexes.

We solve both problems through structuring and rebuilding of our index array and using an additional array. The idea is to store which indexes in our index array are valid. For that we use an array of size m/k which stores for each bucket the last relative index or -1 if it has no indexes. We know that the relative index is bound by $c \cdot k - 1$. As mentioned before k can be taken as ≤ 16. Therefore, we can use int8_t as data type for the last relative index array.

Now let x be the relative index for a bucket b. We structure our index array similar to our fixed elements in our table by keeping all valid indexes in the beginning of our index array. This means all valid indexes are in the interval $[b \cdot c \cdot k, b \cdot c \cdot k + x]$ in our index array. Everything in $(b \cdot c \cdot k + x, (b + 1) \cdot c \cdot k)$ is invalid. If we need an index for a bucket we always check our last valid index array. If it is ≥ 0 we use the index on that relative position and decrease our last valid index by one. Therefore, indexes are always worked off from back to front.

Deleting an index is equivalent to invalidating it. If we delete an index at a relative index $< x$, then we have to maintain our structure. We do so by swapping our index that we want to delete with the index at the last valid relative index and then decrement the last valid relative index by one. We can see the behaviour of our index arrays when fixing an element at our fixing examples in Figure 4.1, Figure 4.3, and Figure 4.4.

Now let us look at a bucket with more than $c \cdot k$ indexes. Let b be a bucket with i indexes where $i > c \cdot k$ and $i > 2 \cdot k$. Therefore, its initial priority is $i - k > k$ and we are unable to store all indexes for b in our index array. Additionally, we delete indexes for b before it becomes active because our algorithm only chooses active buckets with a $pri_{active} \leq k$. If we only delete indexes that were stored in our index array this leads to the following situation: b still has space but no valid indexes although it still has indexes that were not stored. We could do a rebuild of both index arrays, but how can we differentiate b from a bucket that has no more indexes? Each rebuild for such a bucket needs a whole table scan which would cost us a lot of time.

Our solution is to set before every build of the index arrays the last entry, that would be relative index $c \cdot k - 1$, for each bucket to the maximum int of the used int-type. When building the indexes we fill up the indexes from bottom to top for each bucket. We build the indexes by iterating over all elements and creating indexes for both buckets for all non-fixed elements. That means that the last entry for a bucket is only overwritten if the bucket has $\geq c \cdot k$ indexes. When reaching a bucket with a last valid index of -1 later we check for that last entry. If it is set to maximum int, we know that the bucket has no more indexes and that we do not have to do a rebuild. If it is not we know it was overwritten

by an index and we do a rebuild. This may still lead to an unnecessarily early rebuild but keeps the number of rebuilds close to the minimum.

We chose c as 2.5 because it keeps the number of rebuilds very low. Changing the factor c would either lead to more rebuilds (if c is decreased) or to less rebuilds (if c is increased). It is important to mention that this introduces a slight dependency on the load-factor. The lower the load-factor the more likely all buckets can store all their indexes. Therefore, we have to do less rebuilds of our index arrays which makes it faster.

The second thing we discuss is what kind of PQ we use and how we fix elements. From Section 3.3 we know that the pri_{active} is bound and is either staying the same, increasing by one or decreasing by one with only one bucket having the new decreased pri_{active}. We exploit that effect by using an implicit PQ. Our PQ only keeps the active buckets explicitly using a vector. In an extra array we store the priority for all buckets. Buckets that are full or have no more possible elements are marked by having the priority -1. When choosing one of our active buckets we always pick the last entry of our vector.

Apart from adjusting the priority array a priority increase is a simple pop_back from a vector because the only increases happen when fixing into an active bucket and we always pick the last entry of the vector when doing so. A priority decrease is only an insert in our vector if the new priority of the bucket is the pri_{active}.

When the pri_{active} increases, we iterate over all buckets in our priority array to build the vector with the active buckets new because all buckets with the new pri_{active} are activated. This is actually faster than a normal bucketPQ as the number of rebuilds is bound by the maximal pri_{active} which is bound by k.

A priority decrease on an active bucket b has to be handled now. If we were to follow the base algorithm we would decrease the priority of b resulting in a decrease of the pri_{active} with b being the only active bucket. This would require a rebuild of our active buckets vector. Also b would get chosen next by the algorithm, get an element fixed and its priority increased by one. If no other bucket got the new pri_{active}, then the pri_{active} would increase and we would have to rebuild our active buckets vector again.

To avoid these rebuilds of our active buckets vector we directly fix an element into b and do not adjust its priority. This is equivalent to doing a priority decrease and following the base algorithm because the priority and the number of fixed elements of b are the same afterwards. We also fixed an element just like the base algorithm. If another active bucket gets a priority decrease we do the same for it. Otherwise, we have the same pri_{active} and active buckets as the base algorithm.

Fixing an element into a bucket b, an example can be seen in Figure 4.1, consists of the following steps: Through our last valid index array we get a relative index for our index array. We use that to get the actual index of an element from our index array. We swap the content at $b + deg_{in}$ with the element at our index. If the swapped out content

was an element we update its indexes. Then we delete the index of our fixed element from its other bucket. We increment the deg_{in} for b by one. We also adjust our index arrays for b by decreasing the last valid index by one which marks the index we used as invalid. We can do this because we always choose the last valid index in our index array. And we have to do the PQ operations, a priority increase for b and a priority decrease for the other bucket.

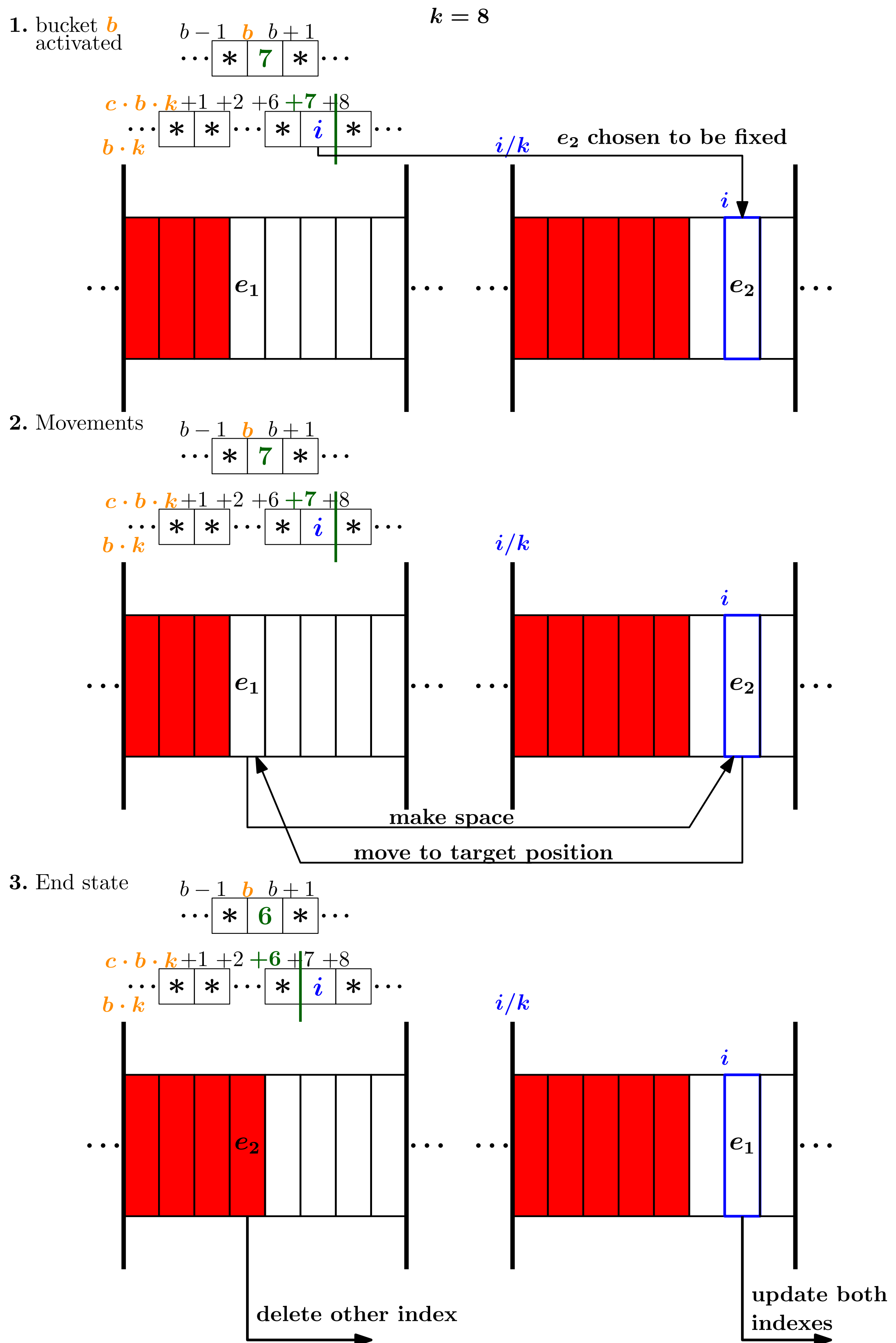


Figure 4.1: Fixing by index

4.2.2 Hybrid

The main idea of the hybrid variant was to try and prehash every element at the start and to create only one index per prehahsed element. To support indexes and a PQ we choose the same implementation as the index implementation (see Subsection 4.2.1). We have one additional array of size m/k which stores for each bucket how many prehashed elements it has. The idea is that for a bucket b in the interval $[b,b + deg_{in})$ are fixed elements, in $[b + deg_{in}, b + deg_{in}$ + prehashedElements) are prehashed elements and at the rest of the bucket are either elements or free space. An example can be seen in Figure 4.2.

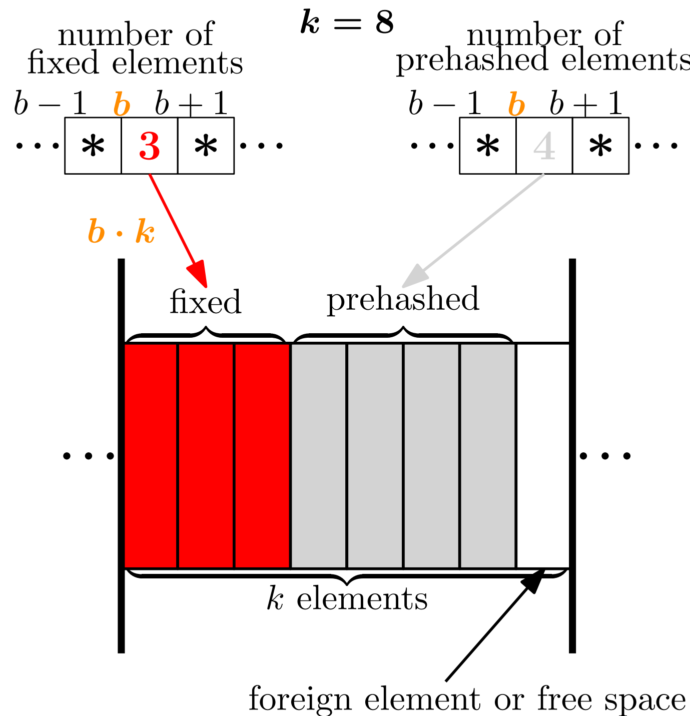


Figure 4.2: Fixed and prehashed arrays

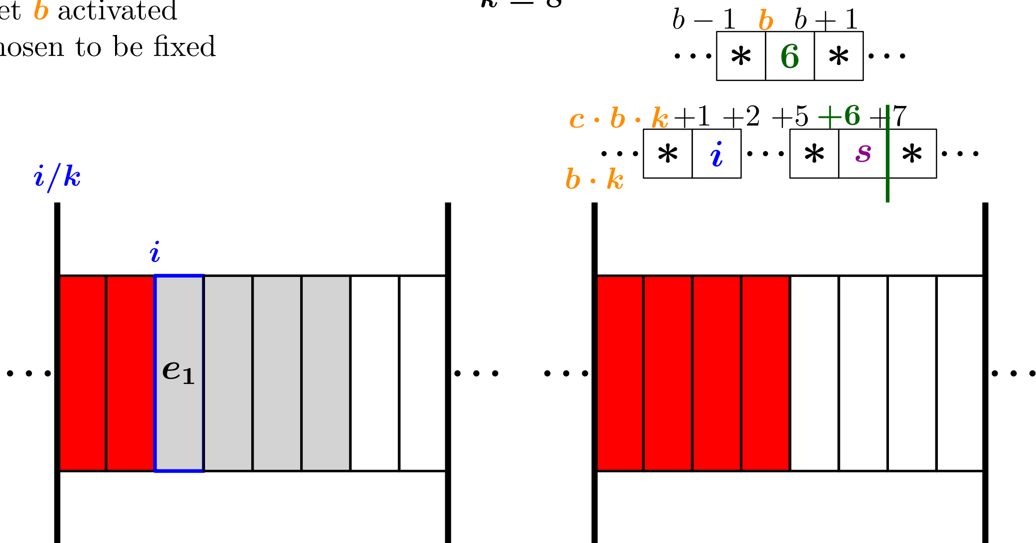
Our initial and only prehashing happens after we built our PQ and before we start choosing active buckets. It is separated into two parts. In the first part we iterate over all elements and prehash each element to the bucket with the smaller *priority* + *number of prehashed elements*. If both values are equal we always prefer the bucket that was calculated by h_0. If we prehash an element, then only its other bucket gets an index. Elements that cannot be prehashed are skipped.

In our second part we iterate over all elements again and create indexes for all elements that are not prehashed. Combining the two steps has proven to be slower as non-prehashed elements are likely to be moved around in the first part. So, if we were to create indexes for them, then we would always have to update indexes when moving them around which is expensive.

During the algorithm the fixing of an element can happen in three different ways. The first and easiest way which the hybrid algorithm tries to do as much as possible is to fix a prehashed element into a bucket b. Figure 4.3 shows an example of that. Such a fixing consists only of incrementing the deg_{in} of b, decrementing the number of prehashed elements for b by one and deleting the index of that element out of its other bucket. And we have to do the PQ operations, a priority increase for b and a priority decrease for the other bucket.

The second and most costly way to fix an element is fixing an element by index which is prehashed in its other bucket. An example is in Figure 4.4. Let b be the bucket we want to fix into. As we always prefer fixing prehashed elements we can be sure that b has no prehashed elements. Through our last valid index array we get a relative index for our index array. We use that to get the actual index of an element from our index array. In our case that element is prehashed in its other bucket, otherwise it would have two indexes and we would proceed with fixing by index.

We have to consider three elements here. In order to maintain a valid prehashed interval we have to swap the element with the last prehashed element in its other bucket and then swap it into b. It is implemented by moving every element to the right position. After that we decrement the number of prehashed elements for the other bucket and increment the deg_{in} for b. We also adjust our index arrays for b by decreasing the last valid index by one which marks the index we used as invalid. We can do this because we always choose the last valid index in our index array. We also have to update the index for the last prehashed element in its other bucket because we moved it. If the content that we swapped out of b was an element we also have to update its indexes. And we have to do the PQ operations, of course. The third way an element can be fixed is if it has two indexes just like in the index implementation. The fixing is therefore analogue and can be looked up at Figure 4.1 and Subsection 4.2.1.



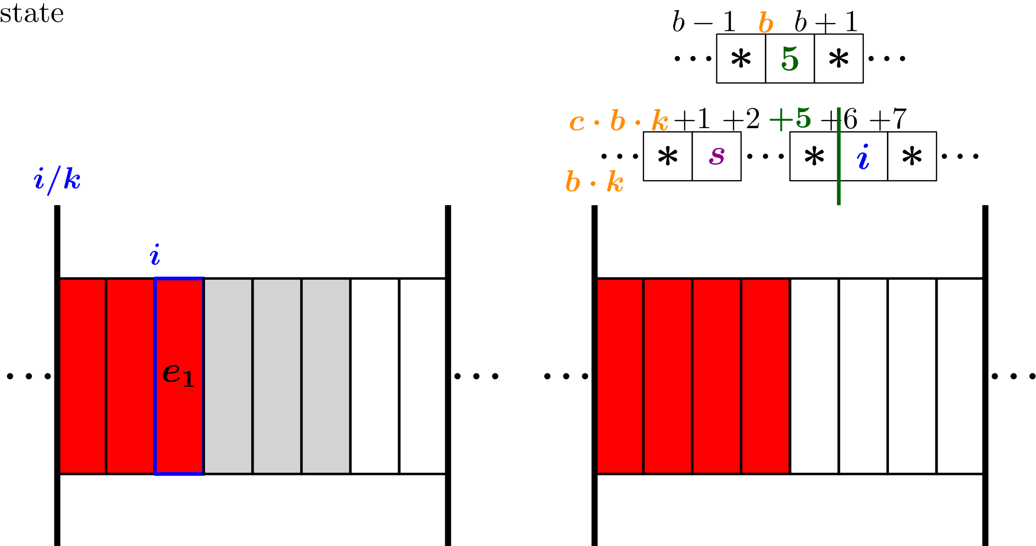


Figure 4.3: Fixing the first prehashed element

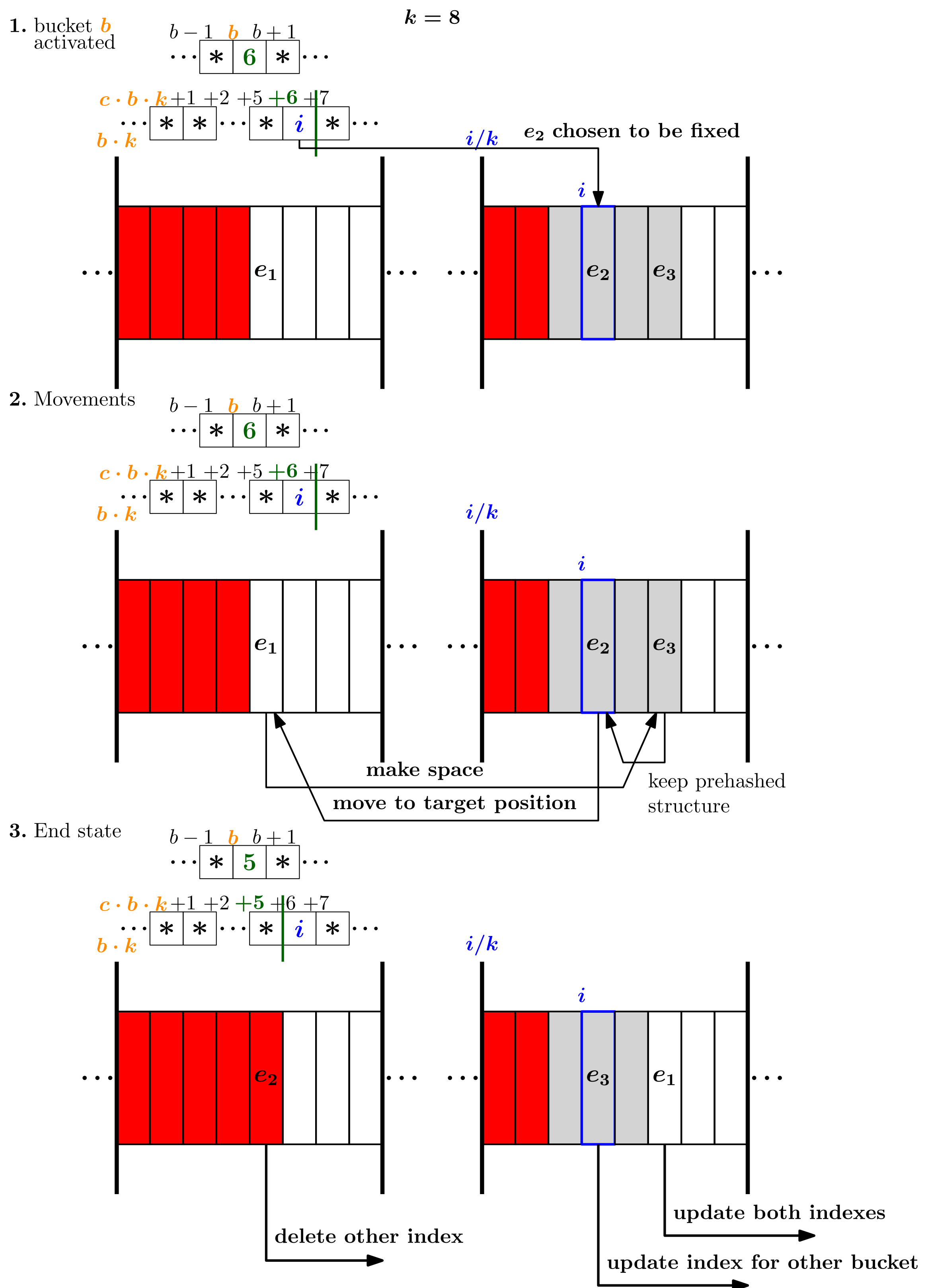


Figure 4.4: Fixing a prehashed element by index

4.2.3 Inplace

Similar to the hybrid implementation, we have an array to store the number of prehashed elements for each bucket. The intervals for fixed and prehashed element for a bucket are identical to those of the hybrid implementation (see Subsection 4.2.2 and Figure 4.2). We also use an implicit PQ like the hybrid and the index implementation do (see Subsection 4.2.1). We do that for the same reasons. With the common structures this covers all our data structures. In Subsection 3.4.3 we segregated the inplace variant into five parts. We now present the implementation details for each part.

Initialization (1.) In Subsection 3.4.3 we said that we calculate the *expected priority* and in Section 3.3 we calculated the *expected priority* as $2 \cdot \textit{load-factor} \cdot k - k$. The problem is that our algorithm goes in its final phase when our pri_{active} reaches *expected priority* $- 1$. Therefore, we need to round the *expected priority*. After much experimenting we decided for the rounding strategy to round down if the *expected priority* $< **,1$ and to round up otherwise. This is done to avoid going into the final phase too early or too late.

For the fixing in the first part we do not create an extra array. We just use our array that stores the priorities for each bucket to store the number of possible elements. We can do that here because we do not need priorities here. We stop our fixing and continue when we fix $\leq 4\%$ of all elements in an iteration. When we fix an element we decrease the number of possible elements only for its other buckets, not for our own. When we are done, the priority array contains the number of possible elements plus the number of fixed elements which would be the $deg(b) + deg_{in}(b)$ for each bucket b. To initialize the PQ we just iterate over all buckets and add for each entry b in our priorities $deg_{in}(b) - k$ which results in our our priority function $deg(b) + 2 \cdot deg_{in}(b) - k$. Now the initialization is done, note that we do not have anything stored explicitly for our PQ at this point.

Prehashing of elements (2.) This part always requires a rebuild of the active buckets which we are keeping explicit and therefore does that at the end. The reason is that we also fix elements directly and have to adjust the PQ. To do that implicitly over the priority array is no problem. But as we might fix into an active bucket we would have to adjust the active buckets vector. The access is in our case random. Therefore, the access-time is linear in vector length which is too slow for us. Additionally, the fourth part is usually run before this one and changes, for the same reason, the priority array without adjusting the active buckets vector as well. So, at the end of this part we rebuild the vector of active buckets by iterating over our priority array and adding each bucket with pri_{active} to the vector.

We always try to fix an element before trying to prehash it like the hybrid variant does it. Therefore, we check for each element if it has a bucket with a priority $< pri_{active}$ or if one of its buckets is full. If that is the case we directly fix it into the appropriate bucket. Otherwise, we do a prehashing like the hybrid implementation does it (see Subsection 4.2.2) except that we do not create indexes.

Application of the base algorithm (3.) This part is much like the application of the base algorithm in the index and hybrid implementation. Because we also have an implicit PQ, we also have to deal with a priority decrease of an active bucket. We do that almost in the same way as the other implementations do it. The only difference is that if a bucket has no prehashed elements, then we actually decrease its priority by one without adjusting the pri_{active} and active buckets and call part four. We leave the bucket in the active buckets vector as part four does not use the vector and part four either finishes or calls part two which rebuilds the vector.

The other way to call part four is if we run into the situation where all our active buckets have no prehashed elements. We either call part four directly or increment the pri_{active} by one if we have less than 10 active buckets. We chose this number to only make a small deviation of the base algorithm while gaining a lot of performance by not calling part two.

In opposition to the index and hybrid implementation, we go through our active buckets vector from the start to the end. We do that because we cannot always choose the last bucket as it may not have prehashed elements. If we perform a priority increase or a delete when a bucket becomes full we simply overwrite the current position in the vector with the last element of the vector and call a pop_back on the vector. A priority decrease only adds to the end of the vector if the new priority is the pri_{active}. In both cases we also have to adjust the priority array. Another advantage to this approach is that new active buckets are looked at last. Because they had a higher priority before it is more likely that they have no prehashed elements.

Restrictive choosing of buckets (4.) This part first decides whether it will finish the algorithm or call part two and then three again. It does so, as stated in Subsection 3.4.3, by checking whether the pri_{active} is one smaller than our calculated *expected priority*. As mentioned before, we do not use the active buckets vector. Instead, we iterate over all elements and if a non-fixed element has a bucket with pri_{active} **and** a bucket with a higher priority, then we fix that element into the bucket with pri_{active} and adjust the priority array. Note that we ignore our prehashing here and destroy the prehashing structure which is why, unless we call part five, part two **has** to be called after this part in order for part three to work.

With a counter we keep track of how many elements we fix in an iteration. In the finishing mode, we iterate 52 times over all elements or until 0 elements have been fixed in an iteration because then we would have the exact same situation for the next iteration and would fix no elements again. The 52 is a finishing-guarantee and was chosen to ensure performance as well as keeping the number of elements fixed in the last iteration low. After that we call part five, the final fixing and correction.

In the non-finishing mode, we do at least three iterations. We iterate once and then check through the counter if we were able to fix an element into a bucket for more than 1% of all buckets. If that is the case, then we do another iteration with the same check at the end. As soon as that fails for the first time, except for the beginning, we do one more

iteration and then call part two and three. If it fails at the beginning, we do two more iterations and then call part two and then three.

Final fixing and correction (5.) This part does a normal prehashing at first except that it fixes the elements instead of prehashing them. After that it performs a correction, which iterates over all elements and checks whether they are in the right bucket. If they are not, then we remove them from the table and store them in a vector. After that we iterate over our vector and insert the elements into the table following the same strategy as the iterated insertion algorithm:

Let e be an element we want to insert and $b_0 = h_0(e)$ and $b_1 = h_1(e)$ its two buckets. If both b_0 and b_1 are full, then we cannot simply insert e and start a bound BFS. The BFS iterates over all elements in b_0 and b_1 and checks whether the other bucket for each element has a free space. If that is the case, then the appropriate element is moved to its other bucket and we can insert e into the freed-up position. If not we iterate over all elements in the other buckets we just checked and so on. As we would run endlessly if a hashing is impossible, we have $log_{10}(n)$ as a bound for our depth.

We implement our BFS using a vector where each bucket is inserted in the order we looked at it. Because each bucket has the same number of elements and therefore the same number of other buckets that we check, we are able to determine our parent bucket by using our current index in the vector and our start and end position of our current iteration. That enables us to calculate all buckets of the chain backwards as soon as we find a bucket with space.

4.3 Multi Hash Functions

In Section 3.5 we intruduced a priorization of elements for active buckets. We now need to support that priorization. Therefore, modifying the inplace variant is not an option as we do not have indexes and each bucket may only have a part of its possible elements as prehashed elements. Modifying the hybrid variant could work but we would have massive problems and overhead with the prehashed elements again as a changing priorizations there is difficult. Imagine a prehased element with a lower priority than an indexed element destroying the whole concept of the hybrid variant that prehashed elements always get preferred.

That leaves us with modifying the index variant or implementing something entirely new as the only viable solutions. We chose to modify the index variant because it already provides a concept for index structures. As in Section 3.5 h will represent the number of hash functions we use.

To represent our new index structure we use a two-dimensional vector. We choose that over a three-dimensional vector with the bucket as first dimension, the index priority as second and the indexes as the third because our solution will be faster as we will

only require two instead of three array-accesses each time we need an index. Our first dimension is $h \cdot$ *amount of buckets* big and represents both buckets and index priorities. The idea is that each bucket b gets the indexes $[b \cdot h, (b+1) \cdot h)$ assigned and that the vector at $b + x$ represents indexes with priority x for the bucket b.

Before we continue we should talk about what kind of PQ we are using because the other contents in this section depend on that. In the implementations above, we have seen only one type, an implicit PQ. We have to use an explicit PQ because one essential condition of the implicit PQ from above is missing. What the implicit PQ from above exploited was that if the pri_{active} decreases, then only one bucket can have that new pri_{active} and be active.

That is not the case here because we decrease the priority for up to $h - 1$ buckets when fixing an element. So, $h - 1$ buckets can have that new pri_{active}. Additionally, if the element that we fix into a bucket with the new pri_{active} could also be fixed into another one of those buckets with the new pri_{active}, then our pri_{active} would further decrease by one. So, to simply save those buckets with the new pri_{active} in a vector, which could grow $h - 2$ buckets per fixed element at worst, instead of the variable from the implicit PQ from above is not an option. That leaves us with an explicit PQ.

For the same argument that k is our maximal pri_{active} in the other implementations $(h-1) \cdot k$ is our max. pri_{active} here. To represent our PQ explicitly we use a two-dimensional vector. The first dimension represents the priorities and is bound by $(h - 1) \cdot k$ with the exception that PQ entries in the last bucket of the PQ represent priority $(h - 1) \cdot k$ **or** higher. To support a constant random access time we use the priority array that we know and another array which stores for each bucket the index of the bucket in the second dimension of our PQ vector. When moving in the PQ these arrays have to be adjusted as well.

We iterate over the active buckets in the same way as the index and the hybrid implementation. The only new thing we have to consider is what happens if a bucket becomes full and still has indexes left? Then we have to update the priority of the indexes for their other buckets. We do that by decreasing them by one which is done by moving them in the first dimension of our two-dimensional index vector. When an index gets priority 0 its bucket has to be moved down by $h - 1$ in the PQ as well.

4.4 Two level

In Section 3.6 we divided our table into $\sqrt{m}$ subtables. In practice it is important to keep all subtable sizes above $70 \cdot 10^3$ because for anything below the probability that a valid hashing is impossible no longer negligible is. Therefore, we divide the table in our implementation into the maximum potence of two which is smaller than the minimum of $\sqrt{m}$ and $m/(70 \cdot 10^3)$ guaranteeing a goal subtable size $\geq 70 \cdot 10^3$. We do that efficiently by using the most significant bit and bit shifts.

After that the number of elements per subtable is calculated using h_0. Because our number

of subtables is 2^z with $z \in \mathbb{N}_0$ we can do that very efficiently by assigning each element to the subtable $h_0(e)$ *AND* z. Then the actual size of the subtables is calculated by giving each subtable the percentage of their *number of elements*$/n$ free space. Therefore, the load-factor for each subtable is guaranteed to be the same, apart from minimal variations caused by rounding. The only thing that we have to store permanently throughout using the hash table are the beginnings of each subtable in the table which we do through an array. Now we actually write each element to its subtable.

Every table entry has the same probability for $h_0(e)$ for a random element e. Therefore, all of our subtables have about the same size. Because our goal subtable size is $\geq 70 \cdot 10^3$ the variety is small. Lastly, we call one of the variants from above on each subtable.

4.5 Parallelization

We parallelize the two level implementation with OpenMP. Thereby, we exploit that the subtables can be calculated independently, which makes it very easy for us to parallelize.

We parallelize the first part, the calculation of the subtables, by letting each thread calculate the subtables for his elements and then adding them recursevily. This only gives little, but some, gain as this calculation is already very fast in its sequential version. We decided to keep the moving of the elements sequentially. An efficient parallelization for this part is rather complicated and there are already well developed algorithms such as the inplace parallel partitioner used in [Axt+17] which can be used here.

Our main parallelization happens now. Each subtable has about the same size and the same load-factor. Therefore, the same variant has about the same speed on any subtable. Because of that and because all of our subtables are independent of each other we can easily parallelize here by letting the subtables be constructed in parallel where each thread, if possible, constructs the same number of subtables.

5 Evaluation

In this chapter we see how our implementations behave regarding running time, memory consumption and changing parameters such as load-factors and number of elements. Section 5.1 presents the legend that is used throughout this chapter. In Section 5.2 we describe common parameters for all tests. Section 5.3 presents the running times of our implementations. Section 5.4 then analyses the running time behaviour with varying load-factors. In Section 5.5 we examine the memory consumption of our variants. We analyse the efficiency of our parallelization in Section 5.6. Lastly, we look at the average error rates of our multi hash functions algorithm in its different versions in Section 5.7.

5.1 Legend

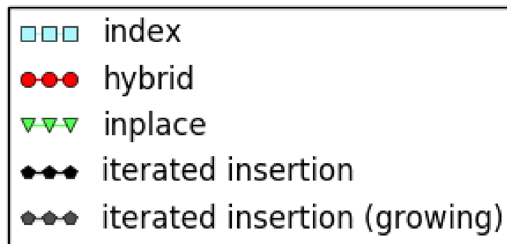


Figure 5.1: Legend

The legend in Figure 5.1 applies to all diagrams throughout this chapter and is not explicitly used below. Note that for diagrams showing the level variants or parallel variants "index" describes the variant that is used by the two level or parallel implementation.

The index variant with the cyan colour and the square markers uses index vectors for a full graph representation. The index variant with the red colour and the circle markers combines prehashing, which is predicting where elements should be fixed into and then writing them in advance, with indexes for a partly implicit graph representation. Both variants apply the base algorithm. The inplace variant with the green colour and the triangle markers uses prehashing to keep the graph fully implicit and does not have indexes. To do so, it slightly deviates from the base algorithm.

The iterated insertion algorithm with the black colour in its non-growing version, with the dark grey colour in its growing version, and the pentagon markers inserts all elements one by one and resolves collisions using a BFS. The non-growing version creates a new table

at once, whereas the growing version creates a table which periodically grows. Therefore, the growing version is closer to our algorithms.

5.2 Common parameters/hardware specs

All tests are executed with $70 \cdot 10^3 \cdot 2^n$ elements where $n \in \mathbb{N}_0$. The reason behind that is that with less than $70 \cdot 10^3$ elements there is a good chance for load-factors close to the respective threshold that a bucket Cuckoo hashing with the chosen hash functions is not possible as there is not enough variety. Therefore, we chose $70 \cdot 10^3$ as our basis. All time measurements are averaged over 5 iterations. Our tests were run on a system consisting of 2 Intel Xeon CPU E5-2683 v4 @ 2.10GHz processors with 16 cores each and 2 threads per core.

5.3 Running times

Figure 5.2 shows a running time plot for the different variants. We chose not to include the growing version of the iterated insertion algorithm because it has a very high running time for these very high load-factors. We can see that the hybrid version is the fastest of our implementations. The cause is that the hybrid implementation has to update much less indexes than the index variant because every prehashed element that is fixed does not cause any index updates. The only extra work compared to the index variant is the prehashing itself and the extra work for every wrong prehashed element.

It is not really surprising that the inplace implementation is mainly the slowest. It requires repeated prehashing which costs the most of its running time. Additionally, the very high load-factors we chose here increase the difficulty even further because our prehashing loses in precision.

In Figure 5.3 we see a running time plot for the two level implementation with the different variants. We chose not to include the growing version of the iterated insertion algorithm again for the same reason as before. For all these n's we are in the case where we try to achieve an optimal subtable size of $70 \cdot 10^3$. Therefore, our running time is proportional to the variant used on that subtable size (see Figure 5.2) leading to the level hybrid being the fastest.

5.4 Load-factor independence

Figure 5.4 shows a running time plot for changing load-factors. As we can see the growing iterated insertion algorithm is exponentially dependent on the load-factor. For a load-factor of 83% and higher any of our variants easily outperform it. The growing iterated insertion algorithm stops sooner than the others in our plots because it is unfeasible for load-factors close to their respective threshold.

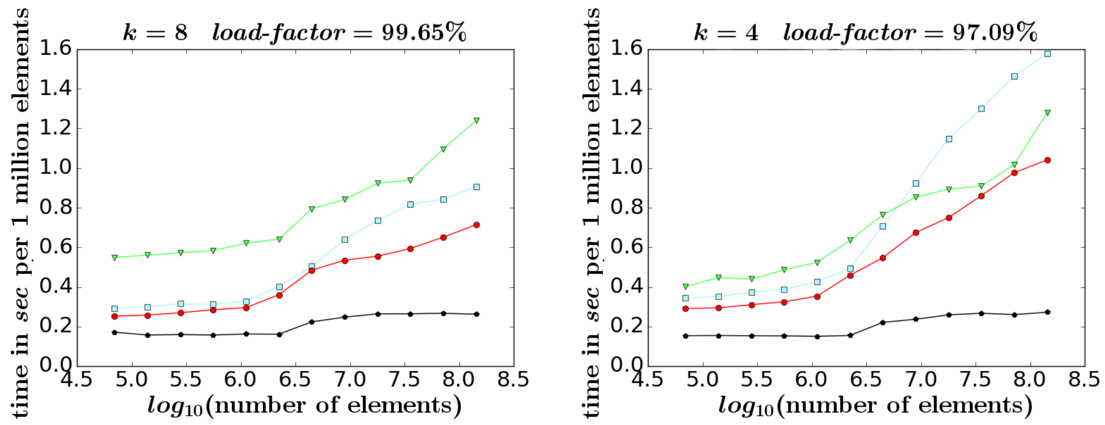


Figure 5.2: Variant running times

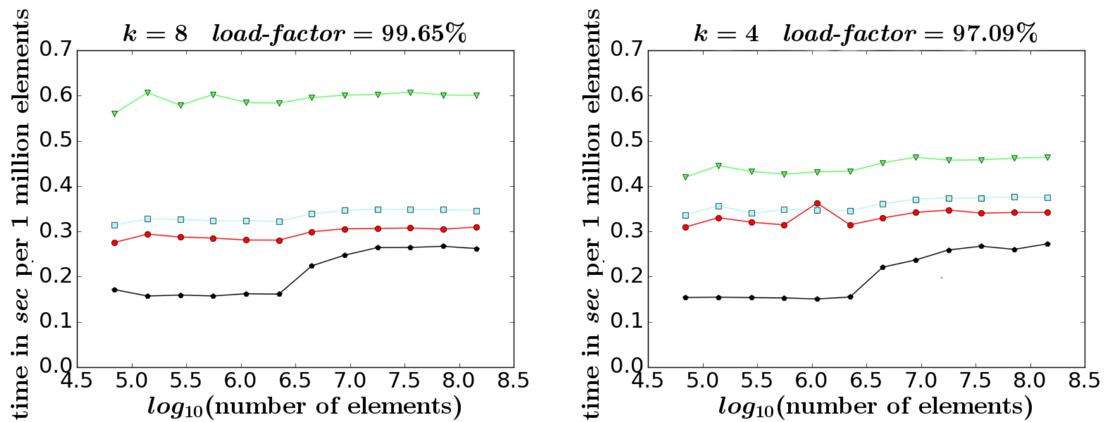


Figure 5.3: Level variant running times

To get a better look at how our implementations behave regarding each other and the non-growing iterated insertion algorithm we take a look at Figure 5.5 which is the same plot without the growing iterated insertion algorithm. As we can see the index and hybrid variant are independent of the load-factor. The slight tendency of the index variant to be a bit faster for low load-factors can be explained by the one-dimensional array for the indexes, see Subsection 4.2.1, which we have to rebuild less often for lower load-factors.

The behaviour of the inplace-variant is dependent on the load-factor. For load-factors between 80% and 98%, however, the inplace-variant is stable. For load-factors higher than 98% our prehashing loses in precision causing the rapid growth of the running time. For low low-factors our instant fixing (see Subsection 4.2.3) becomes more effective, leaving the rest of the algorithm with less elements. Additionally, our prehashing algorithm works better. Therefore, the running time decreases.

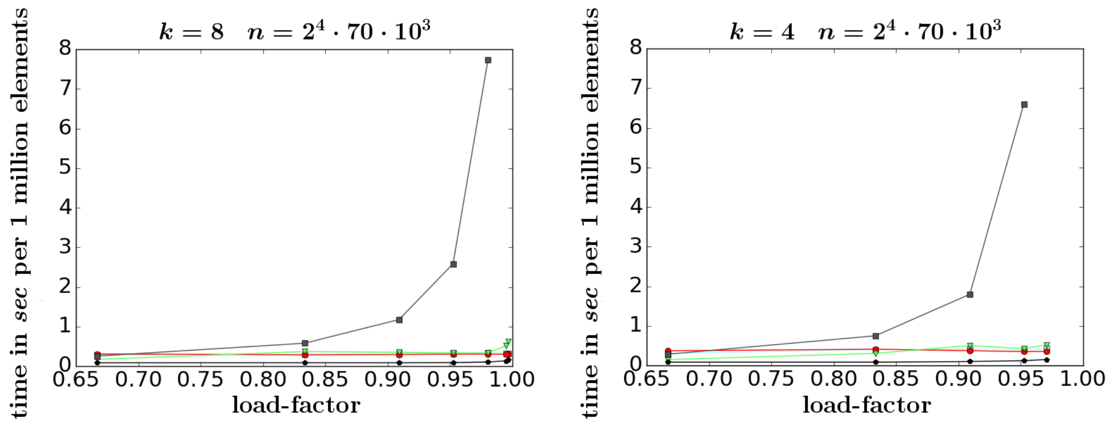


Figure 5.4: Load-factor behaviour

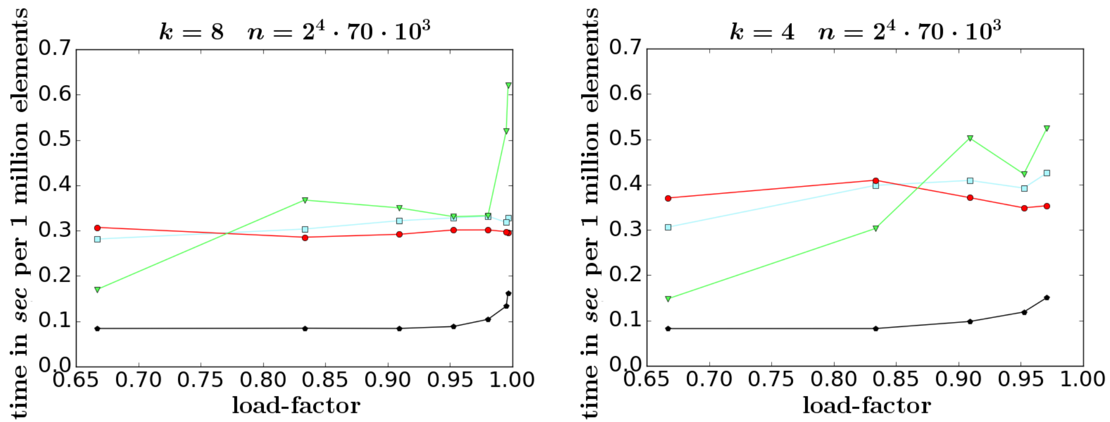


Figure 5.5: Load-factor independence of our variants

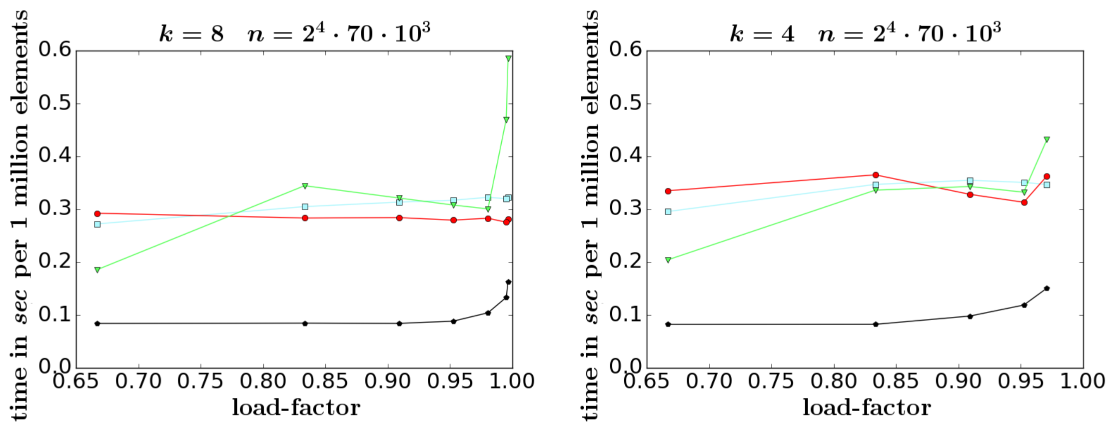


Figure 5.6: Load-factor independence of our level variants

We do not outperform the iterated insertion algorithm. Considering that we work inplace on the table this is not really surprising because insertions in buckets with space still require us to swap elements out of there and, depending on the variant, may also

require us to update indexes, which is costly.

The level variants show the same characteristics as their underlying variants, see Figure 5.6.

5.5 Memory consumption

Figure 5.7 shows how much heap memory each variant consumes. This test was done with approximately 144 million elements and a single iteration. Tests with more iterations concluded that the memory behaviour of the variants is almost constant. Therefore, we chose to only take one iteration here to keep the plot simpler. The data type for our keys and our elements is uint64_t. Note that the memory consumption of our variants is independent of the data type the elements have, whereas the iterated insertion algorithm always uses at least $2 \cdot \textit{table memory}$. Therefore, the iterated insertion algorithms memory consumption is dependent on the element type. The light-blue coloured area is the space the table needs. During the algorithms it is a constantly allocated memory.

To get a better view, let us look at the memory consumption without the table memory (see Figure 5.8). As designed the inplace variant uses far less memory than all others. The reason for that is that the inplace variant does not use indexes in any form which take up most of the memory in the other variants. Additionally, we never store buckets with priority $\textit{expected priority} - 1$ in our PQ vector which make up most of the buckets towards the end. Except for the PQ, which is a vector and for the most part kept implicit, all data structures at the index and hybrid implementation are arrays. Therefore, the main part of the allocation happens instantaneously. The rise later happens because more and more buckets get the *actvpr*, an effect explained in Section 3.3. The slightly higher memory consumption by the hybrid variant is caused by the extra array for the prehashed elements, see Subsection 4.2.2.

The slightly higher memory consumption for $k = 4$ happens because we have twice as much buckets than for $k = 8$. Although it does not affect our primary index array because the indexes per bucket are dependent on d (see Subsection 4.2.1), it does affect our other data structures such as the deg_{in} array, the number of prehashed elements and also the memory of our implicit PQ because we simply have more buckets. The iterated insertion algorithm shows the same effect but for different reasons. The BFS simply has to run deeper because more buckets are full.

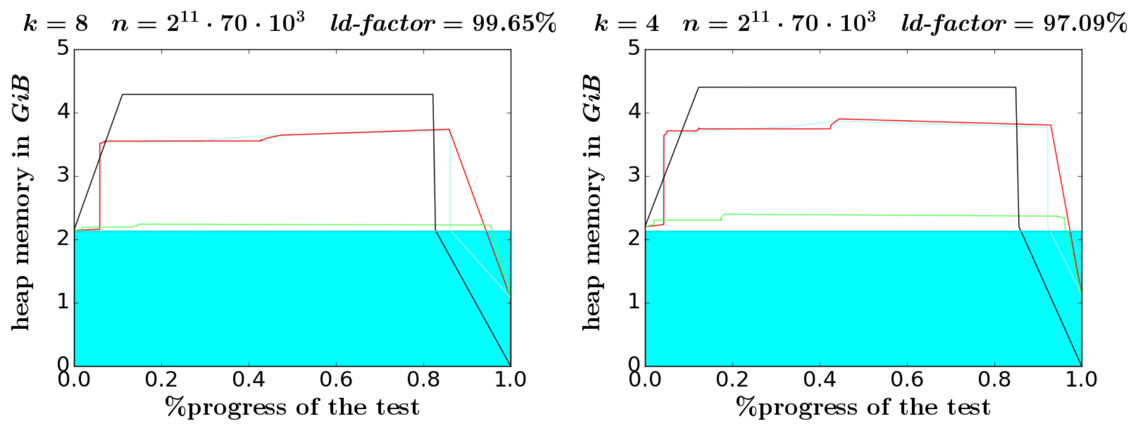


Figure 5.7: Memory consumption with the table

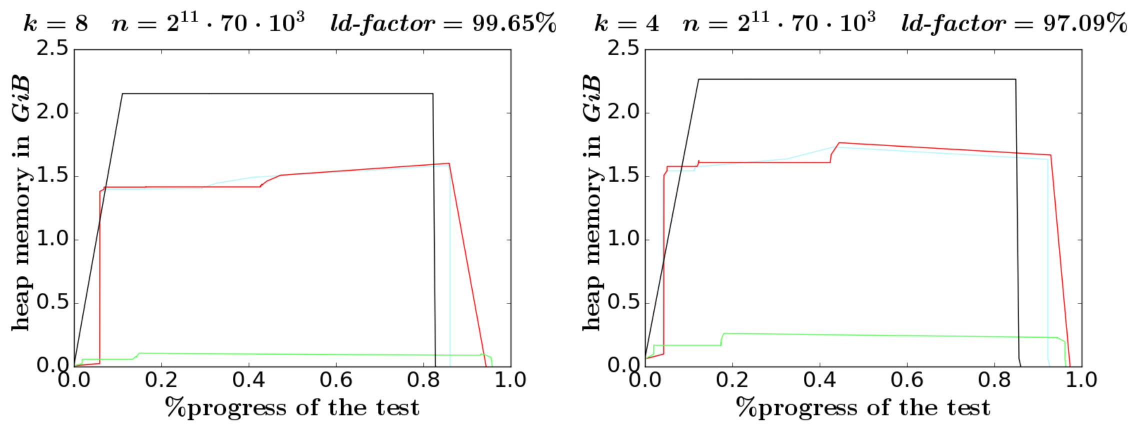


Figure 5.8: Memory consumption without the table

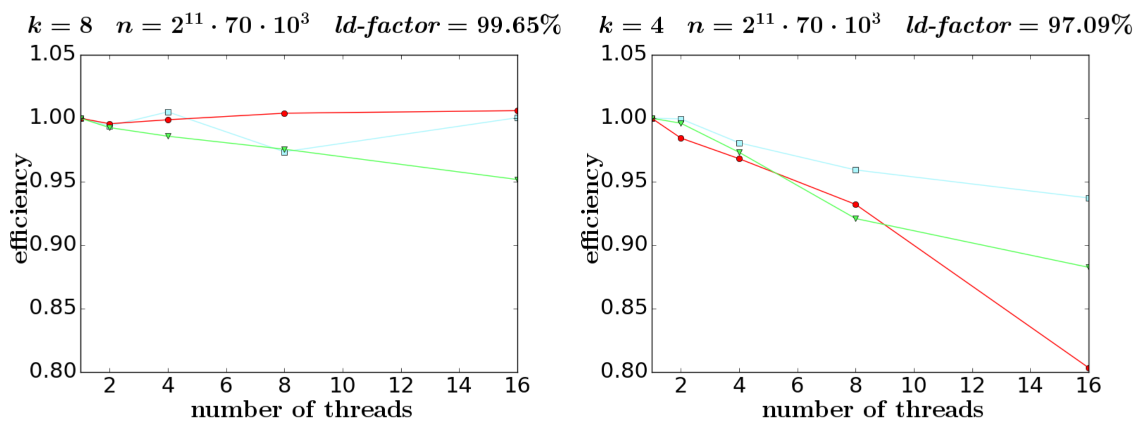


Figure 5.9: Efficiency

5.6 Efficiency

Figure 5.9 shows an efficiency diagram for our parallel versions. Note that this efficiency diagram only considers the parallel part of our algorithm, as explained in Section 4.5 the sequential part can be parallelized using an inplace parallel partitioner, we refer to the one used in [Axt+17]. We can see that we are highly efficient because our efficiency keeps very close to the maximum.

The high parallelism is achieved through our main parallel part which is running the variants on the subtables. We do not need any synchronization or have any dependencies between the subtables. Therefore, the only overhead that our parallel part has is the thread management which is compared to the rest of the algorithm negligible. However, efficiency losses can occur if threads do not finish simultaneously. At these load-factors that are close to their respective thresholds this can occur if in one or more subtables the elements are adversely spread across the buckets.

5.7 Multi hash functions

Table 5.1 and Table 5.2 show the average error rate and average time per one million elements for the different versions of our algorithm (as defined in Section 3.5) before the correction is applied (the correction corrects all errors for the parameters in our tables). We vary the load-factor for two different numbers of elements. As we can see our modifications decreased the error rates significantly.

Version	Load-factor	$n \cdot 70^{-1} \cdot 10^{-3}$	$\varnothing time \cdot 10^{6} \cdot n^{-1}$	$\varnothing$ errors
base1	99.65%	2^7	1.571	7376.8
base2	99.65%	2^7	1.623	1450.0
base3	99.65%	2^7	1.906	8.0
final	99.65%	2^7	1.878	1.0
base1	99.65%	2^{10}	1.882	58881.2
base2	99.65%	2^{10}	1.982	11506.0
base3	99.65%	2^{10}	2.136	69.4
final	99.65%	2^{10}	2.094	8.4
base1	99.90%	2^7	1.564	10980.6
base2	99.90%	2^7	1.586	3880.6
base3	99.90%	2^7	2.055	179.4
final	99.90%	2^7	1.929	22.0
base1	99.90%	2^{10}	1.855	88327.8
base2	99.90%	2^{10}	1.889	30922.2
base3	99.90%	2^{10}	2.118	1400.4
final	99.90%	2^{10}	2.153	184.6
base1	99.95%	2^7	1.637	12089.0
base2	99.95%	2^7	1.642	4721.6
base3	99.95%	2^7	1.885	358.6
final	99.95%	2^7	1.909	55.2
base1	99.95%	2^{10}	1.868	96657.8
base2	99.95%	2^{10}	1.877	38046.8
base3	99.95%	2^{10}	2.150	2712.0
final	99.95%	2^{10}	2.149	415.8
base1	99.98%	2^7	1.625	12763.6
base2	99.98%	2^7	1.578	4538.8
base3	99.98%	2^7	1.958	634.4
final	99.98%	2^7	1.893	111.8
base1	99.98%	2^{10}	1.857	102122.2
base2	99.98%	2^{10}	1.875	43374.0
base3	99.98%	2^{10}	2.119	5071.8
final	99.98%	2^{10}	2.125	877.2

Table 5.1: Error measurements of our different versions for $k = 8$ and $h = 3$

Version	Load-factor	$n \cdot 70^{-1} \cdot 10^{-3}$	$\varnothing time \cdot 10^6 \cdot n^{-1}$	$\varnothing$ errors
base1	97.09%	2^7	1.476	335.0
base2	97.09%	2^7	1.498	0.0
base3	97.09%	2^7	1.569	0.0
final	97.09%	2^7	1.611	0.0
base1	97.09%	2^{10}	1.764	2669.8
base2	97.09%	2^{10}	1.824	0.0
base3	97.09%	2^{10}	1.884	0.0
final	97.09%	2^{10}	2.084	0.0
base1	99.65%	2^7	1.504	7360.8
base2	99.65%	2^7	1.504	1401.0
base3	99.65%	2^7	1.615	9.6
final	99.65%	2^7	1.690	1.2
base1	99.65%	2^{10}	1.788	58918.8
base2	99.65%	2^{10}	1.835	11450.6
base3	99.65%	2^{10}	1.956	77.2
final	99.65%	2^{10}	2.033	9.8
base1	99.90%	2^7	1.466	11061.2
base2	99.90%	2^7	1.504	5219.2
base3	99.90%	2^7	1.663	177.2
final	99.90%	2^7	1.845	22.0
base1	99.90%	2^{10}	1.775	88283.0
base2	99.90%	2^{10}	1.846	30848.0
base3	99.90%	2^{10}	2.008	1397.4
final	99.90%	2^{10}	2.056	179.6
base1	99.95%	2^7	1.504	12314.4
base2	99.95%	2^7	1.505	4702.6
base3	99.95%	2^7	1.670	355.2
final	99.95%	2^7	1.735	54.8
base1	99.95%	2^{10}	1.800	96641.2
base2	99.95%	2^{10}	1.832	38047.2
base3	99.95%	2^{10}	2.001	2912.8
final	99.95%	2^{10}	2.053	422.2

Table 5.2: Error measurements of our different versions for $k = 4$ and $h = 3$

6 Conclusion

6.1 Overview

Our goal was to implement an algorithm solving the bucketized Cuckoo hashing problem inplace and independent of the load-factor. We presented multiple algorithm-variants doing so based on the Selfless(k)-algorithm [CSW07]. These variants either work with a full graph representation or by predicting where elements should be fixed into. One variant even uses a combined approach. We then went further and introduced an algorithm which divides our table into subtables that can then be constructed independently from each other achieving sublinear memory consumption. Our parallelization of this implementation exploiting the independence of the subtables has proven to be highly efficient. Note that we only tested the efficiency for the parallel parts of our algorithms, for the sequential part of our algorithms we referred to a parallel inplace partitioner (see Section 4.5 and Section 6.2).

6.2 Future work

As proposed in Section 4.5 the parallelization of the sequential part of our parallel algorithm using a parallel inplace partitioner, for example the one used in [Axt+17], would be a great improvement because then we would have a fully parallel algorithm. The efficiency of our parallel parts is near one which means the overall efficiency would be solely dependent on the parallel inplace partitioner.

An aspect which we did not cover in this thesis are external algorithms. Let M be the available fast memory (e.g. RAM) and B be the block size of the large external memory. For our analysis we count the number of block accesses. Let H be the memory our table needs. Similar to parallelizing the two level variant of our algorithm (see Section 4.4 and Section 4.5) we could externalize it. We would have to adjust the dividing in the subtables though by introducing an upper bound so that no subtable takes up more than $M - x$ memory where x represents additional memory for the respective variant. This subtable size should also be set as a new goal subtable size.

Because the subtables are calculated independently, it would be enough to keep only one subtable in the memory, construct it, write it back and then get the next subtable. All of our variants use less than $2 \cdot$ *table memory*, therefore this part of our algorithm would have an optimal running time of $O(\frac{H}{M-x} \cdot 2 \cdot \frac{M-x}{B}) \leq O(\frac{H}{\frac{M}{2}} \cdot \frac{\frac{M}{2}}{B}) = O(\frac{H}{M} \cdot \frac{M}{B}) \leq O(2 \cdot \frac{H}{B}) = O(\frac{H}{B})$ with any of our variants (each subtable is read once and written once and we have $\leq \frac{H}{M-x} + 1$

subtables). The calculation of the subtable sizes takes up $O(\frac{H}{B})$ time as well because we simply have to iterate once over the entire table and have to keep one counter per subtable. We achieve that by loading our table block-wise, hence the running time.

The critical part to externalize is the moving of the elements to their appropriate subtables. However, one could use an external partitioner, we refer to [Hu+14], or implement an external partitioner specific to our problem.

Bibliography

[Axt+17] M. Axtmann, S. Witt, D. Ferizovic, and . Sanders. “In-Place Parallel Super Scalar Samplesort (IPSSSSo)”. In: *25th Annual European Symposium on Algorithms (ESA 2017)*. Vol. 87. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 9:1–9:14.

[Aza+99] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. “Balanced Allocations”. In: *SIAM J. Comput.* 29.1 (1999), pp. 180–200.

[CSW07] J. A. Cain, P. Sanders, and N. Wormald. “The random graph threshold for k-orientiability and a fast algorithm for optimal multiple-choice allocation”. In: *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*. SIAM, 2007, pp. 469–476.

[Die+94] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F.Meyer auf der Heide, H.Rohnert, and R. E. Tarjan. “Dynamic Perfect Hashing: Upper and Lower Bounds”. In: *SIAM J. Comput.* 23.4 (1994), pp. 738–761.

[DW05] M. Dietzfelbinger and C. Weidling. “Balanced Allocation and Dictionaries with Tightly Packed Constant Size Bins”. In: *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*. Vol. 3580. Lecture Notes in Computer Science. Springer, 2005, pp. 166–178.

[FKS82] M. L. Fredman, J. Komlós, and E. Szemerédi. “Storing a Sparse Table with O(1) Worst Case Access Time”. In: *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*. IEEE Computer Society, 1982, pp. 165–169.

[FMM09] A. M. Frieze, P. Melsted, and M. Mitzenmacher. “An Analysis of Random-Walk Cuckoo Hashing”. In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, 12th International Workshop, APPROX 2009, and 13th International Workshop, RANDOM 2009, Berkeley, CA, USA, August 21-23, 2009. Proceedings*. Vol. 5687. Lecture Notes in Computer Science. Springer, 2009, pp. 490–503.

[Fot+03] D. Fotakis, R. Pagh, P. Sanders, and P. G. Spirakis. “Space Efficient Hash Tables with Worst Case Constant Access Time”. In: *STACS 2003, 20th Annual Symposium on Theoretical Aspects of Computer Science, Berlin, Germany, February 27 - March 1, 2003, Proceedings*. Vol. 2607. Lecture Notes in Computer Science. Springer, 2003, pp. 271–282.

[FR07] D. Fernholz and V. Ramachandran. “The k-orientability thresholds for $G_{n,\ p}$”. In: *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007.* SIAM, 2007, pp. 459–468.

[Hu+14] X. Hu, Y. Tao, Y. Yang, and S. Zhou. “Finding approximate partitions and splitters in external memory”. In: *26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA ’14, Prague, Czech Republic - June 23 - 25, 2014.* ACM, 2014, pp. 287–295.

[Kar98] R. M. Karp. “Random Graphs, Random Walks, Differential Equations and the Probabilistic Analysis of Algorithms”. In: *STACS 98, 15th Annual Symposium on Theoretical Aspects of Computer Science, Paris, France, February 25-27, 1998, Proceedings.* Vol. 1373. Lecture Notes in Computer Science. Springer, 1998, pp. 1–2.

[Knu98] D. E. Knuth. *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition.* Addison-Wesley, 1998.

[MS17] T. Maier and P. Sanders. “Dynamic Space Efficient Hashing”. In: *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria.* Vol. 87. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 58:1–58:14.

[PR01] R. Pagh and F. F. Rodler. “Cuckoo Hashing”. In: *Algorithms - ESA 2001, 9th Annual European Symposium, Aarhus, Denmark, August 28-31, 2001, Proceedings.* Vol. 2161. Lecture Notes in Computer Science. Springer, 2001, pp. 121–133.

[San04] P. Sanders. “Algorithms for Scalable Storage Servers”. In: *SOFSEM 2004: Theory and Practice of Computer Science, 30th Conference on Current Trends in Theory and Practice of Computer Science, Merin, Czech Republic, January 24-30, 2004.* Vol. 2932. Lecture Notes in Computer Science. Springer, 2004, pp. 82–101.

[SEK03] P. Sanders, S. Egner, and J. H. M. Korst. “Fast Concurrent Access to Parallel Disks”. In: *Algorithmica* 35.1 (2003), pp. 21–55.