Hashing and Random Graphs

Jana Kunert
Department of Computer Science, Aarhus University
20085208
feymour@cs.au.dk
Supervisor: Gerth Stølting Brodal
gerth@cs.au.dk

29th April 2014

Abstract

This thesis studies an idea to obtain simpler constant time evaluable hash functions published by Dietzfelbinger and Woelfel[1] in 2003. They came up with a new way to generate a class of pairs of hash functions and use graph theory to prove the efficiency and practicability of their hash class.
This thesis explains the idea given by Dietzfelbinger and Woelfel in more detail than the article. The first part of the thesis describes a *Dictionary* from a Data Structure perspective and gives some basic theory about *Hashing*; followed by a short presentation of *Cuckoo Hashing*[2], a hashing technique that Rasmus Pagh and Flemming Rodler came up with in 2001.
The second part will introduce some short, but detailed excerpts of Graph Theory, and especially Bipartite Graphs.
The third part introduces the class of hash function pairs created by Dietzfelbinger and Woelfel and establishes a connection between Cuckoo Hashing and Random Bipartite Multigraphs. Using this connection and the basics of Graph Theory introduced in the second part, a more specific subset of Bipartite Graphs is examined in more detail and used to prove the efficiency of the hash class proposed by Dietzfelbinger and Woelfel.
The fourth and final part explains the impact of this hash class on Cuckoo Hashing. Exchanging Siegel's hash functions with hash function pairs created by the class described in the article leads to the same constant time bounds on operations, but achieves a very small constant.

[1]Dietzfelbinger, Martin and Woelfel, Philipp: Almost random graphs with simple hash functions, Symposium on Theory of Computing, ACM 2003

[2]Cuckoo Hashing, Rasmus Pagh and Friche Rodler, European Symposium on Algorithms, 2001

Contents

List of Figures

Chapter 1

Introduction

Hashing is a technique that is used in both Cryptography and a lot of Data Structures. With the amount of data to process getting bigger, efficient storage becomes even more important. Because hash functions are operations on numbers or bits, it comes natural to analyse them algebraically. What distinguishes "Almost Random Graphs with simple Hash functions" by Dietzfelbinger and Woelfel [7] from other publications on Hash functions is, that they use Graph Theory to analyse their Hash Function Class. This is a very interesting idea, that might be usefull in other applications as well. This thesis will therefore give some more detailed background with excerpts from both Hash Functions and Graph Theory; and look at Dietzfelbinger and Woelfels analysis in detail. Hopefully, this will enable some future work using the same principles, and inspire some more research in between different fields in Computer Science.

The main problem discussed in this thesis is the Dictionary Problem. We will start by defining both the static and the dynamic version of this data structure, including the time and space bounds that we want to achieve (Section 2.1); followed by a short description of some Hash strategies (Section 2.2). We will then continue with an introduction to Cuckoo Hashing [6], which will be the technique that we are analysing using the hash function class by Dietzfelbinger and Woelfel. Cuckoo Hashing is easy to explain, but the analysis uses guarantees on hash functions and probabilities to obtain amortised constant time for insertions. We will show in the final chapter that the analysis still holds, but that the structure of the hash function pairs obtained from the class described by Dietzfelbinger and Woelfel gives better (still constant) time bounds.

Chapter 3 contains some important excerpts of Graph Theory, and thereby starts to give some fundamentals used in the proof of the efficiency of Dietzfelbinger and Woelfels hash function class. We define isomorphism for labeled multigraphs and count the number of non-isomorphic graphs with certain properties, to later be able to argue that a Graph G created in a certain way by hash function pairs from the hash function class given by Dietzfelbinger and Woelfel is almost random. Also, we will introduce graph colouring to later bound the probability of subgraphs of G being isomorphic to some given other graph.

Chapter 4 is the main part of this thesis. It draws a connection between bipartite graphs and Cuckoo Hashing, and proves in detail, that the hash function class introduced by Dietzfelbinger and Woelfel is effective.

The final chapter shows how to analyse Cuckoo Hashing with function pairs of this function class instead of Siegel's functions [1]. We will see that this class enables us to choose a very small d, leading to a much smaller constant in the constant time bounds and thus, the article by Dietzfelbinger and Woelfel does indeed matter. They also provide a short overview on how their hash function class can get used in shared memory simulations and d-wise independent hashing without high-degree polynomials. In 2012, they were joined by Martin Aumüller and used it for Cuckoo Hashing with a Stash [8].

It would be great to see their hash function class used in other problems, as well as more extensions of their idea.

Chapter 2

Cuckoo Hashing as solution to the Dictionary Problem

2.1 The Dictionary Problem

One of the very basic data structures is the so-called *Dictionary*: Think of all possible combinations of the letters of the English alphabet, that are of length ≤ 10. Only very few of them are actual words, and we want to be able to check fast, whether a word can be found in an English dictionary. More general, we define this (static) data structure as follows:

Problem 2.1.1 (Static Dictionary)**.**

Instance A subset $S \subset U$, $|S| = n$, $n \ll |U|$.

Query Given some $e \in U$, is $e \in S$?

When we design such a data structure, we judge it by the following criteria:

(1) small space usage

(2) fast queries

(3) short construction (preprocessing) time

We have some very strict lower bounds on all three factors: A data structure that contains n different elements, needs at least $\Omega(n)$ space (1). A query – as any operation – cannot be faster than constant time (2). The preprocessing time spend on building the data structure involves inserting all n elements and therefore equals at least $\Omega(n)$ time (3).

Also, we can easily optimise the space usage by ignoring the query time or vice verse: To only use $\Theta(n)$ space, we sort all elements in S by some one-to-one mapping $U \to U'$ and place them into an array. Using binary search, the query time becomes $\Theta(\log n)$. If we want to optimise the query time, we can use a table of size $|U|$ and place all elements in S into their positions computed by some complete sorting function on U.
What we would like to achieve instead is a data structure that (1) supports constant time queries by (2) only using space linear in n. Furthermore, (3) we want to make sure that the data structure can be constructed in polynomial time. A solution that (in theory) both uses little space and provides fast queries involves *Hashing*.
To get hold of a fast construction time, we will concentrate on the *dynamic* version of the data structure:

Problem 2.1.2 (Dynamic Dictionary)**.**

Instance A subset $S \subset U$, $|S| = n$, $n \ll |U|$. A dictionary containing all elements of S.

Find Given some $e \in U$, is $e \in S$?

Insert Let $S' = S \cup \{e\}$, $e \in U$. Insert e into the dictionary.

Remove Let $S' = S \backslash \{e\}$. Remove e from the dictionary.

2.2 Hashing

The basic idea of hashing for a data structure purpose is the following: We have a somewhat small subset S of a big universe U that we want to store space-efficiently. We choose some function h that maps all elements from the big set U to elements in a much smaller set $[m] = \{0, 1, 2, \ldots, m-1\}$. We then create a table of size m and for each element in S, we find the element in $[m]$ that it is mapped into and save it their. Recall our example, where we look at all combinations of the letters of the English alphabet of length ≤ 10. We could for instance choose to give each letter a number ("A"=0, "B"=1, "C"=2 , ..., "Z"=25), and let the smaller set $[m]$ be all numbers from 0 to 250. For each word, we add the numbers corresponding to its letters to find position in $[m]$. If we want to check whether "DICTIONARY" is part of our dictionary, we compute the corresponding value

$$\begin{array}{ccccccccccccccccccccc} \text{D} & & \text{I} & & \text{C} & & \text{T} & & \text{I} & & \text{O} & & \text{N} & & \text{A} & & \text{R} & & \text{Y} & & \\ 3 & + & 8 & + & 2 & + & 19 & + & 8 & + & 14 & + & 13 & + & 0 & + & 17 & + & 24 & = & 108 \end{array}$$

and check if it placed in position 108 of our table.

Definition 2.2.1 (Hash function)**.** Let U be a universe, let m be a positive integer $m \ll |U|$, and let h be a function $h : U \to [m]$ that maps the universe U into a much smaller image $[m]$. Then h is called a *hash function* from U to $[m]$.

Since the hash set $[m]$ is smaller than the universe U, there must be some *collisions*, viz. at least two elements in U, that hash to the same value in $[m]$. Reusing our example with words in the English language, we find that this applies to "THING" and "NIGHT". Both words consist of the same letters and hence, hash to the same table spot.
Given the fact that the Second edition of the 20-volume Oxford English Dictionary has full entries for more than 170.000 words and our table is of size $m = 251$, we can easily see that we need a bigger table to store them all. Therefore, we need a hash function that hashes to a much bigger set than [251]. More precisely, we want a hash set that is big enough to contain each existing word in its own table spot. Eventually, we would like to find a *perfect hash function*, that actually hashes all values in our dictionary to different table spots.

Definition 2.2.2 (Perfect hash function)**.** Let $S \subset U$ be a small subset of U, and let $h : U \to [m]$ be a hash function from U to $[m]$. If $|h(S)| = |S|$, then h is called a *perfect hash function on S*.

2.2.1 *d*-wise Independent Hash Families

If our dictionary was static, we could just stop here. However, this is usually not the case. Since the set S (containing all the elements of U that are in the dictionary) is dynamic, we cannot construct a perfect hash function. Instead, we want to construct a hash function, that – no matter what S looks like and how we extend it – is very unlikely to have many collisions on S.
A hash function $h : U \to [m]$ must create collisions for at least $|U| - m$ elements in U, and thus, we are not able to find one hash function to cover all possible $S \subset U$. Instead, we use probability theory; and decide to randomly choose some hash function from a family of hash functions, that – for each possible S – only contains very few 'bad' hash functions, i.e. very few hash functions that do not distribute S almost uniformly. To have some proper bounds, we want to use a *Universal Hash Family.*

Definition 2.2.3 (Universal Hash Family)**.** A Family of Hash Functions $\mathcal{H} = \{h : U \to [m]\}$ is called universal, if for a randomly chosen hash function $h \in \mathcal{H}$

$$\forall a, b \in U : \mathbf{Prob}_{h \in \mathcal{H}}[h(a) = h(b) \mid a \neq b] \leq \frac{1}{m}.$$

Definition 2.2.4 (d-wise Independent Hash Family)**.** A family of universal hash functions $\mathcal{H}_m^d = h : U \to [m]$ is called d-wise independent, if for each possible subset of d distinct elements in U, $x_0, x_1, \ldots, x_{d-1}$, a randomly chosen hash function $h \in \mathcal{H}_m^d$, distributes the values $h(x_0), h(x_1), \ldots, h(x_{d-1})$ uniformly and independently.
Let $y_0, y_1, \ldots, y_{d-1}$ be any values in $[m]$.

$$\begin{aligned}\mathbf{Prob}_{h \in \mathcal{H}_m^d}[h(x_0) = y_0, h(x_1) = y_1, \ldots, h(x_{d-1}) = y_{d-1}]\\ \geq \prod_{i=0}^{d-1} \mathbf{Prob}_{h \in \mathcal{H}_m^d}[h_i(x) = y_i] \ .\end{aligned}$$

Clearly, the greater d is chosen, the less likely collisions become for our subset $S \subset U$. Unfortunately, most known constructions for $d \geq 3$ involve polynomials of degree less than but close to d [9]. We will take a short look into Complexity Theory to explain, why this is a problem.

2.2.2 AC^0 vs TC^0

Complexity Theory classifies problems according to their known complexity, i.e., how much time in terms of input size it takes us at least, to find the correct solution to a problem. To compute some lower bound for the complexity of an arbitrary function f, we can look at boolean circuits and try to find minimum requirements for a circuit, that computes our function f. We exclusively look at circuits consisting of AND, OR and NOT gates. There are two types of those classes, NC^k and AC^k, $k \geq 0$, where k is a coefficient for the size and depth of the circuit. NC^k only allows the AND and OR gates to get applied to two input bits, while AC^k allows an unbounded number of input bits to these gate types. We will use the following simplified definition.

Definition 2.2.5 (AC^0)**.** A function f is in AC^0, if $f(x)$ can get computed by a circuit family $\{C_{|x|}\}$, where $C_{|x|}$ consists of AND and OR gates with unbounded input size as well as NOT gates; it has size poly$(|x|) = |x|^{O(1)}$ and constant depth.

We would like our hash functions to be in AC^0, because a circuit with constant depth can get computed it in constant time. Addition, Subtraction and

Division are in AC^0, but Multiplication is shown to be at least in TC^1, and thus, for an input x needs a circuit of depth $O(\log |x|)$.

Definition 2.2.6. A function f is in TC^1, if $f(x)$ can get computed by a circuit family $\{C_{|x|}\}$, where $C_{|x|}$ consists of AND and OR gates with input size 2 as well as NOT gates; it has size poly($|x|$) and depth $O(\log |x|)$.

That means, that the time for the computation of a multiplication is logarithmic in terms of the input size and not constant. Thus, if our hash function h is a high degree polynomial, computation consists of many multiplications and h becomes very slow. If we want our dictionary to be practicable, we have to aim at finding faster solutions.

2.2.3 Space usage

Another problem with hash functions is that we need space to store them. The easiest way of computing a perfect hash function for $S \subset U$ is to store a hash table of size $|S|$. For dynamic dictionaries, a hash function must be defined on all elements of U, and hence, there is no point in storing it as a hash table since the table would need to have size $|U|$ and thus, there would not be any point in hashing. If we use a polynomial of size d,

$$h(x) = \left(\left(a_0 x^d + a_1 x^{d-1} + \ldots + a_{d-1} x + a_d\right) \bmod m\right) ,$$

we need to store all chosen coefficients $a_0, \ldots, a_d$. This means that not only do we have to deal with the multiplication problem, but the better a guarantee we want that the keys are hashed somewhat uniformly, the more space we use, if we stick to d-wise independent hash families, as we already established that the known constructions involve polynomials of a degree close to d.

2.3 Cuckoo Hashing

In 2001, Rasmus Pagh and Flemming Friche Rodler [6] came up with a new approach to build a dictionary, the so-called *Cuckoo Hashing*. Compared with a simple hash table, they double the size of the data structure and keep two tables t_1, t_2 in order to reduce the number of collisions radically. This is done by choosing two hash function h_1 and h_2, and keeping the following invariant:

Invariant 2.3.1 (Cuckoo Hashing)**.** If an element x is in the dictionary, we find it either on position $h_1(x)$ in table t_1 or on position $h_2(x)$ in table t_2 (but never both).

The query time therefore is constant (in theory). In practice, the problem described in Section 2.2.2 applies here as well: Multiplication is not in AC-0 and hence, if we want to be able to query in constant time, we need hash functions where the amount of multiplications does not relate to the size of the data set, nor the size of the data structure (dictionary).

2.3.1 Size

Pagh and Rodler [6] show that it is reasonable to chose the size of each table to be $r \geq (1+\varepsilon)n$ for n being the size of the set $S \subset U$ and some constant $\varepsilon > 0$, such that each table becomes a little less than half-full. Therefore, the whole dictionary uses $2(1+\varepsilon)n = O(n(1+\varepsilon))$ space and Cuckoo Hashing fulfills our requirement of using linear space.

2.3.2 Lookup

If we want to know whether some word x is in the dictionary, we have to look it up in the data structure. Invariant 2.3.1 ensures, that we always can answer such a lookup in constant time: We have to look at the possible position of x in the first table t_1, $h_1(x)$, and the possible position of x in the second table t_2, $h_2(x)$ and return true, if we find x in one of them:

$$\text{return } (t_1[h_1(x)] = x) \wedge (t_2[h_2(x)] = x)$$

If x is not found in one of these two table entries, the invariant ensures, that it is not in the dictionary.

2.3.3 Deletion

To delete an element x from the dictionary, we have to find it first, and then delete it. Once again, we can find out whether x is in the dictionary, and if so, where it is by only looking into the two possible positions $t_1[h_1(x)]$ and $t_2[h_2(x)]$. If we find x in one of these two table entries, we simply delete it from there. Since this does not effect the other elements (words) in the dictionary, there is no cleanup to be done.

2.3.4 Insertion

The crucial operation in a dictionary based on Cuckoo Hashing is insertion. Invariant 2.3.1 gives us exactly two possible cells for the insertion of a new element x, namely $t_1[h_1(x)]$ and $t_2[h_2(x)]$. For the purpose of simplicity in

the description, we will always insert a new element x into the first table t_1. In general, we could also choose to always insert into the second table or alternating insert into the two tables. For the analysis, this makes no difference.
The insertion process gives this technique the name. If $t_1[h_1(x)]$ is empty, we just insert x and are done – but if $t_1[h_1(x)]$ already contains another element x_i, x_i gets thrown out and is now 'nest-less'. We need to move it into it's other possible cell, $t_2[h_2(x_i)]$. If $t_2[h_2(x_i)]$ already contains another element x_j, x_j gets thrown out and becomes the next 'nest-less' element. We need to insert it into $t_1[h_1(x_j)]$, and the element already placed in this cell[1] gets thrown out and is now the 'nest-less' element. This continues until we can place an element into an empty cell – or until we reached a maximum number of insertion tries, *MaxLoop*. If this happens, we choose two new hash functions and rebuild the whole data structure.

2.3.5 Loops

When the insertion of an element takes too much time, we might have encountered a *closed loop*. Therefore, we rebuild the data structure with two new hash functions.

Definition 2.3.2 (Closed loop)**.** Let $h_1, h_2 : U \to [m]$ be two hash functions chosen to build the dictionary, and let $S' \subseteq S$ be a subset of elements in S that are already in the dictionary. Let x be the next element that we want to insert.
If the sum over the number of hash values $h_1(S')$ and $h_2(S')$ equals the total number of elements in S', and if $h_1(x) \in h_1(S')$ and $h_2(x) \in h_2(S')$ i.e.

$$|h_1(S' \cup \{x\})| + |h_2(S' \cup \{x\})| < |S' \cup \{x\}| \ ,$$

then we are trying to fit $|S'| + 1$ elements into $|S'|$ cells. Inserting x will lead us into a *closed loop*, where we continue to hustle elements forward in a circle – and the insert operation will never end.

For a small example of this, look at Figure 2.1.
To prevent this infinite loop, we stop the insertion after *MaxLoop* iterations. Pagh and Rodler show, that by choosing *MaxLoop* to be $\lceil 3 \log_{1+\varepsilon} n \rceil$, the total expected time for a rehash is $O(n)$, and thus, for any insertion, the expected time for rehash is amortised constant.

[1]We always insert new elements into the first table. If we find x_j in the second table, it must have been thrown out of its cell in the first table and hence, there must be an element in $t_1[h_1(x_j)]$.

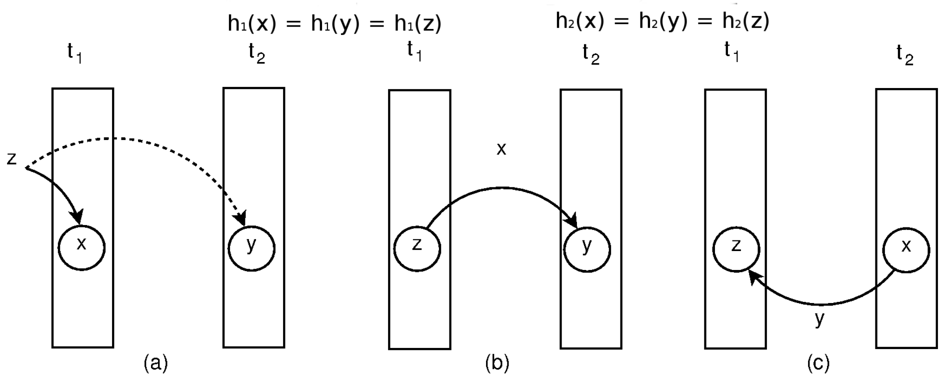


Figure 2.1: Closed loop, a simple example: x, y and z hash to the same table spots in both tables t_1 and t_2. When we insert z (a), x becomes nest-less and throws y out of table 2 (b). Afterwards, y throws z out of table t_1 again (c), and so on.

2.3.6 Re-sizing

If the dictionary is suddenly extended by many new words, or is reduced by some high fraction, the data structure needs to get re-sized to still fulfill our requirement of using linear space. This can be done in expected amortised constant time, for instance with a construction given by Dietzfelbinger et al. [3].

2.3.7 Remarks

In practice, using two hash functions instead of one means, that we need the double amount of random bits. But true random bits are expensive[2], and thus we want to use as few as possible. Furthermore, good random functions are slow[3] or need a lot of space. We want to improve on this.

[2]Not quite anymore, as Intel added a CPU random call to their Ivy Bridge processor line. It cannot be used for Cryptographic purpose as it is suspected to be a backdoor trap from the NSA, but still, the idea behind the architecture might open up for some future development.

[3]Even Siegel's constant time evaluable construction [1], that is used by Pagh and Rodler, uses a huge amount of time due to the very big constant.

2.4 Classes of Pairs of Hash Functions

In 2003, Martin Dietzfelbinger and Philip Woelfel [7] came up with a way to construct a class of pairs of hash functions. Their (simple) construction leads to hash functions with a small constant evaluating time – unlike Siegels construction [1] with a huge constant. The analysis is done by comparison to random bipartite graphs. We will start by looking at some properties of random bipartite graphs and then show how they relate to Cuckoo Hashing. Afterwards, we will look at Dietzfelbinger and Woelfel's construction of a class of pairs of hash functions and use the analogy to bipartite graphs to show that the probability of those functions not acting randomly is very small, as is the probability that those functions do not correspond to graphs with the wanted properties.

Chapter 3

Excerpts from Graph Theory

Dietzfelbinger and Woelfel use properties of random graphs to prove the effectiveness of their hash function family. Thus, we need to take a closer look at some small excerpts from Graph Theory. We will only use undirected graphs, and thus, when using the word *graph*, we implicitly mean *undirected graph.*

3.1 Important properties of (undirected) graphs

An (undirected) graph G has two properties, that are going to be useful for our analysis: The expected length of the longest path starting in an arbitrary vertex and the maximal expected cyclomatic number of a connected component in G.

Definition 3.1.1 (Length of a path in G)**.** The *length of a path p in G* is the total number of edges on p.

Definition 3.1.2 (Cyclomatic number)**.** Let $G = (V, E)$ be a connected graph, and let $T \subseteq E$ be an arbitrary spanning tree of G. The *cyclomatic number* of G is $|E| - |T|$, the number of edges of G not in T.

Lemma 3.1.3. *The cyclomatic number of an (undirected) connected graph $G = (V, E)$ equals $|E| - |V| + 1$.*

Proof. Since a spanning tree on V vertices has $|V| - 1$ edges, the cyclomatic number of a connected graph with $|E|$ edges on $|V|$ vertices is defined to be

$$\begin{aligned}\#\text{ edges in } G - \#\text{ edges in spanning tree} &= |E| - (|V| - 1)\\ &= |E| - |V| + 1\ .\end{aligned}$$

□

Furthermore, we are going to need a definition of when two graphs are isomorphic, as we want to count graphs. We are working with labeled graphs, and thus, our definition of isomorphism will include the labels.

Definition 3.1.4. Let $H = (V_H, E_H)$ and $H' = (V_{H'}, E_{H'})$ be two graphs labeled with elements from some set $S \subseteq U$, E_H and $E_{H'}$ being multisets. We call H and H' isomorphic if and only if there exist two bijections $\sigma : V_H \to V_{H'}$ and $\tau : E_H \to E_{H'}$ such that for all $e \in E_H$ connecting $u, v \in V_H$, $\tau(e) \in E_{H'}$ has the same label as $e \in E_H$ and connects $\sigma(u), \sigma(v) \in V_{H'}$.

3.2 Important properties of multigraphs

Definition 3.2.1. The *degree of* a vertex v describes the number of edges incident to v.

We use the degree to classify the edges in G to be able to count non-isomorphic undirected multigraphs.

leaf edge An edge is called a leaf edge, if it has an endpoint with degree 1.

inner edge An edge is called an inner edge, if it is not a leaf edge.

cycle edge An edge is called a cycle edge, if it lies on a cycle.

We define two numbers $N(k, l, q)$ and $N^*(k, p)$ that we will use to find an upper bound on the probability that a random bipartite graph is isomorphic to a connected graph with a given cyclomatic number.

Definition 3.2.2 ($N(k, l, q)$)**.** Let $N(k, l, p)$ be the number of non-isomorphic connected graphs on k edges, whose cyclomatic number is q and which have l leaf edges.

Definition 3.2.3 ($N^*(k, p)$)**.** Let $N^*(k, p)$ be the number of non-isomorphic connected graphs on k edges, of which p edges are either cycle or leaf edges.

First of all, we need an upper bound on the number of non-isomorphic graphs with a given cyclomatic number. Lemma 3.2.4 is used to prove Lemma 3.2.5 and Lemma 3.2.6, both of which are going to be needed to examine the hash function family given by Dietzfelbinger and Woelfel. All three lemmas are given by them [7], as Lemma 2a-c. Down below, the proofs are slightly extended.

Lemma 3.2.4. $N(k, l, 0) \leq k^{2l-4}$.

Proof. All connected graphs G with cyclomatic number 0 are trees.

Let G_i be a subtree of G with i leaf edges. To construct any tree consisting of k edges with $l \geq 2$ leaf edges (and thus, $k - l$ inner edges), we start by choosing a path G_2 of length $k_2 \in [2, k - (l - 2)]$. For $i = 3, 4, \ldots, l$, we can construct G_i by adding a new path of length k_i to G_{i-1}, such that $k_2 + k_3 + \ldots + k_i \leq k - (l - i)$, starting in an arbitrary (non-leaf) vertex in G_{i-1}. As long as $k_2 + k_3 + \ldots + k_i \leq k - (l - i)$, there are at least $l - i$ edges left that can get added, and thus, we are able to add all remaining leaf edges (and possibly some inner edges as well). The length of the last path is determined by $k_l = k - (k_2 + k_3 + \ldots + k_{l-1})$.
In each of the $l-2$ transitions from G_{i-1} to G_i, there are $\leq k$ possible starting points for the new path, as there are $< k$ edges in any G_{i-1}. Moreover, because there are k edges in total, each k_i can only be chosen in $[1, k[$ and thus, there are $< k$ possible choices for each of the $l-2$ numbers k_i. Therefore, the total number of non-isomorphic trees on k edges is bounded as stated above:

$$\begin{aligned} N(k, l, 0) &\leq k^{l-2} \cdot k^{l-2} = k^{2(l-2)} \\ &= k^{2l-4} \ . \end{aligned}$$

□

Lemma 3.2.5. $N(k, l, q) = k^{O(l+q)}$.

Proof. Lemma 3.2.4 shows that this holds for $q = 0$:

$$k^{2l-4} = k^{O(l)} = k^{O(l+q)} \ .$$

For $q > 0$, by our definition of the cyclomatic number, we can iteratively remove q cycle edges such that only a spanning tree is left. Every time we remove a cycle edge, we might – but do not necessarily – convert two inner edges to leaf edges. Thus, the spanning tree will have l' leaf edges,

$$l \leq l' \leq l + 2q \ ,$$

and the number of the possible spanning trees is $N(k - q, l', 0)$.
Each of the removed cycle edges has less than k^2 possibilities for its two endpoints and therefore, we can estimate $N(k, l, q)$ as follows:

$$\begin{aligned} N(k, l, q) &\leq (k^2)^q \cdot N(k - q, l', 0) \\ &\leq k^{2q} \cdot (k - q)^{2l'-4} \\ &\leq k^{2q} \cdot (k - q)^{2(l+2q)-4} \\ &= k^{O(k+q)} \ . \end{aligned}$$

□

Lemma 3.2.6. $N^*(k,p) = k^{O(p)}$.

Proof. For simplicity, we will use the following number in our proof.

$N^*(k,p,q)$ the number of non-isomorphic connected graphs on k edges whose cyclomatic number is q and which have p edges that are either cycle edges or leaf edges.

By reducing our analysis to graphs with a certain cyclomatic number, it becomes much simpler. We will conclude the proof by summing over all $N^*(k,p,q)$ for $0 \leq q \leq p$ to find a bound on $N^*(k,p)$.
For $q = 0$, the graphs are trees and thus, do not contain any cycle edges. Therefore, they contain p leaf edges and according to Lemma 3.2.4

$$N^*(k,p,0) = N(k,p,0) \leq k^{2p-4} .$$

For $q > 0$, we can use the same technique as in the proof of Lemma 3.2.5: We can remove q cycle edges such that only a spanning tree T is left, that consists of $k - q$ edges. When a cycle edge e is removed, there is once again the possibility, that we convert one or two edges to leaf edges. But this can only happen for edges that lie on the cycle that we are splitting up, and hence, such edges are cycle edges that now get converted to leaf edges. Therefore, the number of leaf edges in T must be bounded by $\leq (p-q)$, as p is the number of edges in G that are either leaf edges or cycle edges and q is the number of cycle edges that have been removed from G to obtain T.
There are $N^*(k-q, p-q, 0)$ possible spanning trees T on $k-q$ edges with $p-q$ leaf edges.
Each of the removed cycle edges has less than k^2 possibilities for its two endpoints and therefore, we can estimate $N^*(k,p,q)$ as follows:

$$\begin{aligned} N^*(k,p,q) &\leq k^{2q} \cdot N^*(k-q, p-q, 0) \\ &= k^{2q} \cdot N(k-q, p-q, 0) \\ &\leq k^{2q} \cdot (k,q)^{2(p-q)-4} < k^{2q+2(p-q)-4} \\ &= k^{2p-4} . \end{aligned}$$

Now we sum up over all $N^*(k,p,q)$ for $0 \leq q \leq p$, as the cyclomatic number cannot exceed the number of cycle edges.

$$\begin{aligned} N^*(k,p) &= \sum_{q=0}^{p} N^*(k,p,q) \\ &\leq p \cdot k^{2p-4} . \end{aligned}$$

As the number of cycle edges cannot exceed the total number of edges, we can bound $N^*(k,p)$ by

$$p \cdot k^{2p-4} \leq k \cdot k^{2p-4} = k^{2p-3} = k^{O(p)} \; .$$

□

3.3 Important properties of bipartite graphs

If the vertices of a graph $G = (U \cup V, E)$ can get parted into two disjoint sets U and V, such that the every edge $e \in E$ has one end points in U and the other end point in V, G is called a *bipartite* graph.

Definition 3.3.1 (k-colourable)**.** A graph $G = (V, E)$ is k-colourable, if there exists a function $f : V \to [k]$, that assigns a colour $0, 1, \ldots, k-1$ to each vertex $v \in V$, such that for each edge $(u,v) \in E$, the two endpoints are coloured differently, $f(u) \neq f(v)$.

Lemma 3.3.2. *Every bipartite graph is 2-colourable.*

Proof. The vertices $U \cup V$ in a bipartite graph G can get split into two sets U and V in such a way, that each edge has one end point in U and one end point in V. If we colour all vertices $u \in U$ with 0 and all vertices $v \in V$ with 1, all vertices in G are coloured and for each edge (u,v), the two endpoints have different colours. □

Lemma 3.3.3. *There are exactly two 2-colourings for a connected bipartite graph.*

Proof. There are at least two 2-colourings for a connected bipartite graph $G = (U \cup V, E)$ with $E \subseteq U \times V$: Either all vertices $u \in U$ are coloured with 0 and all vertices $v \in V$ are coloured with 1 – or vice verse.
Assume for contradiction, that there exists a third colouring $f : U, V \to [2]$. For f to be different from the first two colourings, either U or V must contain nodes of both colours. Assume without loss of generality that there exists a vertex $u_0 \in U$ that has a colour different from all other vertices in U. G is connected and thus, there exists a path from u_0 to each other vertex in U, for instance u_1, $f(u_0) \neq f(u_1)$. Because G is bipartite, each second vertex on the path from u_0 to u_1 lies in V and thus, the length of the path is even. For c to be a valid 2-colouring, the vertices on the path from u_0 to u_1 must be coloured alternating, and therefore, $f(u_0)$ must equal $f(u_1)$, which is a contradiction. □

Chapter 4

Almost Random Graphs with simple Hash functions

Now that we have established both basic Hash Functions and some important properties of Bipartite Graphs, we can look at the hash function class defined by Martin Dietzfelbinger and Phillipp Woelfel and show how we can use Graph Theory to prove its efficiency.

4.1 A class of hash function pairs

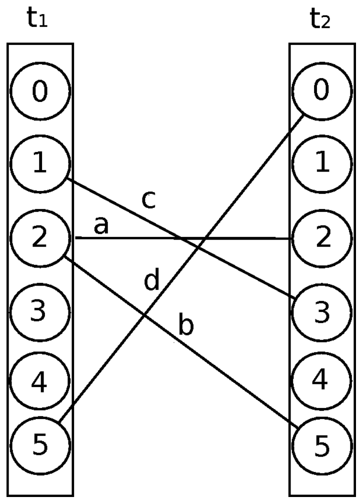
Dietzfelbinger and Woelfel chose not to create a new class of hash functions, but a class of *pairs* of hash functions, $\hat{\mathcal{R}}^d_{r,m}$. It is defined as follows:

Definition 4.1.1. Let $d \geq 2$ and $r, m \in \mathbb{N}$. For $f \in \mathcal{H}^d_m$, $g \in \mathcal{H}^d_r$, and $z = (z_0, z_1, \dots, z_{r-1}) \in [m]^r$ the hash function $h_{f,g,z} : U \to [m]$ is defined by $x \to \Big(f(x) + z_{g(x)}\Big) \pmod{m}$.

Definition 4.1.2. The hash class $\mathcal{R}^d_{r,m}$ is the family of all functions $h_{f,g,z}$.

Definition 4.1.3. The family $\hat{\mathcal{R}}^d_{r,m}$ consists of all hash function pairs (h_1, h_2), where $h_i = h_{f_i,g,z^{(i)}}$ with $f_1, f_2 \in \mathcal{H}^d_m, g \in \mathcal{H}^d_r$ and $z^{(1)}, z^{(2)} \in [m]^r$.

The idea of a hash function class very much alike $\hat{\mathcal{R}}^d_{r,m}$ was already examined by Dietzfelbinger and Meyer auf der Heide [2], and appears again in different work [4], [5]. Previous analysis had never looked on the behaviour of the functions on a given key set S, but concentrated on the use of single functions from $\mathcal{R}^d_{r,m}$. The article from 2003 however, uses graph theory to examine the behaviour when using function pairs from $\hat{\mathcal{R}}^d_{r,m}$ for Cuckoo Hashing. To introduce their main theorem, we need to draw a connection between Cuckoo Hashing and Bipartite Graphs.



x	$h_1(x)$	$h_2(x)$
a	2	2
b	2	5
c	1	3
d	5	0

Figure 4.1: Simple example of a graph $G(S, h_1, h_2)$ with $S = \{a, b, c, d\}$ and $m = \left(1 + \frac{1}{4}\right) n$.

4.2 Cuckoo Hashing and Bipartite Graphs

Look at the following scenario: Assume a set S of n keys from a universe U and some range $[m]$, $m \ll |U|$. Choose two hash functions $h_1, h_2 : U \to [m]$ at random (according to some distribution[1]). We look at the bipartite multigraph on the vertices $V \uplus W$ with $V = [m]$ and $W = [m]$ and assume that $m \geq (1 + \varepsilon)n$ for some $\varepsilon > 0$. For each $x \in S$, we add an edge between $h_1(x)$ in V and $h_2(x)$ in W.

Definition 4.2.1 ($G(S, h_1, h_2)$)**.** Let $S \subseteq U$ be a subset of a universe U, and let $h_1, h_2 : U \to [m]$ be two hash functions chosen at random that hash all elements in U into $[m]$. Furthermore, let V and W be two independent sets of vertices, labeled with the numbers $0, 1, \ldots, m-1$, and let $E = \{(h_1(x)_V, h_2(x)_W) \mid x \in S\}$ be a set of (undirected) edges from V to W. Then we define the bipartite graph $(V \oplus W, E)$ as $G(S, h_1, h_2)$.

For a simple example, see Figure 4.1.
We are particularly interested in graphs with

(a) the expected length of the longest path bounded by a constant and

[1]In the context of randomised algorithms we usually work with some kind of limited randomness instead of assuming complete randomness and hence, we cannot use the tools known from the theory of random graphs. We choose the hash functions from universal hash classes and use functions that have a moderate representation size and a small evaluation time.

(b) the cyclomatic number of each connected component bounded by 1.

Why these two properties are of importance becomes clear, when we map the set S and the hash functions h_1, h_2 onto Cuckoo Hashing instead. Let S be the set of elements we want to insert into the dictionary, and let h_1 and h_2 be the hash functions that we are using. Now the length of the longest path in graph G starting in vertex x becomes the maximum number of nestless keys during the insertion of x into the dictionary[2]. Thus, the expected length of the longest path starting in an arbitrary vertex gives an (expected) upper bound on the insertion for each elements in S. If this number is bounded by a constant, the expected insertion time is constant as well.
If we map Cuckoo Hashing onto G, a the set of vertices V' in a connected component in G corresponds to a subset of table entries, while the set of edges corresponds to the subset $S' \subseteq S$ that hashes into the subset of table entries V'. As stated in Section 2.3.5, we cannot build the data structure if we try to fit $|S'|$ elements into less than $|S'|$ cells. However, this will only be the case if the cyclomatic number of the subgraph (V', S') is greater than 1. For h_1, h_2 to be a successful hash pair, we need the graph $G(S, h_1, h_2)$ to only have connected components with cyclomatic number ≤ 1.

4.3 Truly random bipartite subgraphs

To be able to analyse the bipartite graphs created by function pairs from the hash class in the main theorem in the next section, we will need some more information on the behaviour of truly random connected bipartite (sub)graphs. This is due to the later shown fact that if $G(S, h_1, h_2)$ is created using a hash function pair $(h_1, h_2) \in \hat{\mathcal{R}}^d_{r,m}$, then it will contain a lot of truly random subgraphs.

Definition 4.3.1 (K(T))**.** We construct a graph $G(S, h_1, h_2)$ using two hash functions $h_1, h_2 : U \to [m]$ and a set $S \subseteq U$. Let each edge $(h_1(x), h_2(x))$ in $G(S, h_1, h_2)$ be labeled with the element $x \in S$. We define $K(T)$ to be a subgraph of G, containing only the edges $T \subseteq S$, and disregarding all vertices with degree 0.

See Figure 4.2 for an example.

Definition 4.3.2 (Special edge in K(T))**.** An edge $(h_1(x), h_2(x))$ in $K(T)$ is called *special* if there exists some other key $x' \in T$, $x' \neq x$, such that $g(x) = g(x')$.

[2] i.e. the number of times that a hash function is used in worst case, to insert x

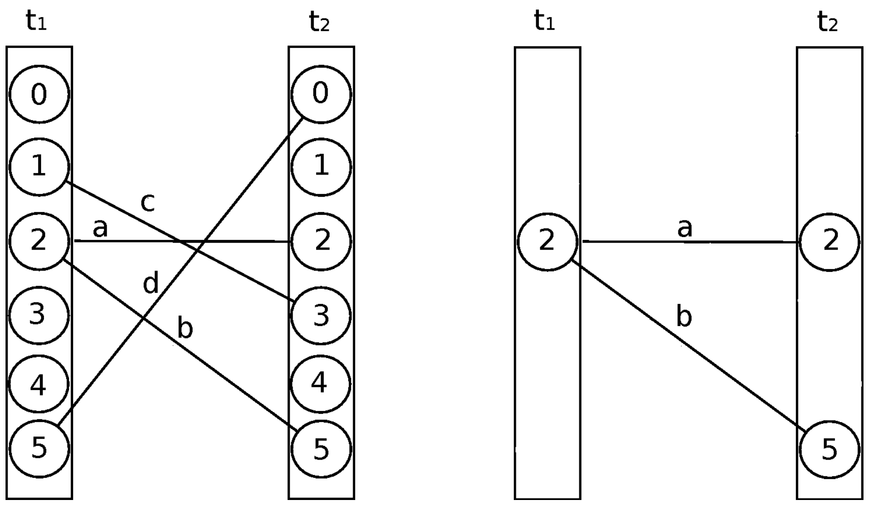


Figure 4.2: $G(S, h_1, h_2)$ shown in Figure 4.1 (left) and the corresponding $K(T)$ for $T = \{a, b\}$ (right)

Recall definition 3.1.4 that defines when two graphs are isomorphic.

Lemma 4.3.3. *Let $T \subseteq U$ and let $H = (V_H, E_H)$ be a bipartite, connected graph, where each edge is labeled with a unique element in T. If the values for $h_1(x), h_2(x)$, $h_1, h_2 : U \to [m]$ are chosen fully randomly for all $x \in T$, then*

$$\mathbf{Prob}[K(T) \textit{ is isomorphic to } H] \leq 2 \cdot m^{-|E_H|-q-1} ,$$

where q is the cyclomatic number of H.

Proof. Let $\deg(v)$ be the degree of vertex v. If $K(T)$ is isomorphic to H, the following must apply to all vertices $v \in V_H$: For all edges $e_0, e_1, \ldots, e_{\deg(v)-1} \in E_H$ adjacent to v, either $h_1(e_0) = h_1(e_1) = \ldots = h_1(e_{\deg(v)-1})$ or $h_2(e_0) = h_2(e_1) = \ldots = h_2(e_{\deg(v)-1})$.
Because H is a connected bipartite graph, we can colour it with two colours $i = \{1, 2\}$ (Lemma 3.3.2). Furthermore, H can only get coloured in two different ways (Lemma 3.3.3(1)).
The probability that $K(T)$ is isomorphic to H is bounded by the probability that there exists a 2-colouring of H such that for each vertex $v \in V_H$ the following is true: If v has colour i and v is incident to the edges $e_0, e_1, \ldots, e_{\deg(v)-1} \in S$, then $h_i(e_0) = h_i(e_1) = \ldots = h_i(e_{\deg(v)-1})$. Let this event for one vertex v be defined as $A(v)$.

All random values $h_i(x)$ are chosen independently and thus,

$$\mathbf{Prob}[A(v)] = \frac{1}{m^{\deg(v)-1}}$$

$$\mathbf{Prob}[\forall v \in V_H : A(v)] = \prod_{v \in V_H} \frac{1}{m^{\deg(v)-1}} \qquad (2)$$

Combining (1) and (2) we get that the probability that H is isomorphic to $K(T)$ is at most

$$\begin{aligned} 2 \cdot \prod_{v \in V_H} \frac{1}{m^{\deg(v)-1}} &= 2 \cdot \frac{1}{m^{\sum_{v \in V_H} \deg(v) - |V_H|}} \\ &= 2 \cdot m^{|V_H| - \sum_{v \in V_H} \deg(v)} \\ &= 2 \cdot m^{|V_H| - 2|E_H|} . \end{aligned}$$

Lemma 3.1.3 tells us that the cyclomatic number equals the following sum:

$$q = |E_H| - |V_H| + 1 \Leftrightarrow |V_H| = |E_H| - q + 1$$

and thus, the probability that H is isomorphic to $K(T)$ is at most

$$2 \cdot m^{|V_H| - 2|E_H|} = 2 \cdot m^{|V_H| - |E_H| - q + 1}$$

as stated. □

4.4 The Main Theorem

Dietzfelbinger and Woelfel show that there exist function pairs in their hash function class $\hat{\mathcal{R}}^d_{r,m}$ that meet our criteria from Section 4.2 and furthermore, that it is very likely that we choose such a function pair if we choose some function pair from this class at random.

Theorem 4.4.1. *Let $\varepsilon > 0$ and $l \geq 1$ be fixed and let $m = m(n) = (1+\varepsilon)n$. For any set $S \subseteq U$, $|S| = n$, there exists a set $R(S)$ of 'good' hash function pairs in $\hat{\mathcal{R}}^{2l}_{r,m}$, such that for randomly chosen $(h_1, h_2) \in \hat{\mathcal{R}}^{2l}_{r,m}$ the following holds.*

(a) $\boldsymbol{Prob}[(h_1,h_2) \in R(S)] \geq 1 - \frac{n}{r^l}$.

(b) For every constant $q \geq 2$ the probability that $(h_1, h_2) \in R(S)$ and that there is a connected subgraph of $G(S, h_1, h_2)$ whose cyclomatic number is at least q, is $O\left(n^{1-q}\right)$.

(c) For every key $x \in S$, the probability that $(h_1, h_2) \in R(S)$ and that in the graph obtained from $G(S, h_1, h_2)$ by removing the edge $(h_1(x), h_2(x))$ the vertex $h_1(x)$ is the endpoint of a simple path of length t, is bounded by $(1+\varepsilon)^{-t}$.

4.5 Good function pairs

Looking at the hash function pair class $\hat{\mathcal{R}}^d_{r,m}$, we will now further identify, what a *good* hash function pair is. Recall that each function pair consists of $h_1 = h_{f_1,g,z^{(1)}_{g(x)}}$ and $h_2 = h_{f_2,g,z^{(2)}_{g(x)}}$, determined by the d-wise independent hash functions f_1, f_2, g and the random offset vectors $z^{(1)}_{g(x)}$ and $z^{(2)}_{g(x)}$. In the Main Theorem 4.4.1, we stated that there exists some subset of the hash function pairs in $\hat{\mathcal{R}}^d_{r,m}$, $R(S)$, as the set of *good* hash function pairs for S. We now define this set.

Definition 4.5.1 ($R^*(T)$)**.** Let $S \subseteq U$. For $T \subseteq S$, the set $R^*(T)$ consists of those hash function pairs (h_1, h_2), whose g function satisfies

$$|g(T)| \geq |T| - l \ ,$$

i.e. the image of T created by g is of size $\geq |T| - l$ and thus, for a reasonable l, g distributes T rather well.

Definition 4.5.2 (l-bad)**.** The graph $G = G(S, h_1, h_2)$ is called l-bad if there exists a subset $T \subseteq S$ such that $K(T)$ is connected and the hash function pair (h_1, h_2) is *not* in $R^*(T)$.

Definition 4.5.3 (R(S))**.** The set $R(S) \in \hat{\mathcal{R}}^{2l}_{r,m}$ contains those hash function pairs (h_1, h_2) for which the graph $G(S, h_1, h_2)$ is *not* l-bad.

In the next theorem, we analyse the properties of $R^*(T)$ further. It is important for us, that we can show that each connected component in $G(S, h_1, h_2)$ is *practically random* if we choose the hash function pair (h_1, h_2) randomly from $R^*(T) \cap R(S)$, such that we can use Lemma 4.3.3. Furthermore, we want to make sure that if a function pair (h_1, h_2) is chosen from $R(S)$, then (h_1, h_2) lies in $R^*(T)$ for all $T \subseteq S$ that are contained in a connected component of $G(S, h_1, h_2)$. If a hash function pair lies in $R^*(T) \cap (\hat{\mathcal{R}}^{2l}_{r,m} - R(S))$, we do not know how it behaves. However, this set will be proven to be very small when we prove part (a) of the Main Theorem 4.4.1.
For now, we want to prove that a subgraph $K(T)$ is a random subgraph if the hash function pair (h_1, h_2) is chosen from $R^*(T)$; and that this will be the case if (h_1, h_2) is chosen from $R(S)$ and $K(T)$ lies in a connected component in $G(S, h_1, h_2)$.

Theorem 4.5.4. *Let $\varepsilon > 0$ and $l \geq 1$ be fixed, let $m = (1+\varepsilon)n$, and consider some $S \subseteq U$, $|S| = n$. Use Definition 4.5.3 for $R(S)$. We claim that for each subset $T \subseteq S$ $R^*(T)$ has the following two properties:*

(1) If $(h_1, h_2) \in R(S)$ and $K(T)$ is a subgraph of one connected component in $G(S, h_1, h_2)$, then $(h_1, h_2) \in R^*(T)$.

(2) *If (h_1, h_2) is chosen at random from $R^*(T)$, then the image pair $(h_1(x), h_2(x))$ is uniformly and independently distributed in $[m]^2$ for $x \in T$.*

Proof. (1) Let $(h_1, h_2) \in R(S)$ and let $K(T)$ be contained in a connected component K' in G, containing all elements from some set $T' \subseteq S$ such that $T \subseteq T'$.

Suppose for contradiction, that $(h_1, h_2) \notin R^*(T)$, which means that $|g(T)| < |T| - l$. If $|g(T)| < |T| - l$, then $|g(T')| < |T'| - l$ as well, as the image of $T' \backslash T$ cannot be bigger than $T' \backslash T$, i.e. $|g(T')| - |g(T)|$ must be less than or equal $|T'| - |T|$. Therefore, $T' \subseteq S$ is a set for which $K(T')$ is connected in $G(S, h_1, h_2)$, but (h_1, h_2) is not in R^* by our assumption.

Per Definition 4.5.2 this means that $G(S, h_1, h_2)$ is l-bad and thus, the hash function pair (h_1, h_2) cannot be in $R(S)$, which is a contradiction and proves part (1) of the theorem.

(2) Let (h_1, h_2) be chosen at random from $R^*(T)$, i.e. $|g(T)| = |T| - l'$ for some $l' \leq l$. We fix g and pick $|T| - l'$ keys from T such that they all hash into different values and call this set T_1, $|T_1| = |g(T_1)| = |T| - l'$.

$$\begin{aligned} T = T_1 \cup T_2 \text{ with } & |T_1| = |g(T_1)| = |T| - l' \\ & |T_2| = l' \geq |g(T_2)| \, . \end{aligned}$$

The remaining l' values, that we will call T_2, hash into $\leq l'$ of the values already covered by T_1.

$$g(T_2) \subseteq g(T_1)$$

Let $T_1' \subseteq T_1$ be the subset of T_1 that hashes into the same $\leq l'$ values as T_2 (see Figure 4.3).

$$T_1' \subseteq T_1 : g(T_1') = g(T_2)$$

All elements in T_1 hash into different values, and thus, $|T_1'| = |g(T_2)| \leq l'$. Therefore, we can give the following upper bound on the size of $T_1' \cup T_2$:

$$|T_1' \cup T_2| \leq |T_1'| + |T_2| \leq 2l' \leq 2l$$

We fix the offsets $z_i^{(1)}$ and $z_i^{(2)}$ randomly for all $i \in g(T_2)$. Recall that f_1 and f_2 are both $2l$-independent, and thus,

$$h_1(x) = \left(f_1(x) + z_{g(x)}^{(1)}\right) \pmod m \text{ and } h_2(x) = \left(f_2(x) + z_{g(x)}^{(2)}\right) \pmod m$$

are distributed independently and uniformly in $[m]$ for all $x \in T_1' \cup T_2$. This leaves us with $g(T_1 - T_1')$. Because of our definition of T_1'

$$g(T_1 - T_1') \cap g(T_1' \cup T_2) = \emptyset$$

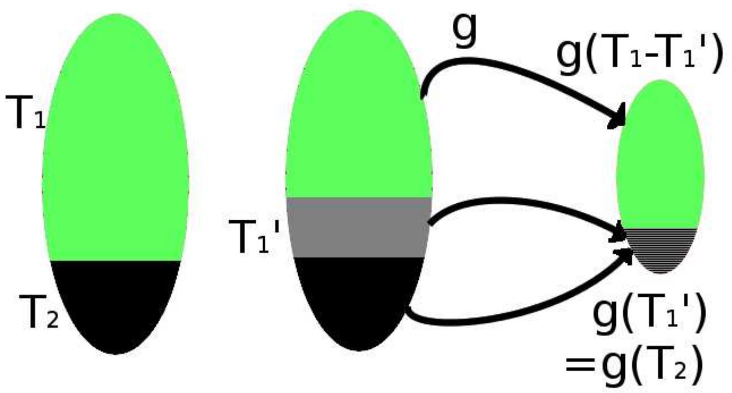


Figure 4.3: T is divided into T_1 and T_2, where each value in T_1 hashes to a different value in $g(T)$ and $g(T_1) = g(T)$. $T_1' \subseteq T_1$ is the subset of T, for which $g(T_1') = g(T_2)$, i.e. that contains all the elements in T_1 that hash into the same values as the elements in T_2.

but $|g(T_1)| = |T_1|$, and thus, the offsets $z_i^{(1)}$ and $z_i^{(2)}$ can get chosen for all $i \in g(T_1 - T_1')$ independently of each other, but also independently of the (now already chosen) z-values for $i \in g(T_1')$. This implies that the hash values $h_1(x)$ and $h_2(x)$ for $x \in T_1 - T_1'$ are distributed independently and uniformly, as well as independently from the values for $x \in T_1' \cup T_2$.

□

We have now established that the connected components in $G(S, h_1, h_2)$ are fully random if we choose (h_1, h_2) from $R(S)$. Now we want to set an upper bound on the number of cycle and leaf edges in a graph $K(T)$, such that we can use $N(k, l, q)$ to count the possible number of graphs $K(T)$ with edges $T \in S$ and compute the probability for subgraph to have a cyclomatic number ≥ 2.
Recall the definition of a *special edge* in K (Definition 4.3.2).

Lemma 4.5.5. *If $G = G(S, h_1, h_2)$ is l-bad, then there exists a subset $T \subseteq S$ such that $|g(T) = |T| - l$, $K(T)$ is connected and the total number of leaf and cycle edges in $K(T)$ is at most $2l$.*

Proof. G is l-bad and thus, we can find some set $T \subseteq S$ such that $K(T)$ is connected and $|g(T)| < |T| - l$. We start deleting keys from T and the corresponding edges from $K(T)$, while the new set $T' \subseteq T$ does not break the following two conditions: $K(T')$ still connected and $|g(T')| \leq |T'| - l$.

When this process stops, all cycle and leaf edges in $K(T')$ are special edges in $K(T)$, because removing non-special edges will always reduce both $|g(T')|$ and $|T'|$ and thus be possible, unless we are about to remove an inner non-cycle edge, which would split the graph into two. Removing a special edge however will reduce the size of T' by one while $|g(T')|$ stays the same and therefore, removing special edges will only be possible until

$$|g(T')| = |T'| - l \ .$$

When this status quo is reached, no more special cycle or leaf edges can get removed. If $|g(T')| = |T'| - l$, then we can split T' into T'' and $T' - T''$, such that the elements in $T' - T''$ hash to $|T'| - l$ different values and all l elements in T'' hash into values already covered by $g(T' - T'')$. The edges corresponding to the keys in T'' and the keys in T' that are hashed into $g(T'')$ must be the ones that are special, and there are at most $|T''| + l$ of them.

$$|T''| \leq l \Rightarrow \ \#\text{special edges } \leq 2l \ .$$

Since all leafs and cycle edges in T' are special, there are at most $2l$ cycle and leaf edges in $K(T')$ and $|g(T')| = |T'| - l$. $\square$

4.6 Proof of the Main Theorem

We are now able to prove the Main Theorem.

Theorem 4.4.1. *Let $\varepsilon > 0$ and $l \geq 1$ be fixed and let $m = m(n) = (1 + \varepsilon)n$. For any set $S \subseteq U$, $|S| = n$, there exists a set $R(S)$ of 'good' hash function pairs in $\hat{\mathcal{R}}^{2l}_{r,m}$, such that for randomly chosen $(h_1, h_2) \in \hat{\mathcal{R}}^{2l}_{r,m}$ the following holds.*

(a) $\boldsymbol{Prob}[(h_1, h_2) \in R(S)] \geq 1 - \frac{n}{r^l}$.

(b) For every constant $q \geq 2$ the probability that $(h_1, h_2) \in R(S)$ and that there is a connected subgraph of $G(S, h_1, h_2)$ whose cyclomatic number is at least q, is $O(n^{1-q})$.

(c) For every key $x \in S$, the probability that $(h_1, h_2) \in R(S)$ and that in the graph obtained from $G(S, h_1, h_2)$ by removing the edge $(h_1(x), h_2(x))$ the vertex $h_1(x)$ is the endpoint of a simple path of length t, is bounded by $(1 + \varepsilon)^{-t}$.

Proof. (a) Let each edge $(h_1(x), h_2(x))$ in $G = G(S, h_1, h_2)$ be labeled with x. Recall that for (h_1, h_2) to be in $R(S)$, G may not be l-bad. We are going to find an upper bound on the probability that G is l-bad, to compute a lower bound on the probability that $G(S, h_1, h_2) \in R(S)$.

But if G *was* l-bad, then from lemma 4.5.5 we know that there exists some subset $T \subseteq S$ such that $|g(T)| = |T| - l$ and $K(T)$ contains at most $2l$ leaf and cycle edges. Let k be the number of elements in T.

Furthermore, let H be a connected bipartite graph with k edges, each marked with a different element from T, and let H have at most $2l$ leaf and cycle edges. We define the following two events:

L(K(T)) $|g(T)| = |T| - l$

$\mathbf{I_{K(T)}(H)}$ $K(T)$ is isomorphic to H

We want to bound the probability that both events occur, i.e.

$$\mathbf{Prob}[I_{K(T)}(H) \wedge L(K(T))]$$

If $L(K(T)$ occurs, then T can be partitioned into to two disjoint subsets T_1 and T_2, $|T_1| = |T| - l$, such that every element in T_1 is hashed into a different element by g,

$$|T_1| = |g(T_1)| = |T| - l$$

and all elements in T_2 are hashed into values already covers by T_1,

$$g(T_2) \subseteq g(T_1).$$

(Recall Figure 4.3.)

Let $T_1' \subseteq T_1$ be the subset of T_1 that is mapped into the same values as T_2,

$$g(T_1') = g(T_2).$$

Because all elements in T_1 hash into different values, the size of T_1' must be bounded by the size of T_2.

$$|T_1'| = |g(T_1') = |g(T_2)| \leq |T_2| = l$$

Recall that T contains k elements, and thus, there are $\leq k^{2l}$ possibilities to choose T_2 and T_1' from T. We want to bound the probability that $g(T_1') = g(T_2)$ and $|g(T_1')| = |T_1'|$.

$$\begin{aligned} &\mathbf{Prob}[g(T_1') = g(T_2) \wedge |g(T_1')| = |T_1'|] \\ = \ &\mathbf{Prob}[g(T_1') = g(T_2)] \cdot |\{T_1' \mid |g(T_1')| = |T_1'|\}| \end{aligned}$$

Because g is chosen randomly from the $2l$-wise independent hash family $\mathcal{H}_r^{2l}$, the probability that $g(T_1') = g(T_2)$ for a given T_2 equals

$$\mathbf{Prob}[g(T_1') = g(T_2)] = \left(\frac{1}{r}\right)^{|T_2|}.$$

The number of possibilities to choose $|T_1'|$ elements such that $|T_1'| = |g(T_1')|$ is bounded by

$$|\{T_1' \mid |g(T_1')| = |T_1'|\}| \leq |T_1'|^{|T_2|}$$

and therefore, we get that

$$\begin{aligned}\mathbf{Prob}\left[g(T_1') = g(T_2) \wedge |g(T_1')| = |T_1'|\right] &\leq \left(\frac{|T_1'|}{r}\right)^{|T_2|} \\ &\leq \left(\frac{l}{r}\right)^{l}.\end{aligned}$$

Thus, we get

$$\mathbf{Prob}[L(K(T))] \leq k^{2l} \cdot \left(\frac{l}{r}\right)^{l} = \left(\frac{k^2 l}{r}\right)^{l}$$

Now we look at the cases, where this event occurs. Because $|g(T)| = |T| - l$, reuse the partition from above into T_1 and T_2 to show that $h_1(x)$ and $h_2(x)$ are independently and uniformly distributed. We start by choosing the offsets $z_i^{(1)}$ and $z_i^{(2)}$ randomly for all $i \in g(T_2)$ and assume that they are fixed. Because $f_1(x)$ and $f_2(x)$ are $2l$-wise independent,

$$h_1(x) = \left(f_1(x) + z_{g(x)}^{(1)}\right) \pmod{m} \text{ and } h_2(x) = \left(f_2(x) + z_{g(x)}^{(2)}\right) \pmod{m}$$

are still distributed independently in $[m]$ for $x \in T_2 \cup T_1'$. This leaves us with $g(T_1 - T_1')$. Because of our definition of T_1'

$$g(T_1 - T_1') \cap g(T_1' \cup T_2) = \emptyset$$

but $|g(T_1)| = |T_1|$, and thus, the offsets $z_i^{(1)}$ and $z_i^{(2)}$ can get chosen for all $i \in g(T_1 - T_1')$ independently of each other, but also independently of the (now already chosen) z-values for $i \in g(T_1')$. This implies that the hash values $h_1(x)$ and $h_2(x)$ for $x \in T_1 - T_1'$ are distributed independently and uniformly, as well as independently from the values for $x \in T_1' \cup T_2$.

Now we can apply Lemma 4.5.5 and get the probability that $K(T)$ is isomorphic to H is bounded as follows:

$$\mathbf{Prob}[I_{K(T)}(H)] \leq 2m^{-|E_H|-q-1},$$

q being the cyclomatic number of H. We can now compute the overall probability that $|g(T)| = |T| - l$ *and* $K(T)$ is isomorphic to H, recalling that the number of edges in H is k and the cyclomatic number of H is 0.

$$\begin{aligned}\mathbf{Prob}[I_{K(T)}(H) \wedge L(K(T))] &= \mathbf{Prob}[L(K(T))] \cdot \mathbf{Prob}[I_{K(T)}(H)] \\ &\leq \left(\frac{k^2 l}{r}\right)^l \cdot 2m^{-|E_H|-q-1} \\ &= \left(\frac{k^2 l}{r}\right)^l \cdot 2m^{-k-1}\end{aligned}$$

Now we need the number of non-isomorphic graphs on k edges that have at most $2l$ cycle and leaf edges, which is the number of possibilities to choose H. Following Lemma 3.2.6 we know that this number is $N^*(k, 2l) = k^{O(2l)} = k^{O(l)}$. Because l is a constant, the number of of possibilities to choose H is $k^{O(1)}$. Also, because S contains n elements and T contains k elements, there are less than n^k possible ways to choose T and mark the edges in H uniquely with the k keys in T.

If we sum over all k, we obtain an upper bound on the probability that $G(S, h_1, h_2)$ is l-bad. Recall that l is a constant, and that $m/n = (1+\varepsilon)$.

$$\begin{aligned}\mathbf{Prob}[\exists H : I_{K(T)}(H) \wedge L(K(T))] &\leq \sum_{k=2}^{n} k^{O(1)} n^k \left(\frac{k^2 l}{r}\right)^l \cdot 2m^{-k+1} \\ &= \frac{1}{r^l} \sum_{k=2}^{n} k^{O(1)} \cdot 2(lk^2)^l \frac{n^k}{m^{k-1}} \\ &= \frac{1}{r^l} \sum_{k=2}^{n} k^{O(1)} \frac{n^k}{m^{k-1}} \\ &= \frac{n}{r^l} \sum_{k=2}^{n} k^{O(1)} \left(\frac{n}{m}\right)^{k-1} \\ &= \frac{n}{r^l} \sum_{k=2}^{n} \frac{k^{O(1)}}{(1+\varepsilon)^{k-1}} \\ &= O\left(\frac{n}{r^l}\right)\end{aligned}$$

Thus, we can bound the probability that $(h_1, h_2) \in R(S)$ by

$$\begin{aligned}\mathbf{Prob}[\nexists H : I_{K(T)}(H) \wedge L(K(T))] &= 1 - \mathbf{Prob}[\exists H : I_{K(T)}(H) \wedge L(K(T))] \\ &\geq 1 - \frac{n}{r^l}\end{aligned}$$

(b) Let $G = G(S, h_1, h_2)$ be a graph with cyclomatic number at least q.

G must contain some connected component with cyclomatic number $\geq q$. Recall that neither the deletion of cycle edges in a component, nor the deletion of leaf edges in a component will break the components connectivity. Take the connected component with cyclomatic number $q' \geq q$, and recursively delete $q' - q$ cycle edges until the cyclomatic number of the component equals q. Now recursively continue deleting leave edges, until there are no leaf edges left. Let $T \subseteq S$ be the set of elements corresponding to edges left in the graph, and call the connected graph with cyclomatic number q and no leaf edges H.

We will proof that for every constant $q \geq 2$, the probability that $(h_1, h_2) \in R(S)$ and that there exists a connected subgraph $K(T)$ of $G(S, h_1, h_2)$ with cyclomatic number at least q *and without any leafs* is $O(n^{1-q})$, which proofs the Theorem. If $(h_1, h_2) \in R(S)$ and $K(T)$ is contained in a connected component of G, then Theorem 4.5.4 (1) proves that $(h_1, h_2) \in R^*(T)$. Thus, Theorem 4.5.4 (2) proves that all edges in $K(T)$ are chosen completely at random.

Now Lemma 3.2.5 applies, and we can compute the number of possibilities to choose a connected bipartite graph on k edges without any leaves and cyclomatic number q, keeping in mind that q is a constant.

$$N(k, 0, q) = k^{O(0+q)} = k^{O(1)}$$

S contains n elements, and thus, there are less than n^k possibilities to choose the k elements in $T \subseteq S$, and label all edges in H with distinct elements in T.

From Lemma 4.3.3 we have that the probability that $K(T)$ is isomorphic to H is at most

$$\frac{2}{m^{|E_H|+q-1}}$$

and thus, the probability that there exists a $T \subseteq S$, where $|T| = k$ and $K(T)$ has no leaves and cyclomatic number q is bounded by

$$k^{O(1)} \cdot n^k \cdot \frac{2}{m^{k+q-1}}$$

Summing over all k we get the probability that there exists such a T of arbitrary size and thus, the probability that $(h_1, h_2) \in R(S)$ and that there is a connected subgraph of $G(S, h_1, h_2)$ whose cyclomatic number is at least q. Recall that $m/n = 1 + \varepsilon$, and that k cannot be bigger than n. The probability that $(h_1, h_2) \in R(S)$ and that there exists a connected subgraph of $G(S, h_1, h_2)$ whose cyclomatic number is at least q is therefore bounded by

$$\begin{aligned}\sum_{k=q+1}^{n} k^{O(1)} \cdot n^k \cdot \frac{2}{m^{k+q-1}} &= \sum_{k=q+1}^{n} 2k^{O(1)} \cdot \frac{1}{n^{q-1}} \cdot \frac{n^{k+q-1}}{m^{k+q-1}} \\ &= \frac{1}{n^{q-1}} \cdot \sum_{k=q+1}^{n} k^{O(1)} \cdot \frac{1}{(1+\varepsilon)^{k+q-1}} \\ &= \frac{1}{n^{q-1}} \cdot \sum_{k=q+1}^{n} \frac{k^{O(1)}}{(1+\varepsilon)^{k+q-1}} \\ &= O(n^{1-q})\end{aligned}$$

(c) Let $h_1(x)$ be the endpoint of a simple path of length t, that does not contain $e_x = (h_1(x), h_2(x))$. Let $x_1, x_2, \ldots, x_t$ be the elements on the path, such that $h_1(x) = h_1(x_t)$, $h_2(x_t) = h_2(x_{t-1})$, $h_1(x_{t-1}) = h_1(x_{t-2})$, and so on. Let e_i denote edge $(h_1(x_i), h_2(x_i))$

Let $T = \{x, x_1, x_2, \ldots, x_t\}$, and notice, that the path

$$p = (h_1(x) \rightarrow) e_{x_1} \rightarrow e_{x_2} \rightarrow \ldots \rightarrow e_{x_t}$$

only exists, if $K(T)$ is a connected graph. Thus, we can bound the probability that $(h_1, h_2) \in R(S)$ and $h_1(x)$ is an endpoint of a path of length t by the probability that p exists if $(h_1, h_2) \in R(S)$ and $K(T)$ is connected. According to Theorem 4.5.4 (1), if $(h_1, h_2) \in R(S)$ and $K(T)$ is connected then $(h_1, h_2) \in R^*(T)$. Theorem 4.5.4 (2) then proves, that the values $h_1(x)$, $h_2(x)$, $h_1(x_i)$ and $h_2(x_i)$ for $i = 1, 2, \ldots t$ are all completely random and thus, we can to bound the probability that the path p is formed in the completely random bipartite graph $K(T)$ by $\frac{1}{m^t}$.

There are less than n^t possibilities to choose t edges from S, $|S| = n$. The probability that $(h_1, h_2) \in R(S)$ and $h_1(x)$ is an endpoint of a path of length t is therefore bounded by

$$\frac{1}{m^t} \cdot n^t = \left(\frac{n}{m}\right)^t = (1+\varepsilon)^t$$

□

Using the Main Theorem 4.4.1, we have established some important properties of hash function pairs from the class $\hat{R}^d_{r,m}$, that we now can use to show there effectiveness on Cuckoo Hashing.

Chapter 5

Cuckoo Hashing with $\mathcal{R}^d_{r,m}$ instead of Siegel's functions

Recall, that we want to use hash functions from the hash class $\mathcal{R}^d_{r,m}$ examined in Chapter 4 to build a fast and space-saving dictionary. We will exchange the usage of Siegel's hash functions in Cuckoo Hashing with function pairs chosen from the hash family $\mathcal{R}^d_{r,m}$, and see how this affects the performance. We choose function pairs from the hash family $\mathcal{R}^d_{r,m}$, and let $m = (1+\varepsilon)n$, $r = m^\delta$, $\frac{1}{2} < \delta < 1$ and $d = 4$. The bound L for insertions is chosen to be $\Theta(\log n)$. Clearly, both the *deletion* and *lookup* operations are still running in constant time. Insertions are somewhat more complicated and thus, we will analyse the procedure described in Section 2.3.4, using the new hash function pairs. We will look at a phase consisting of ρn operations on n keys.

5.1 Insertion

We analyse a phase of ρn, $\rho > 0$, updates involving n keys, and want to show, that the expected time spend on these ρn operations is $O(n)$. We let a *subphase* describe the lifespan of a hash function pair, i.e. the time from choosing a hash pair until we either

- are done with all operations or
- meet a series of L nestless keys during an insertion and thus, choose a new hash function pair.

Lemma 5.1.1. *In worst case, the total time spent on the insertions in one subphase is $O(n \log n)$.*

Proof. We chose $L = \log n$ and thus, each insertion in a subphase takes time $< \log n$. At most ρn elements get inserted during a subphase and therefore, the total time spent on the insertions in one subphase is

$$\leq \rho n \cdot \log n = O(n \log n)$$

$\square$

Lemma 5.1.2. *The expected number of subphases is* $1 + \frac{n}{r^2}$.

Proof. To find the expected number of subphases, we need to compute the probability that a subphase ends before all ρn elements are inserted.
Let (h_1, h_2) be the hash function pair now used. By Theorem 4.4.1 (a), the probability that (h_1, h_2) is not in $R(S)$ is

$$1 - \frac{n}{r^l} = 1 - \frac{n}{r^2}.$$

We can therefore expect (h_1, h_2) to be in $R(S)$.
If the subphase ends before all ρn elements are inserted, then there must be some key x that causes a series of L nestless keys. This can only happen if either

(1) x belongs into a subgraph with cyclomatic number $q \geq 2$, or

(2) x belongs into a subgraph with cyclomatic number $q \leq 1$, but there exists a path of length $\geq \frac{(L-1)}{3}$ that ends in $h_1(x)$ or $h_2(x)$ and does not contain edge $(h_1(x), h_2(x))$[1].

The probability that case (1) happens is bounded by

$$O(n^{1-q}) = O\left(\frac{n}{r^2}\right) ,$$

as stated in Theorem 4.4.1 (b).
The situation as described in case (2) is examined in Theorem 4.4.1 (c), that bounds the probability of this event for a path length t to

$$2 \cdot (1+\varepsilon)^{-t} = 2 \cdot (1+\varepsilon)^{-\frac{L-1}{3}}$$

[1]For the proof we refer to Lemma 4.1 in Pagh' and Rodlers article on Cuckoo Hashing [6].

If we set $L \geq 6\lceil \log_{1+\varepsilon} n \rceil + 1$, then the probability that this happens to some key x is bounded by

$$\begin{aligned}(1+\rho)n \cdot 2 \cdot (1+\varepsilon)^{-\frac{L-1}{3}} &\leq 2 \cdot (1+\varepsilon)^{-\frac{6\log_{1+\varepsilon} n}{3}} \\ &= (1+\rho)n \cdot 2 \cdot (1+\varepsilon)^{-2\log_{1+\varepsilon} n} \\ &= (1+\rho)n \cdot 2 \cdot (1+\varepsilon)^{-\log_{1+\varepsilon} n}(1+\varepsilon)^{-\log_{1+\varepsilon} n} \\ &= (1+\rho)n \cdot 2n^{-1}n^{-1} = (1+\rho)n \cdot 2n^{-2} \\ &= 2 \cdot (1+\rho)n^{-1} \cdot = O\left(\frac{n}{r^2}\right) .\end{aligned}$$

Now that we have shown that the probability that a subphase ends before all ρn elements are inserted is $O\left(\frac{n}{r^2}\right)$, we can compute the expected number of subphases to be

$$1 + O\left(\frac{n}{r^2}\right).$$

□

Theorem 5.1.3. *Let (h_1, h_2) be chosen at random from $\mathcal{R}^4_{r,m}$, where $m = (1+\varepsilon)n$ and $r = m^\delta$ for $\frac{1}{2} < \delta < 1$ a constant. Assume that we run a phase consisting of ρn operations on n keys. Then the lookup and deletion can be executed in worst case constant time, and insertion takes expected constant time.*

Proof. As argued before, the use of hash function pairs from $\mathcal{R}^4_{r,m}$ does not worsen the performance of lookups and deletions and thus, they take constant time in the worst case.
The expected time spent in all subphases that end before the phase has ended is bounded by the expected number of such subphases times the worst case time spent on all insertions during such a subphase.

$$\begin{aligned}O\left(\left(\frac{n}{r^2}\right) \cdot n \log n\right) &= O\left(\frac{n^2 \log n}{r^2}\right) \\ &= O\left(\frac{n^2 \log n}{(m^\delta)^2}\right) = O\left(\frac{n^2 \log n}{((1+\varepsilon)^\delta)^2}\right) \\ &= O\left(\frac{\log n}{n^\delta (1+\varepsilon)^{2\delta}}\right) = o(n) .\end{aligned}$$

Thus, for a total of ρn insertions, we expect to spend $o(n)$ time in total on these subphases and thus, amortised $O(1)$ time per element.
If the last subphase uses a hash function pair $(h_1, h_2) \notin R(S)$, the calculation becomes the same, as the probability of choosing such a hash function pair

is $\leq O\left(\frac{n}{r^2}\right)$ and again, the total time spent on all insertions during the last subphase is bounded by $O(n \log n)$.
If the last subphase uses a hash function pair $(h_1, h_2) \in R(S)$, we can argue just like Pagh and Rodler [6]. For each key x inserted during this last subphase, we know that if it creates a sequence of at least t nestless keys, then the graph $G(S, h_1, h_2)$ must contain a simple path of length $\geq \frac{t-1}{3}$, that starts in $h_1(x)$ or $h_2(x)$ and does not contain the edge $(h_1(x), h_2(x))$. Because $(h_1, h_2) \in R(S)$, Theorem 4.4.1 (c) proves that the probability that this happens is bounded by

$$2 \cdot (1+\varepsilon)^{-\frac{t-1}{3}} \ .$$

Thus, the expected number of nestless keys during the whole insertion of x is bounded by

$$\sum_{t \geq 1} 2 \cdot (1+\varepsilon)^{-\frac{t-1}{3}} = O(1) \ ,$$

and hence, the expected insertion time per element is $O(1)$ in this case.
Because we expect to spend constant time per element in total in all subphases except for the last, and we expect to spend constant time per element during the final subphase as well, the amortised expected insertion time is constant. $\square$

5.2 Space usage and efficiency

The efficiency of the hash function class $\hat{R}^d_{r,m}$ allows us to keep d a very small constant by still achieving the same bounds as Pagh and Rodler. As shown above, we can easily achieve the right bound using $d = 4$, which reduces the space used to save the function pair, but also reduces the number of multiplications to a small constant. This makes the hash function pairs chosen from this class very fast and achieves, as wanted, constant time bounds with a very small constant.

Chapter 6

Conclusion

The hash function pair class $\hat{R}^d_{r,m}$ introduced by Martin Dietzfelbinger and Philipp Woelfel has been proven to be useful for a dynamic dictionary, as it allows us to exchange Siegel's functions in Cuckoo Hashing, by keeping the constant time bounds. The construction is much simpler than Siegel's functions and also leads to a much smaller constant in the time bounds. The special structure of the hash function pairs chosen from $\hat{R}^d_{r,m}$ allow us to choose a very small d and still obtain very strong randomness in dense areas.

The idea of finding a different representation for hash functions to examine their behaviour has shown to be a still very unique, but also promising one to obtain better bounds on hash functions. Interestingly, the fact that two functions in a function pair share one g function has been shown to be very useful for the analysis, and does not affect the randomness negatively.

There are definitely possibilities for further work on this, either by applying $\hat{R}^d_{r,m}$ to other problems or by extending $\hat{R}^d_{r,m}$ to span over more than two functions, such that it can be used in other contexts.

Appendix A

A.1 List of symbols

Symbol	Explanation / Use
G	undirected bipartite graph
H	undirected connected graph
$\mathcal{H}$	Family of hash functions
$\mathcal{H}^{\lceil}_{\Updownarrow}$	d-wise independent Hash Family
V, W	Sets of vertices in a graph
E	Set of edges in a graph
η	Edge in a graph
S	Set of keys
S'	Modified set S
U	Universe
$\|U\|$	Size of universe U
a, b, e	Elements in a set
c	Connected component
d	Variable, integer ≥ 2
f	Function
h, h_1, h_2	Hash function
i	Index
j	Index
k, l, p, q	Variable, positive integer
$[m]$	Hash set consisting of integers $\{0, 1, 2, \ldots, m-1\}$
n	Size of set S
r	Size of one table used in Cuckoo Hashing
t_1, t_2	Hash table
x	Input to function f
$\log x$	$\log_2(x)$
$a \overset{uar}{\in} A$	Element a chosen uniformally at random from set A
$\text{poly}(n)$	$2^{O(\log n)} = n^{O(1)}$

Bibliography

[1] Alan Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications (extended abstract). In *IEEE Symposium on Foundations of Computer Science*, pages 20–25. IEEE Computer Society, 1989.

[2] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. Dynamic hashing in real time. In Johannes Buchmann, Harald Ganzinger, and Wolfgang J. Paul, editors, *Informatik*, volume 1 of *TEUBNER-TEXTE zur Informatik*, pages 95–119. Vieweg+Teubner Verlag, 1992.

[3] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM, J. Comput.*, 23(4):738–761, 1994.

[4] Richard M. Karp, Michael Luby, and Friedhelm Meyer auf der Heide. Efficient pram simulation on a distributed memory machine. *Algorithmica*, 16(4/5):517–542, 1996.

[5] Rasmus Pagh. Hash and displace: Efficient evaluation of minimal perfect hash functions. In Frank K. H. A. Dehne, Arvind Gupta, Jörg-Rüdiger Sack, and Roberto Tamassia, editors, *Workshop on Algorithms and Data Structures*, volume 1663 of *Lecture Notes in Computer Science*, pages 49–54. Springer, 1999.

[6] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In Friedhelm Meyer auf der Heide, editor, *Eupean Symposium on Algorithms*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2001.

[7] Martin Dietzfelbinger and Philipp Woelfel. Almost random graphs with simple hash functions. In *Proceedings of the thirty-fifth Annual ACM Symposium on Theory of Computing*, pages 629–638. Association for Computing Machinery, 2003.

[8] Martin Aumüller, Martin Dietzfelbinger, and Philipp Woelfel. Explicit and efficient hash families suffice for cuckoo hashing with a stash. In *Proceedings of the 20th Annual European Conference on Algorithms*, volume 7501 of *Lecture Notes in Computer Science*, pages 108–120, Berlin, Heidelberg, 2012. Springer-Verlag.

[9] Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics.* Springer, 2012.