Technische Universität Ilmenau

An Alternative Analysis of Cuckoo Hashing with a Stash and Realistic Hash Functions

Diplomarbeit

vorgelegt von:	Martin Aumüller
eingereicht am:	22.03.2010
Studiengang:	Informatik
Betreuer:	Univ.-Prof. Dr. Martin Dietzfelbinger
Fachgebiet:	Komplexitätstheorie und Effiziente Algorithmen Fakultät für Informatik und Automatisierung

Zusammenfassung

Diese Arbeit beschäftigt sich mit Cuckoo-Hashing, einem speziellen Hash-Verfahren. Ein Nachteil dieses Verfahrens ist, dass das Einfügen eines Elements mit einer kleinen Wahrscheinlichkeit fehlschlägt. Diese ist für einige Anwendungen groß genug, um Cuckoo-Hashing trotz seiner guten Laufzeiteigenschaften dort nicht praktikabel einsetzen zu können. Eine aktuelle Arbeit von Kirsch et al. schlägt eine Lösung für dieses Problem vor, indem ein kleiner zusätzlicher Speicher, der sogenannte Stash, eingeführt wird. In dieser Diplomarbeit geben wir einen neuen Beweis dafür, dass der Stash in Cuckoo-Hashing zu einer enorm reduzierten Fehlerwahrscheinlichkeit führt. Weiterhin werden wir zeigen, dass dieser keinen wesentlichen Einfluss auf die Laufzeiten der *insert*-, *delete*- und *lookup*-Operationen hat. Wir werden beweisen, dass unsere Analyse für eine Klasse von effizient auswertbaren Hashfunktionen gilt und somit eine offene Frage von Kirsch et al. beantworten. Für diese Klasse wird ein abstraktes Ergebnis vorgestellt, das es ermöglicht die Idee dieser Hashklasse auch in anderen Zusammenhängen zu nutzen. In Experimenten zeigen wir auf, dass schon ein kleiner zusätzlicher Speicher mit nur neun Speicherplätzen für Schlüssel eine enorme Verbesserung für die praktische Einsatzfähigkeit von Cuckoo-Hashing bringt. Wir können damit problemlos in der Nähe der theoretisch maximalen Tabellenauslastung von Cuckoo-Hashing arbeiten, ohne Gefahr zu laufen, dass die Datenstruktur durch eine fehlschlagende Einfügeoperation neu aufgebaut werden muss. Insbesondere im Falle kleiner Tabellen wird sich der Stash als wertvolle Erweiterung erweisen. Weiterhin werden wir auch die zwei populären Varianten d-äres Cuckoo-Hashing und geblocktes Cuckoo-Hashing näher betrachten. Dabei wird sich herausstellen, dass in diesen Varianten ein zusätzlicher Speicher die Fehlerwahrscheinlichkeit bei kleinen Tabellengrößen deutlich senkt. Im Gegensatz zu normalem Cuckoo-Hashing ist ein Stash jedoch im Falle großer Tabellen nutzlos.

Abstract

We consider the hashing scheme cuckoo hashing which provides worst-case constant lookup time. On the downside, insertions fail with a small probability, which makes cuckoo hashing unsuitable for some applications, because such failing insertions result in a reconstruction of the whole data structure. A recent work by Kirsch et al. proposes a solution for this problem by introducing a small constant-sized memory called stash. We give a new proof for the dramatically reduced failure probability of cuckoo hashing when using such a stash. Furthermore, we show that it has no influence on the general performance of *insert*, *lookup* and *delete* operations. We show that a certain class of hash functions that can be efficiently evaluated will work in the same way as fully random hash functions do in cuckoo hashing with a stash. We also present a general framework for this class of hash functions, which allows us to reuse our results for other problems. We experimentally show that a stash size of only nine allows us to operate close to the theoretical thresholds of cuckoo hashing without having to reconstruct the whole data structure. Especially in the case of small tables, a stash will prove itself to be very useful. Furthermore, we evaluate d-ary cuckoo hashing and blocked cuckoo hashing enhanced with the stash approach in practical scenarios. A stash will help these data structures in the case of small table sizes. When tables grow bigger, a stash becomes unnecessary in these variants of cuckoo hashing.

Contents

1. Introduction and Motivation

Hash tables are crucial for the performance of many real world applications. They provide a data structure to store elements, sometimes satellite data associated with them, in a space-efficient way and provide fast access to these items. Moreover, in many hash table approaches elements can be inserted into and deleted from a hash table dynamically.

Over the last decades, many different approaches evolved in this research area, starting with some of the most popular variants like linear probing and chained hashing. For performance-crucial applications like packet routing on the internet, early approaches do not fulfill the demands. If a packet arrives at a router, the time slot to decide on which interface this packet should leave is sometimes extremely short and spending tremendous time on this decision yields undesired congestion. In some applications, packets have to be analyzed in these short time frames. Thus, provable worst-case running time of a specific operation, especially the lookup operation, is important to such applications.

In 2001, Pagh and Rodler [PR01] (full version [PR04]) introduced a new dynamic approach called cuckoo hashing. It provides worst-case constant running time of lookups and deletions. Furthermore, insertions are also expected to run in amortized constant time. In contrast to previously known hashing schemes, it is space-efficient and at most two independent table accesses are required for lookups and deletions. They need $\Theta(\log n)$-wise independent hash functions for conducting their analysis.

On the downside, there is a small chance that the insertion of a key fails and the hash table must be rebuilt. Although cuckoo hashing has a good performance, such failures happen at a too high rate for some applications and have a great impact once they occur. Thus, in some scenarios, e.g., the router scenario mentioned above, applying cuckoo hashing seems impossible due to time constraints.

Recently, an article by Kirsch, Mitzenmacher and Wieder [KMW08] provided an approach to handle such failures in a simple way. They introduce a small constant-sized memory, the so-called stash, and show that the failure probability of cuckoo hashing is dramatically reduced in this way. The main motivation for such a solution is that small and fast memory is available in hardware, e.g., content-addressable memory (CAM) is both, cheap and extremely fast in such a scenario, for it can look up keys by querying memory cells in parallel.

The analysis in [KMW08] is conducted in a technically involved way. Moreover, for their analysis to succeed they assume fully random hash functions to be available for free. While it is possible to construct such hash functions, they require large space and are thus cumbersome in practice.

In [DW03], Dietzfelbinger and Woelfel found a class of realistic hash functions that is applicable to the analysis of cuckoo hashing. Interestingly, their approach seems to be adaptable to the refinement proposed by Kirsch, Mitzenmacher and Wieder in two ways.

Apparently, we might avoid their full randomness assumption using the class of hash functions proposed in [DW03]. More astonishingly, we might use parts of this work to prove the dramatically decreased failure probability of cuckoo hashing using a stash in a simpler way than in [KMW08], even in the fully random case. Both of these conjectures are the foundation of this thesis.

We will start by introducing all topics that are necessary to understand the subsequent chapters of this thesis in Chapter 2. It will cover two basic mathematical results, which are of tremendous importance to us. Furthermore, we will have a look at some basic graph and probability theory, followed by a brief introduction to hashing.

Chapter 3 will introduce cuckoo hashing and stash-based cuckoo hashing. We will describe the most important algorithms, results on the failure probability and performance of the different operations, and state main differences between the standard approach and the stash variant.

In Chapter 4, we will present an alternative analysis of stash-based cuckoo hashing using fully random hash functions. In this chapter, we will prove the main result of [KMW08] in a less involved way. At the end of this chapter, we will compare the performance of the lookup, delete and insert operation to traditional cuckoo hashing.

Subsequently, we will consider a class of realistic hash functions in Chapter 5. The term "realistic" means in this case that these hash functions can be evaluated efficiently in constant time and are more space-efficient than fully random hash functions. We will see that parts of the analysis conducted in the previous chapter are also applicable if we use these hash functions, but observe that we are missing one necessary fact to conduct the whole analysis. But all is not lost, as we will be able to reuse some of our observations to provide a generalization of the ideas in this chapter and the approach used in [DW03].

Chapter 6 will describe that abstraction and give some examples, most importantly filling the gap of the previous analysis. This will eventually finish our theoretical analysis of stash-based cuckoo hashing.

Towards the end of this thesis, we will report on some experiments that emphasize the practical usefulness of a stash in cuckoo hashing. While the theoretical properties of cuckoo hashing provide good results for high-performance applications, there are two popular refinements that are especially focused on the space-efficiency. We will observe experimentally if the addition of a stash is practically sound there as well.

2. Preliminaries

This chapter will introduce notions and concepts used in this thesis. The first section will provide two important mathematical results that are frequently used. Section 2.2 introduces the most important graph theoretical concepts. It will introduce the notion of a graph and finish with some results about the *cyclomatic number* of a graph and *graph isomorphism*. Section 2.3 gives a brief introduction to some well-known results in basic probability theory, which we will often use. In Section 2.4, we introduce the notion of hash functions and hash classes. Subsequently, we will consider *universal hash functions*.

2.1. Basic Mathematics

We consider the infinite geometric series

$$\sum_{k=0}^{\infty} ar^k.$$

This series converges absolutely if and only if $|r| < 1$. We will often consider the following infinite series

$$\sum_{k=0}^{\infty} k^c r^k,$$

where c is a nonnegative constant and $0 < r < 1$. By applying the ratio test, it can easily be seen that this series converges absolutely as well,

$$\begin{aligned}\lim_{k\to\infty}\left|\frac{(k+1)^c}{k^c}\cdot\frac{r^{k+1}}{r^k}\right| &= \lim_{k\to\infty}\frac{(k+1)^c}{k^c}\cdot r\\ &= r\\ &< 1.\end{aligned}$$

It is a well-known result that the binomial coefficient

$$\binom{n}{k} = \frac{n!}{(n-k)!\cdot k!}$$

is the number of k-element subsets of an n-element set. It follows immediately that

$$\binom{n}{k} = \frac{n\cdot(n-1)\cdot\ldots\cdot(n-k+1)}{k!}$$

and thus

$$\binom{n}{k} \leq \frac{n^k}{k!}.$$

We will often choose k-element subsets of an n-element set and use the upper bound n^k to simplify our calculation. We will only use a more accurate value if it is either necessary to obtain our results or to simplify future expressions. Furthermore, we define $[n] = \{0, 1, \ldots, n-1\}$.

2.2. Basic Notation of Graph Theory

This section introduces our notation of graph theory. The references were [BM08] and [Die05].

A graph G is an ordered pair $(V(G), E(G))$ consisting of a set $V(G)$ of *vertices* and a set $E(G)$, disjoint from $V(G)$, of *edges*, where $E(G) \subseteq \{\{u, v\} \mid u, v \in V(G)\}$. A vertex v is incident with an edge e if $v \in e$, and e is incident with v. The two vertices incident with an edge are its ends, and an edge *joins* its ends. An edge $\{u, v\}$ is usually written as uv. Two vertices u, v of G are adjacent, if uv is an edge of G. We call two distinct adjacent vertices *neighbors*. The set of neighbors of a vertex v in a graph G is denoted by $N_G(v)$. Two distinct edges are adjacent if they have one end in common.

Throughout this thesis, the letter G denotes a graph and when there is no scope for ambiguity, we omit the letter G and write V and E instead of $V(G)$ and $E(G)$. In such cases, we also use $|V| = n$ and $|E| = m$.

A graph is called *simple*, if it has no loops and parallel edges. Throughout this thesis, we will often focus on multigraphs. In a multigraph, the edge set E is a multiset and thus parallel edges are allowed[1]. A *complete graph* is a graph where any two vertices are adjacent to each other. A graph is *bipartite* if its vertex set can be partitioned into two subsets L and R in such a way that every edge has one end in L and one end in R. We call such a partition (L, R) a *bipartition*. If G is a bipartite graph and every vertex in L is joined with every vertex in R then G is called the *complete bipartite graph*. Figure 1 shows a complete graph and a complete bipartite graph.

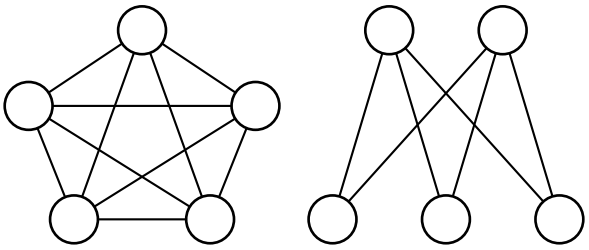
A *simple path* is a non-empty graph $P = (V, E)$ of the form

$$V = \{v_0, v_1, \ldots, v_k\}, E = \{v_0v_1, v_1v_2, \ldots, v_{k-1}v_k\},$$

where the v_i are all distinct. The number of edges of a path is its length. We often refer to a path by the natural sequence of its vertices and write $P = v_0v_1 \ldots v_k$. If $P = (V, E) = v_0 \ldots v_{k-1}$ is a path and $k \geq 3$, then the graph $C = (V, E \cup v_{k-1}v_0)$ is called a *cycle*. A graph is called *acyclic* if it contains no cycle, and *unicyclic* if it contains exactly one cycle. The length of a cycle is the number of its edges.

For graphs $G = (V, E)$ and $G' = (V', E')$, let $G \cup G' = (V \cup V', E \cup E')$ and $G \cap G' = (V \cap V', E \cap E')$. If $G \cap G' = (\emptyset, \emptyset)$, G and G' are said to be *disjoint*. If $V' \subseteq V$ and $E' \subseteq E$, $G' = (V', E')$ is a subgraph of G. We denote this with $G' \subseteq G$.

[1] When there is no scope for ambiguity in later sections, we mean multigraph when we say graph.



(a) Complete graph with five vertices.

(b) Complete bipartite graph with five vertices.

Figure 1: Examples for complete graphs.

A graph is said to be *connected* if for every partition of its vertices into two nonempty sets X and Y, there is an edge with one end in X and one in Y. Otherwise the graph is *disconnected*. A maximal connected subgraph of G is called a *component* of G and $\mathfrak{C}(G)$ is the set of components of G. Although components are connected by definition, we will often use the term *connected component*. A subgraph G' is said to be *induced*, if it contains all edges $uv \in E$ for $u, v \in V'$. We denote the induced subgraph G' with $G[V']$. On the other hand, a subgraph is a *spanning subgraph* if $V' = V$, hence G' is obtained from G by edge deletions only.

The *degree* of a vertex v in G, denoted by $d_G(v)$, is the number of edges of G incident with v. If G is a simple graph, $d_G(v)$ is the number of neighbors of v in G. A vertex with degree zero is called an *isolated vertex*; a vertex with degree one is called a *leaf vertex* and the edge incident to this vertex is called a *leaf edge*. Every non-leaf edge is called an *inner edge*.

A graph is called an *even graph* if every vertex in the graph has even degree, and *odd graph* vice versa. The following theorem contains fundamental information of the relation between vertices and edges in a graph.

Lemma 2.1 ***(Handshaking Lemma)***
For any graph G,

$$\sum_{v \in V} d_G(v) = 2m.$$

Proof. If we sum up the degrees of the vertices in G, every edge uv is counted twice, once for $d_G(v)$ and once for $d_G(u)$, hence the sum over all degrees must equal $2 \cdot |E| = 2m$. □

We call an acyclic graph G a *forest*. A *tree* is a connected forest. The components of a forest are hence trees. We prove two simple theorems about trees.

Lemma 2.2
Let T be a tree. Then

$$m = n - 1.$$

Proof. If v is a leaf of a tree T, the subgraph $T' = T[V - \{v\}]$ is a tree and $|E'| = |E| - 1$. Because the tree $T = (\{v_0\}, \emptyset)$ has no edges, the lemma holds by induction on the number of vertices in T. □

Lemma 2.3
Let T be a tree. Then any two vertices u and v of T are linked by a unique path.

Proof. There must be at least one path between u and v, because T is connected. Assume for a contradiction two paths P_1, P_2 between u and v. Clearly, both start in u and might go along vertices $u_1, u_2, \ldots, u_i$ until they separate. On the other hand, they both end in v and might join in a vertex $v_j, v_{j-1}, \ldots, v_1, v$. If $P_1 \neq P_2$ then the two edge sets between u_i, v_j in P_1 and P_2 form a cycle. T is acyclic, thus $P_1 = P_2$. □

A subtree of a graph is a subgraph which is a tree. If this tree is a spanning subgraph, it is called a *spanning tree* of the graph. For given spanning subgraphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ of a graph $G = (V, E)$, we can form the spanning subgraph of G whose edge set is the symmetric difference of E_1 and E_2 (all edges that are exclusively in either E_1 or E_2). This graph is called the *symmetric difference* of G_1 and G_2, denoted by $G_1 \Delta G_2$. See Figure 2 for an example.

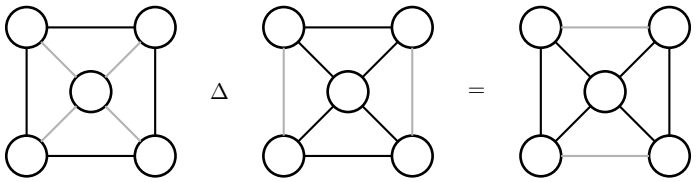
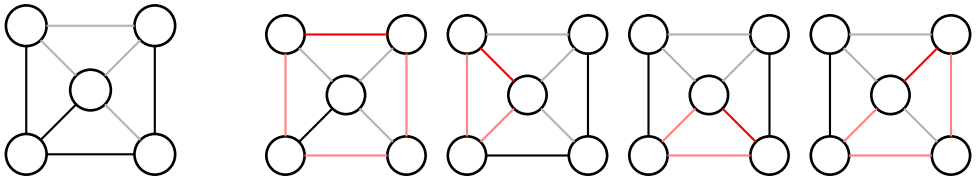


Figure 2: Symmetric difference of two subgraphs (black lines) of W_4. An edge exists in the resulting subgraph, if it is present in exactly one of the subgraphs on the left side.

Let $G = (V, E)$ denote a connected graph and $T = (V, E')$ a spanning tree of G. The complement $(V, E - E')$ of a spanning tree T is called its *cotree* and is denoted by $\overline{T}$. We know from Lemma 2.3 that for every edge $e = vw$ of a cotree $\overline{T}$ there is a unique vw-path in T connecting its ends, namely vTw. Thus $T' = (V, E' \cup e)$ contains a unique cycle. This cycle is called the *fundamental cycle* of G with respect to T and e. We denote this subgraph by $C_{T,e}$. See Figure 3 for an example.

Without a proof, we will state the following important theorem.



(a) A graph with a spanning tree T (black lines).

(b) 4 fundamental cycles with respect to T. Light red lines denote edges of the spanning tree used in the fundamental cycle. The strong red line marks the edge of the cotree forming the fundamental cycle.

Figure 3: A graph with a spanning tree T and the fundamental cycles with respect to T.

Theorem 2.4
Let T be a spanning tree of a connected graph G. Every even subgraph of G can be expressed uniquely as a symmetric difference of fundamental cycles with respect to T.

The *edge space* $\mathcal{E}(G)$ is the vector space over the 2-element field $\mathbb{F}_2 = \{0, 1\}$ of all functions $E \to \mathbb{F}_2$. Every element of $\mathcal{E}(G)$ corresponds to a subset of E, the set of those edges to which it assigns a 1, and every subset of E is uniquely represented in $\mathcal{E}(G)$ by its indicator function. The set of all even subgraphs of a graph G forms a subspace $\mathcal{C}(G)$ of the edge space of G. We call this subspace the *cycle space* of G, because it is generated by the fundamental cycles of a spanning tree of G (see [Die05] for the details). We call the dimension of the cycle space the *cyclomatic number* $\gamma(G)$.

Lemma 2.5
Let G be a connected graph, then

$$\gamma(G) = m - n + 1.$$

Proof. By definition $\gamma(G)$ is the dimension of the cycle space $\mathcal{C}(G)$. The number of fundamental cycles is exactly the number of edges in the cotree $\overline{T}$, because every edge $e \in \overline{T}$ is a member of a fundamental cycle $C_{T,e}$ exactly once. Furthermore, T is a spanning tree and has $n - 1$ edges (Lemma 2.2). Hence,

$$\gamma(G) = m - n + 1.$$

□

It is interesting to note that this yields a simple algorithm to obtain an acyclic connected graph. By repeatedly removing cycle edges in a connected graph G, we obtain an acyclic graph after $\gamma(G)$ steps, because every removed cycle edge destroys a unique fundamental cycle (with respect to the spanning tree we obtain after $\gamma(G)$

steps). Note that it follows that a unicyclic component has the same number of nodes as edges.

By using Lemma 2.5 we can state the following corollary by summing over all connected components of a graph.

Corollary 2.6
Let G be an arbitrary graph with c connected components. Then

$$\gamma(G) = m - n + c.$$

We are often interested to obtain graphs that have tree or unicyclic components. We define the *excess* of a graph G as follows.

Definition 2.7
The excess $\mathrm{ex}(G)$ is the minimal number of edges we have to remove from G such that all connected components in G contain at most one cycle.

Clearly, the excess depends on the cyclomatic number of a graph.

Lemma 2.8
Let G be a connected graph. Then

$$\mathrm{ex}(G) = \begin{cases} 0 & \text{if } G \text{ is acyclic} \\ \gamma(G) - 1 & \text{otherwise.} \end{cases}$$

Proof. If G is acyclic we are done, so assume that G is cyclic. To obtain an acyclic subgraph, one has to remove at least $\gamma(G)$ edges. We only require a unicyclic component and thus have to remove $\gamma(G) - 1$ edges. □

We can extend this result in a natural way to an arbitrary graph by summing over its connected components.

Corollary 2.9
Let $G = (V, E)$. Then

$$\mathrm{ex}(G) = \sum_{C \in \mathfrak{C}(G)} \mathrm{ex}(C) \text{ and}$$
$$\mathrm{ex}(G) = \gamma(G) - \mathrm{cc}(G),$$

where $\mathrm{cc}(G)$ denotes number of cyclic connected components in G.

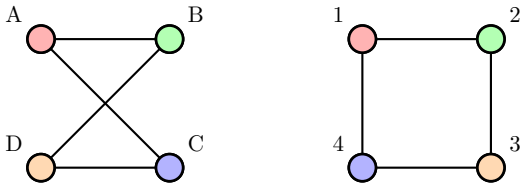
Proof. To obtain a graph whose components are trees or unicyclic, we have to remove edges in each component. By Lemma 2.8 follows that we have to remove $\mathrm{ex}(C)$ edges in a component $C \in \mathfrak{C}(G)$ that has more than one cycle. The first equation follows.

The second equation follows by inserting Lemma 2.8 into the first equation.

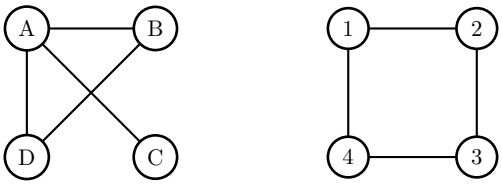
$$\begin{aligned}
\operatorname{ex}(G) &= \sum_{C \in \mathfrak{C}(G)} \operatorname{ex}(C) \\
&= \sum_{\substack{C \in \mathfrak{C}(G), \\ C \text{ acyclic}}} \operatorname{ex}(C) + \sum_{\substack{C \in \mathfrak{C}(G), \\ C \text{ cyclic}}} \operatorname{ex}(C) \\
&= \sum_{\substack{C \in \mathfrak{C}(G), \\ C \text{ acyclic}}} 0 + \sum_{\substack{C \in \mathfrak{C}(G), \\ C \text{ cyclic}}} \gamma(C) - 1 \\
&= \Big(\sum_{\substack{C \in \mathfrak{C}(G), \\ C \text{ acyclic}}} \gamma(C) + \sum_{\substack{C \in \mathfrak{C}(G), \\ C \text{ cyclic}}} \gamma(C) \Big) - \operatorname{cc}(G) \\
&= \sum_{C \in \mathfrak{C}(G)} \gamma(C) - \operatorname{cc}(G) \\
&= \gamma(G) - \operatorname{cc}(G).
\end{aligned}$$

□

Furthermore, we have to define *graph isomorphism*. We call graphs $H = (V_H, E_H)$ and $H' = (V_{H'}, E_{H'})$ isomorphic if there exists a bijection $\sigma : V_H \to V_{H'}$ such that $u, v \in V_H$ are adjacent in H if and only if $\sigma(u), \sigma(v) \in V_{H'}$ are adjacent in H'.



(a) Two isomorphic graphs with isomorphism $\sigma(A) = 1, \sigma(B) = 2, \sigma(C) = 4, \sigma(D) = 3$.

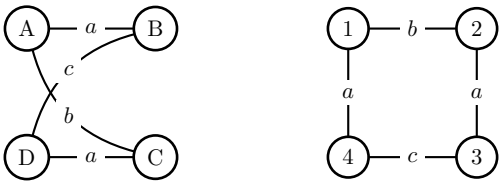


(b) Two non-isomorphic graphs, because there is no vertex with degree 3 in the right graph.

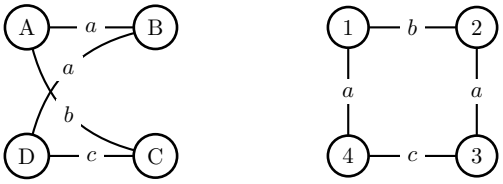
Figure 4: Example for a graph isomorphism and a non-isomorphism.

We are also interested in labeled graphs. In a labeled graph, each edge has a label from a set S of labels.

We call two graphs $H = (V_H, E_H)$ and $H' = (V_{H'}, E_{H'})$ with labeled edges isomorphic if there are bijections $\sigma : V_H \to V_{H'}$ and $\tau : E_H \to E_{H'}$ (possibly as multisets in the multigraph setting) such that if $e \in E_H$ connects $v, w \in V_H$, then $\tau(e)$ connects $\sigma(v), \sigma(w)$ and such that for all $e \in E_H$, e and $\tau(e)$ share the same label (Fig. 5).



(a) Labeled graph isomorphism with isomorphism $\sigma(A) = 1, \sigma(B) = 4, \sigma(C) = 2, \sigma(D) = 3$.



(b) Non-isomorphic labeled graphs, because there does not exist a vertex which is incident to both edges labeled a in the second graph.

Figure 5: Example for a labeled graph isomorphism and a non-isomorphism.

Our definition of labeled graph isomorphism equals the previous definition of graph isomorphism if all edges share the same label.

For more information about graphs we refer to [BM08] and [Die05].

2.3. Basic Notation of Probability Theory

This section will introduce our notation of probability theory and provide some important lemmas. If the reader is not familiar with the basics of probability theory, [MU05] gives an introduction to this field. The content of this section is mostly taken from this source.

We will often implicitly use an inequality known as the *union bound*.

Lemma 2.10
For any finite or countably infinite sequence of events $E_1, E_2, \ldots$,

$$\Pr\Big(\bigcup_{i\geq 1} E_i\Big) \leq \sum_{i\geq 1} \Pr(E_i).$$

We are also going to use the conditional expectation of a random variable.

Definition 2.11

$$\mathrm{E}(Y \mid Z = z) = \sum_y y \Pr(Y = y \mid Z = z),$$

where the summation is over all values in the range of Y.

When dealing with two random variables, we will often use the following lemma.

Lemma 2.12
For any random variables X and Y,

$$\mathrm{E}(X) = \sum_y \Pr(Y = y)\mathrm{E}(X \mid Y = y),$$

where the sum is over all values in the range of Y and $\mathrm{E}(X)$ exists.

Proof.

$$\begin{aligned}
\sum_y \Pr(Y = y)\mathrm{E}(X \mid Y = y) &= \sum_y \Pr(Y = y) \sum_x x \Pr(X = x \mid Y = y) \\
&= \sum_x \sum_y x \Pr(X = x \mid Y = y) \Pr(Y = y) \\
&= \sum_x \sum_y x \Pr(X = x \cap Y = y) \\
&= \sum_x x \Pr(X = x) \\
&= \mathrm{E}(X).
\end{aligned}$$

□

Furthermore, we are often interested in the expectation $\mathrm{E}(X)$ of a discrete random variable X. The following lemma simplifies our task in many situation.

Lemma 2.13
Let X be a discrete random variable that takes on only nonnegative integer values. Then

$$\mathrm{E}(X) = \sum_{i=1}^{\infty} \Pr(X \geq i).$$

Proof.

$$\begin{aligned}\sum_{i=1}^{\infty} \Pr(X \geq i) &= \sum_{i=1}^{\infty} \sum_{j=i}^{\infty} \Pr(X = j) \\ &= \sum_{j=1}^{\infty} \sum_{i=1}^{j} \Pr(X = j) \\ &= \sum_{j=1}^{\infty} j \Pr(X = j) \\ &= \mathrm{E}(X).\end{aligned}$$

The interchange of (possibly) infinite summations is justified, since the terms being summed are all nonnegative. □

Lemma 2.13 can be applied to conditional expectation as well and we will formulate this result as a corollary, because the proof is a straight-forward modification of the proof in said lemma.

Corollary 2.14
Let Y be a discrete random variable that takes on only nonnegative integer values. Then

$$\mathrm{E}(Y \mid Z = z) = \sum_{i=1}^{\infty} \Pr(Y \geq i \mid Z = z).$$

Let $(\Omega_n, \Pr_n)$ be a sequence of probability spaces and let E denote an event that is defined in each of them. If we say that E has "high probability", we mean that $\Pr_n(E) = 1 - \mathcal{O}(1/n^{\alpha})$ for some constant $\alpha \geq 1$.

2.4. Hashing and Hash Classes

The *dictionary* data structure is a fundamental concept in computer science. A dictionary is used for maintaining a set S under insertion and deletion of keys from a given universe U. One can access data associated with a given key (*satellite data*) by membership queries.

A lot of different ways are known for implementing a dictionary data structure, e.g., *hashing schemes*, *self-balancing binary search tress* or *skip lists*. Hashing schemes characterize themselves by using a *hash table* to store the elements of a set and using one or several *hash function(s)* to find the location of an element inside the hash table. Clearly, the performance of a hash scheme heavily depends on the used hash function. Formally, we can define hashing schemes as follows.

Given a *universe* U of keys and a hash table T of size $m \in \mathbb{N}$, a hash function $h : U \to [m]$ maps the keys from U to table positions in a range $[m]$. We want to store

a key x in table cell $T[h(x)]$. Note that there also exist hashing schemes using multiple hash functions and we will get to know one of them soon. The important question is: What should we do if two different keys collide? Formally, $h(x) = h(y)$ and $x \neq y$. We can see such a situation in Figure 6.

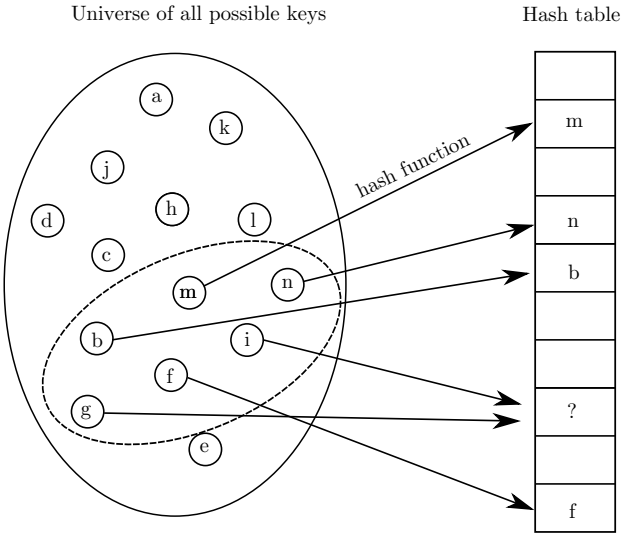


Figure 6: Basic hashing scheme. Keys are taken from a universe U of possible keys and mapped to cells in the hash table by a hash function.

Different solutions for this question led to a multitude of different hashing schemes over the past years. Clearly, the best solution would be a hash function that would completely avoid these collisions. Different ways are possible to generate such a function if S is known and we call such a hash function a *perfect hash function*. Two efficient variants for finding a perfect hash function for a static key set S are described in [BPZ07] and [CKRT04]. We often need a dynamic data structure, and thus require different solutions.

Two well-known approaches for dealing with collisions are *chained hashing* and *hashing with open addressing and linear probing*. In the former, collisions are resolved by using a linear list in each table cell. All keys with the same hash value are stored in this list. In the latter, we successively examine the m cells of a table until we discover an empty position. The first position for an element is of course the value of the hash function $h(x)$. If this place is already occupied, we inspect a new cell. The *linear probing* approach for instance visits the cells $h(x) + 1, h(x) + 2, \ldots$ next, and so on (addition modulo m) until an empty cell has been found.

A natural model for a hash function is that it is fully random, meaning it is a random element of $[m]^U = \{h \mid h : U \to [m]\}$; the set of functions with domain U and target $[m]$. Under this assumption the hash values $\{h(x) \mid x \in U\}$ are independent random variables uniformly distributed over $[m]$. Unfortunately, the space required to store

such a function is the same as that required to encode an arbitrary function in $[m]^U$ as a lookup table, making such an assumption cumbersome in practical scenarios.

Clearly, the choice of the hash function $h(x)$ is of tremendous importance. If we recall the first solution for collision handling (chained hashing) and our hash function would map every key to a constant cell $k \in [m]$, looking up an element could take up to n steps, because all elements are stored in a linked list. Even with more serious hash functions, a given hash function could always work well on a given key set, but fail on others. As the distribution according to which the keys are drawn is often unknown, this leads to the problem that there could be patterns in the input that yield a higher number of collisions. A solution for this problem was presented by Carter and Wegman in [CW77] with *universal hashing*. In this scheme the hash function h is sampled from a given well-designed universal hash function class $\mathcal{H}$, where sampling and evaluating a function from $\mathcal{H}$ should be possible in an efficient way.

Definition 2.15
A set of hash functions $\mathcal{H} \subseteq \{h \mid h : U \to [m]\}$ is *c-universal*, if for any $x, y \in U, x \neq y$:

$$Pr(h(x) = h(y)) \leq \frac{c}{m}$$

An important generalization of universal hashing is d-wise independent hashing.

Definition 2.16
A set of hash functions $\mathcal{H} \subseteq \{h \mid h : U \to [m]\}$ is *d-wise independent*, if for any distinct keys $x_1, \ldots, x_k$ and for all hash values $y_1, \ldots, y_k$:

$$Pr(h(x_i) = y_i, 1 \leq i \leq k) \leq \frac{1}{m^d}$$

Informally spoken, if $x_1, \ldots, x_d$ are distinct, their hash values are uniformly and independently distributed in $[m]$.

One well-known technique for constructing independent hash functions with a given degree of independence are polynomial hash functions. Let U be a finite field and $a_{d-1}, \ldots, a_0$ be a sequence of elements of U. Then define the members of the d-wise independent class by

$$h(x) = a_{d-1}x^{d-1} + a_{d-2}x^{d-2} + \ldots + a_1x + a_0 \mod p \mod m,$$

where multiplications and additions are carried out in U and p is a prime greater than m, see [DGMP92].

We will now consider a specific hashing scheme that will accompany us in the remaining chapters.

3. Cuckoo Hashing

This section will consider a high performance hashing scheme called cuckoo hashing. We will first introduce the algorithms for the insert, lookup and delete operations and will subsequently analyze their performance. At the end of this chapter, we will consider cuckoo hashing with a stash.

3.1. Standard Cuckoo Hashing

Cuckoo hashing was introduced by Pagh and Rodler in [PR01] (extended version in [PR04]). It is a *multiple choice* hashing scheme. In such schemes, each key can be placed into one out of d different places in the hash table for a fixed $d \geq 2$. It provides worst-case constant lookup time.

Cuckoo hashing is based on two tables T_1 and T_2 of size m each and makes use of two hash functions h_1 and h_2, mapping a key to a cell in the first resp. second table. We set m slightly larger than the size of the key set $|S| = n$, thus $m = (1+\varepsilon)n$, where $\varepsilon \in (0, 1)$. The maximal load factor of the data structure is hence $n/2m = 1/(2(1 + \varepsilon)) < 0.5$.

It deals with collisions by moving keys between the two tables. A new key x_1 is always inserted into $T_1[h_1(x_1)]$. If this cell is occupied by a key x_2, x_2 is moved out of T_1 into T_2 at $T_2[h_2(x_2)]$. If this cell is occupied by x_3, the key x_3 becomes "nestless" and is evicted from T_2 to T_1. This is repeated until an empty cell is found. See Figure 7.

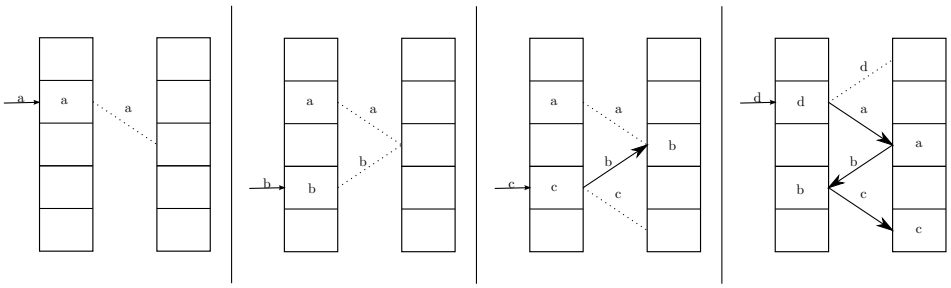


Figure 7: An example for the insertions of 4 elements into the hash table in cuckoo hashing. Solid lines visualize evictions, dashed lines illustrate the two different places for an element.

Clearly, this procedure could also end up in an infinite loop, if the same keys are evicted again and again. In order to avoid infinite loops, Pagh and Rodler suggest to define a maximum number *MaxLoop* of nestless keys during the insertion procedure, and to perform a rehash of the whole data structure once this limit is reached. We will first formalize the used algorithms for the *lookup*, *delete* and *insert* operations and describe requirements for infinite loops in the insertion procedure. These observations will yield possible values for *MaxLoop*.

The lookup procedure is pretty simple in cuckoo hashing. We can look up an element x by checking the table cells $T_1[h_1(x)]$ resp. $T_2[h_2(x)]$.

Procedure `lookup(`X`)`

Input: In X: an element $x \in U$ that we want to look up.
Output: True, if the element is in the hash table, false otherwise.

return $(T_1[h_1(X)] = X) \vee (T_2[h_2(X)] = X)$;

Let $\perp$ denote an empty table cell. Let $x \leftrightarrow y$ denote the operation of swapping the contents of x and y. To insert an element x into the hash table, we start by inserting x into the first table. If some element y was in cell $T_1[h_1(x)]$, we evict it. This element is now put into the second table. If the table cell is occupied as well, we insert the next element in the first table again, and continue in this manner until we succeed and find a vacant cell or fail because the algorithm runs for more than *MaxLoop* steps. In this case we "rehash" the data structure (with new hash functions) and try to insert the after *MaxLoop* steps nestless element again.

We can formalize the insertion procedure in the following way, where all keys in the data structure are said to be unique.

Procedure `insert(`X`)`

Input: In X: an element $x \in U$ that we want to insert.

if `lookup(`X`)` **then return**;
$i \leftarrow 0$;
repeat
  $X \leftrightarrow T_1[h_1(X)]$;
  if $X = \perp$ **then return**;
  $X \leftrightarrow T_2[h_2(X)]$;
  if $X = \perp$ **then return**;
  $i \leftarrow i + 2$;
until $i \geq MaxLoop$;
`rehash`();
`insert`(X);

It is an interesting task to implement and analyze the rehash function. At this moment, we are not armed with results on the running time and the behavior of the insert procedure. Thus, we postpone the description and analysis of the rehash procedure and keep focused on the missing delete operation.

To delete a key we have to check both possible positions and remove the key if we find it. As the insertion procedure never includes the same key twice, we can return from the procedure if we found the key in the first cell to avoid the memory access in the second table.

Procedure `delete`(x)

Input: In X: an element $x \in U$ that we want to delete.

if $T_1[h_1(X)] = X$ **then**
  $T_1[h_1(X)] \leftarrow \perp$;
  return;
if $T_2[h_2(X)] = X$ **then**
  $T_2[h_2(X)] \leftarrow \perp$;

We are now interested in describing the events that could happen and possibly yield an infinite loop in the insertion procedure. We can describe the insertion of an element x into a hash table based on the cuckoo hashing scheme by looking at the sequence of nestless keys during an insertion. The insertion of the element d in Figure 7 can thus be described with the sequence d, a, b, c of nestless keys, where d is placed into the first table and starts a chain of key movements that moves all following keys into their alternative location. For such a sequence of keys without a repetition, we can clearly state that the insertion procedure will succeed.

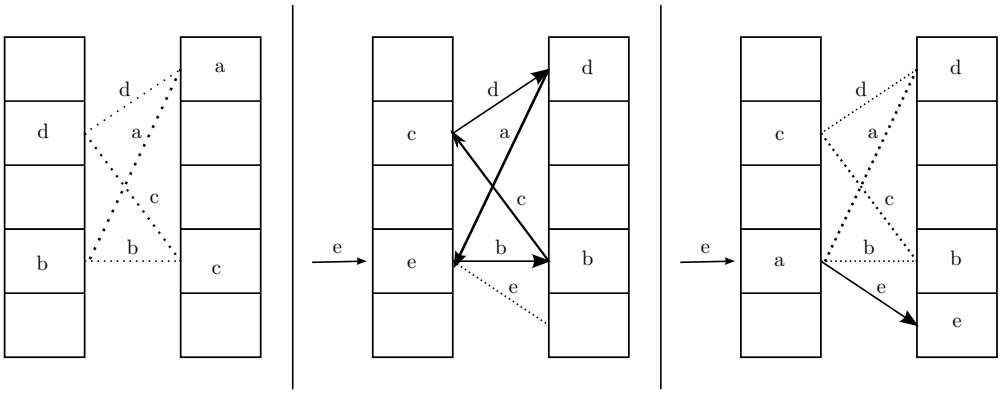


Figure 8: An example for the case that the inserted key is placed into the second table. The insertion of e yields the nestless keys e, b, c, d, a, where the alternative choice of a is the cell of e and thus e is moved to the second table.

As we can see in Figure 8, it could also happen that the insertion algorithm evicts x to the second table. This happens if a key is evicted to a previously visited cell. In that case, a key y that was evicted into its alternative cell is moved to its original position prior to the insertion of x. Obviously, this key will start a chain of evictions

that ends in x being evicted to the second table, because all keys between x and y are placed into their original place and eventually x is thrown out of its cell.

The same game starts all over again, because the alternative table cell of x could be occupied, too. If the eviction chain ends with a key that is placed into a vacant cell, it clearly succeeds as well.

If such a vacant cell does not exist, there is a key z that is moved to a previously visited cell after x was placed into the second table. Note that this closes the loop, because all keys have already been put into their alternative places and hence there cannot exist a vacant cell[2]. See Figure 9 for an example.

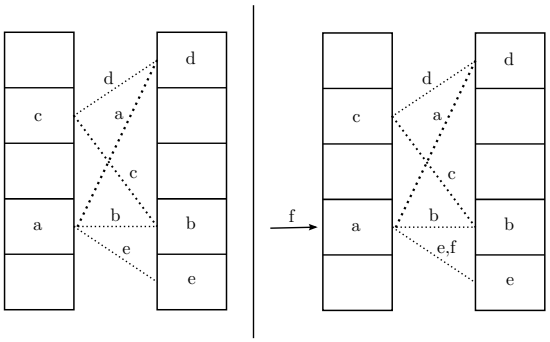


Figure 9: An example for the case that the inserted key yields a loop and a rehash is required. The new key f can be placed into the same table cells as the key e. Inserting f yields the eviction chain $f, a, d, c, b, f, e, b, c, d, a, e, \ldots$.

We want to point out that discovering a loop is possible in another way. If a key is moved to its alternative cell, then moved back to its original place and after a while is evicted again to its alternative choice, it is impossible that the insertion can succeed, because after it has been placed into its original cell a further eviction is an indicator of the loop structure mentioned above.

This observation can be used to find a value for *MaxLoop*. If a loop structure occurs after three evictions of an element, we just have to run $2n + 1$ steps, where n denotes the number of elements stored in the hash table. Using the pigeonhole principle, there must exist one element that has been moved for a third time after this amount of steps.

Clearly, this result is disappointing, because it is bounded by $\mathcal{O}(n)$. As an advantage, a rehash only occurs if the insertion procedure cannot successfully insert the key x into the hash table. We call a value for *MaxLoop idealized*, if a rehash is only considered if the data structure cannot be built successfully. We can use a lower value for *MaxLoop*, but have to take into account the probability that the insertion starts a rehash although the key could be successfully inserted.

The analysis of cuckoo hashing in general can be simplified by considering it from a different point of view.

[2] An animation that visualizes these different cases can be found at `http://pkqs.net/~tre/projects/cuckoo_hashing_viz/insertion_procedure.html`. Firefox 3 unfortunately required!

3.2. The Cuckoo Graph

Following [Sch09], for a given set $S \subseteq U$ and hash functions h_1, h_2, we say that h_1 *and* h_2 *are suitable for* S if it is possible to store the keys S in T_1, T_2 according to h_1, h_2 in such a way that distinct keys are stored in distinct table cells. We can examine the cuckoo hashing scheme from a graph theoretical point of view, where we define the *cuckoo graph*[3] $G(S, h_1, h_2)$ as follows.

The vertices of the cuckoo graph correspond to the cells in the tables T_1 and T_2, and an edge labeled with $x \in S$ connects two vertices u, v if $u = h_1(x)$ and $v = h_2(x)$. $G(S, h_1, h_2)$ is by definition an undirected bipartite multigraph (L, R, E) with $L = [m]$ and $R = [m]$ disjoint, and edge multiset $E = \{(h_1(x), h_2(x)) \mid x \in S\}$ (Fig. 10).

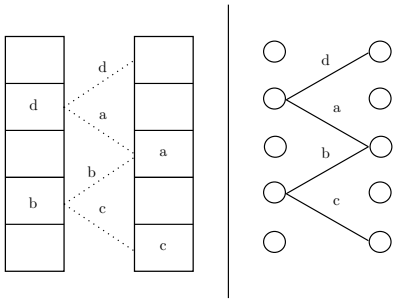


Figure 10: The cuckoo graph representing a given hash table allocation in cuckoo hashing. Table cells are represented by the vertices of a bipartite multigraph and the two possible positions of the keys inside the tables by edges between two vertices.

As keys in the data structure are unique edges in the cuckoo graph, we use the term *key* in the cuckoo graph for the edge labeled with this key.

Interestingly, the cuckoo graph $G = G(S, h_1, h_2)$ provides us with information if h_1, h_2 are suitable for S. Obviously, each connected component C in G must have at least as many vertices as edges; otherwise there would be more keys than available memory cells. Recalling our results about the cyclomatic number of a graph, this means that C has a cyclomatic number of at most 1. On the other hand, if the cyclomatic number is equal or less than 1, we need to show that the insertion procedure will succeed for all keys $x \in S$.

Following [Sch09], we introduce the notion of a *bad edge set*.

Definition 3.1
Let $G(S, h_1, h_2) = (L, R, E)$ be a bipartite undirected multigraph. We call $E' \subseteq E$ a *bad edge set* if E' induces a connected subgraph and $|E'|$ is larger than the number of vertices that are incident with edges in E'.

[3] The notion of the cuckoo graph goes back to [DM03] and was also used by [Kut09].

If a connected graph has more edges than vertices, we already know that it contains at least two cycles. In the first part of this chapter we described the insertion algorithm and found requirements for the insertion procedure to succeed. In the same manner, we consider a random walk on the edges of the cuckoo graph to describe the events that happen during the insertion of a key. We can state the following two lemmas.

Lemma 3.2 ***(Devroye and Morin [DM03])***
The hash functions h_1 and h_2 are suitable for S if and only if $G(S, h_1, h_2)$ does not contain a bad edge set.

Proof. If G contains a bad edge set, G has a connected component that has more edges than vertices and thus there are more keys than available memory cells. Clearly, h_1, h_2 are not suitable in this case.

Now assume G does not have a bad edge set. Insert the keys of S one after the other. If all insertions succeed, the hash functions are suitable. Thus we assume that the insertion of a key x does not succeed. Then the last of the previously discussed cases happens. The insertion of x will evict a key in T_1 and there will exist a key that is moved to a previously visited cell. Note that this closes a first cycle and will result in x being evicted to the second table. There we will also find a key that is moved to a previously visited cell, which closes a second cycle. Thus, there exists a connected component with two cycles and G contains a bad edge set. □

Lemma 3.3
The hash functions h_1 and h_2 are suitable for S if and only if each connected component of $G(S, h_1, h_2)$ is either a tree or unicyclic.

Proof. We use Lemma 3.2 for both directions of the proof. If h_1 and h_2 are suitable for $G = G(S, h_1, h_2)$ then G has no bad edge set. In this case every connected component contains at most as many edges as nodes, and hence they are either acyclic or unicyclic. On the other hand, if h_1 and h_2 are not suitable for G, it contains a connected component C with a bad edge set. Following the definition of a bad edge set, C has more edges than nodes and thus contains at least two cycles. □

One can get a good impression of graph structures with bad edge sets by looking at *minimal bad edge sets*, where we call a bad edge set $E' \subseteq E$ minimal if $\forall E'' \subset E' : E''$ is no bad edge set. If E' is a minimal bad edge set then the induced subgraph $G[E']$ is connected, contains one edge more than the number of vertices and has no leaves. The two cases for a minimal bad edge set are shown in Figure 11. Either it is a cycle with a chord (a path connecting two non-adjacent vertices on the cycle) or two edge-disjoint cycles connected by a simple path of length at least 0.

This close relationship between the hash scheme and the graph representation allows us to analyze cuckoo hashing by analyzing properties of the cuckoo graph $G(S, h_1, h_2)$.

The probability of the successful insertion of all elements $x \in S$ into the hash table is bounded by the probability that a random bipartite multigraph with $2m$ vertices and n edges has no connected component that has more than one cycle. Furthermore,

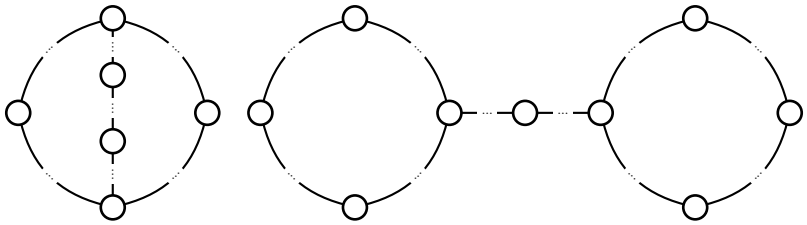


Figure 11: The two possible structures of minimal bad edge sets.

one can analyze the costs of inserting a given key x by observing that this is equal to inserting an edge $(h_1(x), h_2(x))$ into the cuckoo graph and looking at the size of its connected component.

One could ask about the probability that h_1 and h_2 are not suitable for S. We will prove the following theorem.

Theorem 3.4 ***(Pagh and Rodler [PR01])***
Let $m = (1+\varepsilon)n$ for an arbitrary constant $\varepsilon \in (0,1)$. Then the insertion procedure fails to insert all n keys with probability $\mathcal{O}(m^{-1})$ if the hash functions behave fully randomly.

Proof. We know that the insertion fails if the cuckoo graph contains a bad edge set. In a bad edge set, we can identify a minimal bad edge set. We can find an edge-disjoint walk that visits every edge exactly once. Label these edges with $x_1, \ldots, x_k$. We first omit the first and last vertex in this walk and remark that a simple path remains (Fig. 12).

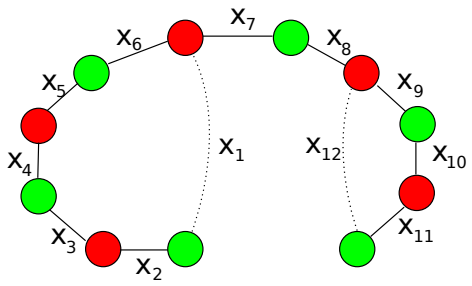


Figure 12: Minimal bad edge set with $k = 12$ edges. Green vertices represent table cells in the first table, red vertices in the second table.

Let the simple path start with hash function h_i and let h_j denote the other hash function. A simple path on $x_1, \ldots, x_k$ is then based on the sequence

$$h_i(x_1) = h_i(x_2), h_j(x_2) = h_j(x_3), h_i(x_3) = h_i(x_4), \ldots, h_{i/j}(x_{k-1}) = h_{i/j}(x_k).$$

As the hash function behave fully randomly, the probability for a collision is $1/m$. Let us consider the endpoints of the walk again, which close the cycles of the bad edge set.

Clearly, the keys associated with the cycle edges have one place in the first and one in the second table. As we know one endpoint, $\lfloor (k-1)/2 \rfloor$ choices can be made for the unknown endpoint of a cycle edge[4]. The probability for a collision in one of these endpoints is $1/m$ and hence the probability for such a cycle edge endpoint is bounded by $k/2m$. We have fewer than n^k choices for the sequence of edges that forms the path $x_1, \ldots, x_k$ and it could either start on the first or the second hash function. Hence we can express the probability that the insertion procedure fails to insert all keys and contains a bad edge set as follows.

$$\begin{aligned}
\Pr(\text{the insertion of all keys yields a bad edge set}) &\leq \sum_{k=3}^{n} n^k 2\Big(\frac{1}{m}\Big)^{k-1}\Big(\frac{k}{2m}\Big)^2 \\
&= \frac{1}{2m}\sum_{k=3}^{n}\Big(\frac{n}{m}\Big)^k k^2 \\
&= \frac{1}{2m}\sum_{k=3}^{n}\frac{k^2}{(1+\varepsilon)^k} \\
&= \mathcal{O}\Big(\frac{1}{m}\Big).
\end{aligned}$$

□

Note that the same argument holds if we ask if a specific key x is contained in such a bad edge set. In this case, we have only n^{k-1} other keys to choose from. Furthermore, x could be on an arbitrary location in the sequence of edges and thus

$$\begin{aligned}
\Pr(x \text{ is in a bad edge set}) &\leq \sum_{k=3}^{n} k n^{k-1} 2\Big(\frac{1}{m}\Big)^{k-1}\Big(\frac{k}{2m}\Big)^2 \\
&\leq \frac{1}{2m^2}\sum_{k=3}^{n}\Big(\frac{n}{m}\Big)^{k-1} k^3 \\
&= \frac{1}{2m^2}\sum_{k=3}^{n}\frac{k^3}{(1+\varepsilon)^k} \\
&= \mathcal{O}\Big(\frac{1}{m^2}\Big).
\end{aligned}$$

Thus, the probability that the insertion of a key x yields a bad edge set is bounded by $\mathcal{O}(n^{-2})$.

[4] A minimal bad edge set has cyclomatic number 2. If it has k edges, it contains $k-1$ vertices. If $k-1$ is odd, the simple path starts and ends in the same table i. The endpoints lie in the other table j and there are $\lfloor (k-1)/2 \rfloor$ such table cells on the path. See Figure 12.

It is interesting to note that weak but commonly used hash functions can yield a higher failure probability in cuckoo hashing than one could expect from the previous results. Dietzfelbinger and Schellbach proved in [DS09a] that in the case of a dense key set in the universe (like $|S| \geq |U|^{11/12}$), the linear and multiplicative class of hash functions perform much worse. Even in the case of a large universe, the multiplicative class of hash functions is unsuitable for cuckoo hashing [DS09b]. Other simple hash functions, e.g., the class of cubic hash functions, perform well in experiments [PR04].

After we have seen that cuckoo hashing yields a low failure probability if we avoid some special classes of hash functions, we are interested in the running time of the insert operation.

3.3. Analysis of the Insert Operation

Analyzing the insertion time is equivalent to considering the number of keys that are evicted when a key is inserted. We will first consider keys that can be successfully inserted into the hash table without triggering a rehash and start by proving the following lemma.

Lemma 3.5
If a key x is successfully inserted in t steps then the cuckoo graph contains a simple path starting at $h_1(x)$ or $h_2(x)$ of length at least $\lceil \frac{t-1}{3} \rceil$.

Proof. Consider the component that is induced by the key movements when a key x is inserted. If x can be inserted successfully, we consider two mutually exclusive cases. In the first case, x is inserted into the first table at position $h_1(x)$ and is not moved to the second table afterwards. Then no key is moved twice and hence the complete path is simple.

In the second case, x is moved to the second table. Consider the path starting from $h_2(x)$. This path is simple, because the insertion of x succeeds. If this path has length at least $\lceil \frac{t-1}{3} \rceil$, we are done. Otherwise, the simple path p starting at $h_1(x)$ has length at least $\lceil (t - \lceil \frac{t-1}{3} \rceil)/2 \rceil \geq \lceil \frac{t}{3} \rceil$. This situation is visualized in Figure 13. □

Using this lemma, we can state an upper bound on the probability of the event that a successful insertion of a key x takes more than t steps.

Lemma 3.6
Assume that a key x can be inserted successfully in the hash table and let $\#\text{evictions}_x$ denote the number of evictions in the table until the insertion of x succeeded. Then

$$\Pr(\#\text{evictions}_x \geq t) \leq 2(1+\varepsilon)^{-\lceil \frac{t-1}{3} \rceil}.$$

Proof. If the insertion of a key $x_1 = x$ succeeds in t steps, then there exists a simple path of length at least $l = \lceil \frac{t-1}{3} \rceil$ starting at $h_1(x_1)$ or $h_2(x_1)$. Let denote with h_i the hash function that is used for the start of the path and h_j the other hash function.

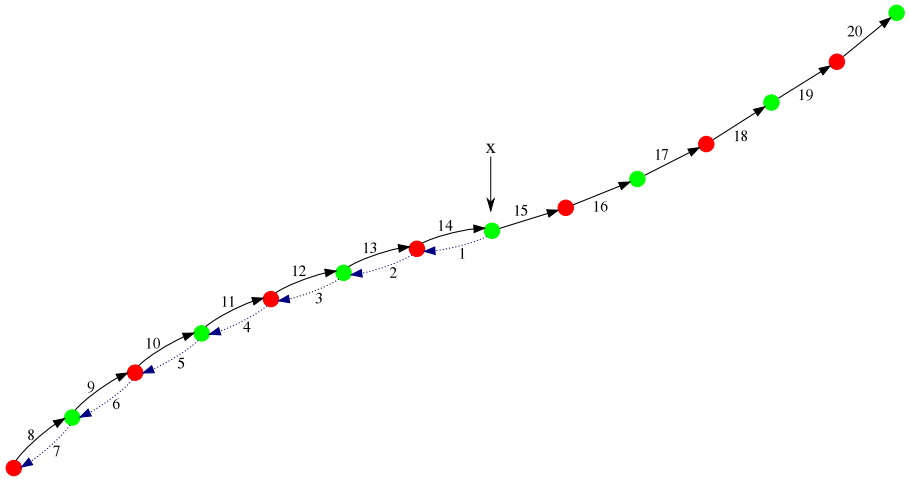


Figure 13: An example for a successful insertion of a key x with $t = 20$ evictions. Green vertices represent table cells in the first table, red vertices in the second table. The longest simple path starting at $h_1(x)$ or $h_2(x)$ is dashed and has length $\lceil \frac{t-1}{3} \rceil = 7$.

Furthermore, for such a path to exist, the hash functions have to provide the following structure on keys $x_1, \ldots, x_l$.

$$h_i(x_1) = h_i(x_2), h_j(x_2) = h_j(x_3), h_i(x_3) = h_i(x_4), \ldots, h_{i/j}(x_{l-1}) = h_{i/j}(x_l),$$

where the probability for a collision of two distinct keys is bounded by $1/m$, because we assume the hash functions to behave fully randomly. There are fewer than n^l ways of choosing the sequence of keys $x_1, \ldots, x_l$. Moreover, we could start the path at $h_1(x_1)$ or $h_2(x_1)$. Hence we can express the probability that the insertion of a key x moves at least t keys by

$$\begin{aligned} \Pr(\#\text{evictions}_x \geq t) &\leq 2n^l \Big(\frac{1}{m}\Big)^l \\ &= 2\Big(\frac{n}{m}\Big)^l \\ &= 2(1+\varepsilon)^{-l}, \end{aligned}$$

and the lemma follows for $l = \lceil \frac{t-1}{3} \rceil$. □

This gives us the opportunity to obtain the expected insertion time of a successful insertion.

Lemma 3.7
Let x be a key that can be successfully inserted into the hash table. Then the expected insertion time of x is constant.

Proof.

$$\begin{aligned}\mathrm{E}(\#\text{evictions}_x) &= \sum_{t\geq 1} \Pr(\#\text{evictions}_x \geq t) \\ &= \sum_{t\geq 1} 2(1+\varepsilon)^{-\lceil \frac{t-1}{3} \rceil} \\ &= \mathcal{O}(1).\end{aligned}$$

□

We have already seen that we can use $2n+1$ as an idealized value for *MaxLoop*, where n denoted the number of keys in the data structure. If we choose a lower value for *MaxLoop*, we have to take into account that the insertion procedure rehashes although the key could be inserted.

By Lemma 3.6 we can choose a value of $c\lceil \log_{1+\varepsilon} n \rceil + 1$ for *MaxLoop* and know that the probability that the insertion procedure will fail, although the key could be inserted, is at most

$$2(1+\varepsilon)^{-\lceil \frac{MaxLoop-1}{3} \rceil} \leq 2n^{-c/3}.$$

For the following analysis, we will use *MaxLoop*$= 6\lceil \log_{1+\varepsilon} n \rceil + 1$, resulting in a failure probability due to long paths of $2n^{-2}$. We have already seen that the probability that a key x can be inserted successfully without triggering a rehash event is high, but we are further interested in the insertion time for a realistic scenario involving rehashes. As the rehash operation is of importance to us now, we should step back and take a closer look at it.

The rehash operation can be implemented in several ways, but we always start by choosing new hash functions h_1 and h_2.

A naïve implementation is to allocate two new tables, subsequently iterate through the table cells of the old tables and insert every key we find into the new table. This way would involve n insert operations and the allocation of two arrays of length m. Does there exist a variant without additional memory involved?

The solution is an in-place rehash operation. Once again, we iterate through every table cell. If the table cell is occupied by a key x, we remove the key from its current cell and insert the key using the traditional insert operation at $h_1(x)$ into the first table. If this cell is free, we are done. On the other hand, if it is occupied, two different kinds of keys could be located there. If x is moved to a table cell that already contains a key processed during our rehash operation, this collision is "real". If we did not visit this cell before during the rehash operation, the key in this cell is most likely at the wrong

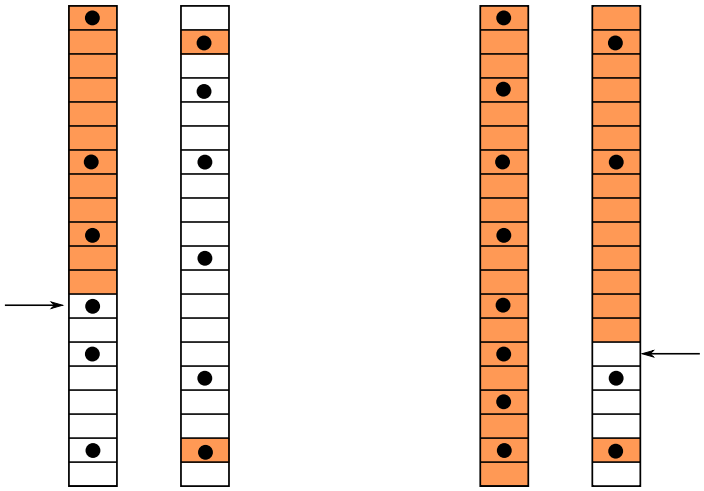


Figure 14: Different states of the rehash operation. On the left side, the rehash function processed the first half of the first table and 5 keys are already in correct positions. On the right side, the rehash operation is almost finished.

position. In this case, the insertion of x virtually ends here and the wrongly placed key is now inserted for the first time.

In both cases, we use the common approach to store x in the table cell, move the other key to the alternative table and continue with the insert operation. It is important to note that for both cases the key is placed into a correct table cell — that is an actual table cell using the new hash functions. If we mark cells that are processed or already contain keys in their correct new position orange, Figure 14 visualizes different states of the rehash operation.

Note that $2m$ cells are processed in this way, and all cells prior to the current cell only contain correctly placed keys. We can refine this approach in several ways.

First of all, it could happen that we insert a key twice, because it could be placed ahead of the currently processed cell and thus will be visited again. We can look up a key before we try to insert it to skip these unnecessary insertions, leading to at most $2n$ additional lookup operations.

Moreover, we could unnecessarily place keys into the second table, although their cell in the first table is free. This happens if we evict a key from the first table that was not in its correct position and move it to the second table. This yields a longer average lookup time, because lookups for elements in the first table require only one memory access. We could solve this problem by using flags on each table cell to indicate if it has been processed already. If this is not the case, we could insert this key to the first table[5]. Note that this solution involves additional memory. We could also use the

[5]This behavior actually matches the observation that a collision in a previously unvisited cell finishes the insertion and starts to insert the colliding key which was most likely at a wrong position.

lookup approach, adding some constant overhead to every insertion step.

No matter which refinement we actually use, it can be summarized that the in-place rehash operation works using $\mathcal{O}(n)$ insert operations if the new pair of hash functions is suitable for S. We can now continue to develop bounds on the running time of the insert operation.

Theorem 3.8
Assume that (h_1, h_2) behaves fully randomly and let $m = (1 + \varepsilon)n$.
Then insertions can be carried out in amortized expected constant time.

Proof. We have already seen that the expected time for a successful insertion is constant. With a probability of $\mathcal{O}(n^{-2})$, the insertion of a key forces a rehash in the data structure. As we have seen, a rehash requires $\mathcal{O}(n)$ insertions. The expected running time for a rehash is hence $\mathcal{O}(n^{-1})$ and recursive rehashes during a rehash only add lower order terms, because the probability that a rehash fails is $\mathcal{O}(n^{-1})$. If we insert n keys to the data structure, we expect that this happens in $\mathcal{O}(n)$ and thus the amortized expected running time is $\mathcal{O}(1)$. □

Following [PR04], we also invoke the rehash operation after m^2 successful insertions. The expected running time of the rehash operation is $\mathcal{O}(n)$ and thus the amortized cost per insertion for this extra rehash is only $\mathcal{O}(n^{-1})$.

Knowing this important result, we should consider practical idealized insertion algorithms. We have seen that an idealized insertion algorithm may abort if there exists a key evicted for a third time. As those insertions only take place in a connected component, such an algorithm has worst-case running time $2n' + 1$, where n' is the size of the connected component in which the insertion takes place. It is an interesting question to obtain the expected size of such a component.

The following lemma comes from *Cuckoo Hashing for Undergraduates*[6] by Rasmus Pagh. We provide a different (but slightly inaccurate) proof in Section A.1.

Lemma 3.9
Let $G = G(S, h_1, h_2)$ denote the cuckoo graph for the key set $S \subseteq U$ of size n and fully random hash functions. Let $m = (1 + \varepsilon)n$. Let $x \in S$ denote an edge of G.
Then the expected size of the connected component of x is constant.

Proof. Let X_{xy} denote the random variable whose value is 1 if the edges x and y are contained in the same component. In this case, there exists a path of the form $x, x_1, x_2, \ldots, x_l, y$ for $l \geq 0$ in the cuckoo graph. The argument is now similar to the one in Lemma 3.6.

If such a path exists, the keys $(x, x_1), (x_1, x_2), \ldots, (x_l, y)$ collide with each other and the probability for these $l + 1$ collisions is m^{-l-1} as we assume fully random hash functions. Furthermore, this path could start with hash function h_1 or h_2 and we have

[6] http://www.it-c.dk/people/pagh/papers/cuckoo-undergrad.pdf

fewer than n^l choices for the sequence of edges $x_1, \ldots, x_l$. It follows

$$\begin{aligned}\Pr(X_{xy} = 1) &\leq \sum_{l=0}^{n} 2n^l m^{-l-1} \\ &= 2/m \sum_{l=0}^{n} \Big(\frac{1}{1+\varepsilon}\Big)^l \\ &= \mathcal{O}(n^{-1}).\end{aligned}$$

The expected size of the connected component containing x follows directly by

$$\begin{aligned}\mathrm{E}(\sum_{y \in S \setminus \{x\}} X_{xy}) &= (n-1)\mathcal{O}(n^{-1}) \\ &= \mathcal{O}(1).\end{aligned}$$

□

Thus, the expected running time of the idealized insertion algorithm is constant. The implementation of such an algorithm is an interesting question. We cannot state which key is being moved for a third time first. Thus, the algorithm would involve some bookkeeping to store the evictions of each item. This would be a tremendous effort for such an apparently simple task. We propose and analyze a more practical variant of an idealized insertion algorithm.

Let x denote the key being inserted. We only focus on the number of evictions of x, where its first insertion is already counted. If x is moved for the third time, we abort and can be sure that the insertion cannot succeed. What is the worst-case running time of this algorithm?

Let us consider the example in Figure 15.

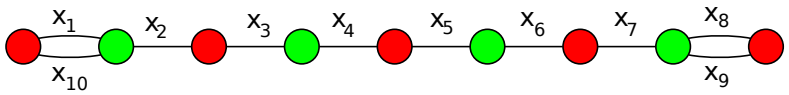


Figure 15: Insertion of a key x_1 such that x_1 is not the first item that is moved for the third time. Green vertices represent table cells in the first table, red vertices in the second table.

In this example, x_1 evicts x_2 and the algorithm runs through the cycle at the end and eventually comes back to x_1 and moves it into the second table. It evicts x_{10} there, and the game starts anew. Note that x_2 is thus the first item that is evicted for the third time. The insertion algorithm would stop after the algorithm runs through the loop again, x_2 evicts x_{10} and x_{10} finally moves x_1 for the third time. If we partition this example into "the left side" x_{10} and "the right side" $x_2, \ldots, x_9$, it is clear that in a worst-case scenario one key on the left side evicts one key on the right side because otherwise x_1 would be the first key being evicted for the third time. Note that all keys on the right side, except the two keys on the cycle, are evicted four times. To get the longest eviction chain, there should be a minimum of keys moved only twice and a maximum of keys moved four times. It is easy to see that at least four keys are moved only twice. The number of evictions in the worst-case is hence $2 \cdot n + 2(n-4) = 4n - 8$ and this represents the worst-case running time of the idealized memory-efficient insertion algorithm for $n \geq 5$. The expected running time of this algorithm is still constant, but it runs almost twice as long as the $2n + 1$ variant in the worst case. The absence of any involved bookkeeping represents a big advantage though.

In summary it can be said that cuckoo hashing provides an efficient data structure to maintain a dynamic dictionary. Lookups and deletions can be carried out in worst-case constant time and insertions in amortized expected constant time. Moreover, the probability that a key set cannot be inserted with given hash functions is bounded by $\mathcal{O}(n^{-1})$.

Following [KMW08], a failure probability of $\mathcal{O}(n^{-c}), c \leq 3$, could already lead to failures at a too high rate for commercial applications, e.g., in packet statistics in high-performance routing scenarios or database indexing. In particular, in many applications such as indexing, elements are inserted and deleted from the hash table over a long period of time, increasing the probability of a failure at some point throughout the life of the table. If that is true, one should try to improve the failure probability of the cuckoo hashing scheme.

Kirsch et al. [KMW08] overcome this weakness by suggesting a small constant-sized memory, the so-called *stash*, to store keys that could not be inserted into the hash table due to the structures we presented in this section.

3.4. Cuckoo Hashing with a Stash

Cuckoo hashing with a stash was introduced by Kirsch et al. in [KMW08]. It handles the problem of cuckoo hashing failing to insert a key set S with probability $\mathcal{O}(n^{-1})$. Keys that cannot be inserted into the data structure are placed into a small extra piece of memory called *stash*. This memory is said to be a simple array, but could also be implemented as an associative array, using, e.g., a hash table approach. In embedded scenarios, we could use special hardware to realize the stash. Their analysis shows that the failure probability is reduced dramatically, assuming fully random hash functions are available.

We will first describe the algorithms for the *lookup*, *insert* and *delete* operations in our new scenario. We denote the stash with St and define the membership operation '$x \in \mathrm{St}$', returning true if the element x is stored in the stash. In the pseudocode, we will use array notation to iterate through the stash; thus $\mathrm{St}[i]$ is the i-th element of the stash. Furthermore, we will denote the current stash load with s and the maximum physical stash size with $\hat{s}$.

The *lookup* procedure has to check the two possible locations of x in the tables T_1 and T_2. Furthermore, x could reside in the stash and thus we have to check the stash elements as well.

Procedure `lookup(`X`)`

Input: In X: an element $x \in U$ that we want to look up.
Output: True, if the element is in the hash table or the stash, false otherwise.

return $(T_1[h_1(X)] = X) \vee (T_2[h_2(X)] = X) \vee (X \in \mathrm{St})$;

One drawback of cuckoo hashing with a stash is thus a worse running time on lookup operations, for more than two locations are now possible for a key. We will restrict the capacity of the stash to be a small constant $\hat{s}$. At the end of this thesis, we will observe experimentally that such an assumption is practically sound.

Looking at modern computer architectures using a cache, we argue that keys in the stash are frequently accessed during a series of lookup operations and thus are likely to be located in the cache of the CPU, whereas random access in the two tables T_1 and T_2 is arguably slower. The impact on the practical running time is hence not as dramatic as it seems. Implementing cuckoo hashing on embedded systems has to cope with further restrictions, often missing a suitable cache structure or caches of very restricted size. If we could determine the practical size of the stash in a given scenario and if a small constant-sized stash is sufficient, content-addressable memories, which are cost-efficient on small scales, could be used to improve cuckoo hashing there.

The insertion procedure is almost similar to its equivalent in standard cuckoo hashing, but instead of rehashing as soon as an insertion has taken *MaxLoop* steps, we place the key that is currently nestless in the stash while this is possible. If the stash is full and the insertion of another element fails, we execute the already known rehash approach in a slightly modified way. We start it with insertions of all stash elements and afterwards use the method described in the previous section.

Procedure `insert(`X`)`

Input: In X: an element $x \in U$ that we want to insert.

```
if lookup(X) then return;
i ← 0;
repeat
    X ↔ T1[h1(X)];
    if X = ⊥ then return;
    X ↔ T2[h2(X)];
    if X = ⊥ then return;
    i ← i + 2;
until i ≥ MaxLoop ;
if s = ŝ then
    rehash();
    insert(X);
    return;
St[s] = X;
s ← s + 1;
```

Obviously, the insertion procedure always succeeds in storing the keys into the hash table until the stash is full, because the insertion of an element is aborted after *MaxLoop* steps and the key that is currently nestless is placed into the stash. If the stash overflows, we rehash.

To delete a key we have to check both possible positions and the whole stash. As the insertion procedure never includes the same key twice, we can return from the procedure if we found the key to avoid additional memory accesses.

Procedure `delete(`X`)`

Input: In X: an element $x \in U$ that we want to delete.

```
if T_1[h_1(X)] = X then
    T_1[h_1(X)] ← ⊥;
    return;
if T_2[h_2(X)] = X then
    T_2[h_2(X)] ← ⊥;
    return;
for i ← 0 to s − 1 do
    if St[i] = X then
        St[i] ← ⊥;
        s ← s − 1;
        return;
```

This algorithm is a naïve extension of the deletion algorithm for standard cuckoo hashing presented in the first section of this chapter. We have to consider one important problem.

What happens if the deletion of a key x makes it possible to store a stash key y in the tables again? Ideally, inserting and deleting x should yield the same stash size as if x had not been inserted at all. At the moment we cannot state and analyze a feasible solution and assume that we only use the *insert* and *lookup* operations as described. We will consider suitable variants of the deletion algorithm in Section 4.4.

We are especially interested in the size of the stash and bounds on the event that the stash size exceeds a given value (especially the maximum physical size), but we will first consider the structure of the cuckoo graph when using the described insertion algorithm and get back to the former questions in the next chapter. For the following observations, we consider only the idealized variant of the insertion procedure.

As we have seen in the previous sections, we can decide if a set S of keys can be successfully inserted into the data structure using two hash functions h_1 and h_2 by looking at the structure of the cuckoo graph $G = G(S, h_1, h_2)$. We have identified a property of the graph that made the insertion certainly fail and called the associated structure a *bad edge set*. A bad edge set was a connected subgraph induced by edges of the cuckoo graph G that contained two or more cycles. As soon as such a structure was implicitly detected by the insertion procedure (it was stuck in a loop), we built the data structure anew using new hash functions.

As we can see in the description of the insertion procedure, we try an alternative approach now. Instead of rehashing the complete structure, we place a key into the stash and only start a rehash if the stash cannot store an element, because it is already full. Putting a key into the stash is equivalent to removing an edge from the bad edge set, because the only keys visited by the insertion procedure are contained in the bad edge set. Thus we obtain a graph whose edge set is a subset of the cuckoo graph's edge

set, because some keys have been moved to the stash. If the stash is not sufficiently large, we try to insert the keys anew with new hash functions. We denote the resulting graph with G_s.

The components of G and G_s are associated with each other. A component C in G is possibly decomposed into a number of components $C'_1, \ldots, C'_k$ in G_s. In the case $k = 1$, the component C'_1 has exactly the same vertices as C, but might miss some edges. If $k > 1$, the components form a partition of the vertex set of C. We observe the following properties. All components of G_s are either acyclic or unicyclic, because otherwise they would contain a bad edge set. Furthermore, C' is only a tree if C is a tree, because the idealized insertion algorithm never removes an edge from a unicyclic component, which, by definition, cannot contain a bad edge set. It follows that if a component C of G is decomposed, all these components are unicyclic, because the idealized insertion algorithm only destroys connectivity by removing an edge that connects cyclic components.

The next chapters will consider an analysis of cuckoo hashing with a stash that is different from the one given in [KMW08], assuming fully random hash functions first and considering hash functions from a special class of hash functions introduced by Dietzfelbinger and Woelfel in [DW03] later.

4. Analysis of Cuckoo Hashing with a Stash and Fully Random Hash Functions

This chapter will analyze the behavior of cuckoo hashing with an additional stash using fully random hash functions. In the first part we consider the size of the stash. As we have seen in the previous preliminaries section, we can break down the analysis of cuckoo hashing by observing specific properties of the bipartite random multigraph $G(S, h_1, h_2)$. Using the observations of the previous chapter, we are able to present the size of the stash as a simple equation.

Furthermore, it is important to obtain an upper bound for the probability that a random bipartite multigraph $G(S, h_1, h_2)$ has at least s edges that must be placed into the stash. We prove the result that the probability that this event happens is bounded by $\mathcal{O}(n^{-s})$. It is important to note that this analysis was already conducted by Kirsch et al. [KMW08] using sophisticated stochastic methodology and Kutzelnigg [Kut09] using generating functions and the saddle point method. We present an alternative analysis.

We denote the standard cuckoo graph introduced in Chapter 3 by $G = G(S, h_1, h_2)$ and the graph without stash keys by G_s. Furthermore, we use the idealized version of the insertion algorithm during the first two sections to assure that it only puts an edge into the stash if the insertion is stuck in an infinite loop and therefore cannot succeed.

At the end of this section, we will take a look at insertions and deletions in stash-based cuckoo hashing. The stash scenario adds an interesting new condition to the insertion of keys and we will see, if they are still expected to run in amortized constant time. We will also consider the problem of the naïve deletion algorithm observed during the last chapter and try to obtain a suitable solution.

Let us start by describing the size of the stash.

4.1. The Size of the Stash

As we have seen in Section 3.4, the idealized insertion algorithm puts edges into the stash to resolve loops. For now we restrict the operations to insertions and lookups, but we never remove a key considering the problems we observed with a naïve deletion algorithm in the stash-based scenario.

The main goal of this section is proving the following lemma that expresses the size of the stash by using a well-known property of a graph – its excess.

Lemma 4.1
After all elements $x \in S$ have been inserted into the data structure represented by the cuckoo graph $G = G(S, h_1, h_2)$, there are exactly

$$\mathrm{ex}(G) = \gamma(G) - \mathrm{cc}(G)$$

keys in the stash, where $\gamma(G)$ denotes the *cyclomatic number* of G and $\mathrm{cc}(G)$ the number of *cyclic components* in G.

To simplify the proof of this lemma, we will start by processing the generated graph G_s and stash after inserting all keys $x \in S$. Recall that we have seen in Section 3.4 that whenever a component C of G is decomposed into more than one component in G_s, these components are unicyclic and form a partition of C.

Lemma 4.2
Let $G = G(S, h_1, h_2)$ be the cuckoo graph and G_s the cuckoo graph without stash keys. Let C denote a cyclic component of G that has been decomposed into k unicyclic components $C'_1, \ldots, C'_k$ in G_s.

Then there exists a unicyclic subgraph C' of G with the same vertices as C and the same number of edges as $C'_1 \cup \ldots \cup C'_k$.

Proof. Proof by induction on the number of components $C'_1, \ldots, C'_k$. If $k = 1$, then $C' = C'_1$.

Assume that the lemma holds for $k-1$ components and consider k components $C'_1, \ldots, C'_k$. If C is decomposed into these components, there must exist an edge in the stash connecting C'_k with a component $C'_j, j < k$, because C is connected. Both of these components are unicyclic. Remove a cycle edge from C'_k and put it into the stash. Insert the edge connecting C'_k and C'_j into the graph. The joined component is still unicyclic, because we joined a unicyclic component with a tree. Furthermore, we did not change the number of edges in the graph. Now we are in the situation of $k-1$ components and the lemma follows by induction. □

Obviously, we can find such a unicyclic subgraph C' associated with a component C for every component in $G(S, h_1, h_2)$ and are able to preserve the stash size. Furthermore, these subgraphs are still suitable for cuckoo hashing, because each subgraph is a tree or unicyclic. Hence we can state the following corollary.

Corollary 4.3
Let $G = G(S, h_1, h_2)$ be the cuckoo graph with k components and G_s the cuckoo graph without l stash keys.

Then there exists a subgraph G' of G with k components that contain at most one cycle and G' contains $|E(G)| - l$ edges.

Now we can easily prove Lemma 4.1.

Proof. Using Corollary 4.3, we can assume without loss of generality that G_s and G have the same number of components and share the same vertices in each component. Furthermore, every component in G_s is a tree or unicyclic.

We order these components in such a way that for every component C in G the associated component in G_s is denoted by C' and prove that the number of elements in the stash for every component C is exactly $\mathrm{ex}(C)$. Then Lemma 4.1 follows directly by

$$\mathrm{ex}(G) = \sum_{C \in \mathfrak{C}(G)} \mathrm{ex}(C).$$

If C is a tree or unicyclic, it does not contain a bad edge set and hence $C' = C$ and $\mathrm{ex}(C) = \gamma(C) - \mathrm{cc}(C) = 0$.

Assume that C has at least two cycles. Removing one cycle edge in C' yields a spanning tree T, because C' is unicyclic. As T is also a spanning tree of C, the number of edges that are missing in C' is exactly $\gamma(C) - 1 = \mathrm{ex}(C)$. □

In summary it can be said that the size of the stash can be expressed by the excess of the cuckoo graph. We consider the size of the stash to be a random variable and we have to take a closer look at the probability that this random variable exceeds some constant value.

4.2. Probabilistic View on the Stash Size

In this section we consider the probability that the stash size exceeds a specific constant value after all keys have been inserted into the data structure. The methodology is closely connected to the work of Dietzfelbinger and Woelfel in [DW03]. In the first part we obtain an upper bound for the number of non-isomorphic graphs with a given number of edges, connected components and a given excess. Afterwards, we consider the probability that a specific bipartite graph is isomorphic to one of these graphs, which will allow us to express the probability that the stash size exceeds a given value. Let us start by recalling a result of Dietzfelbinger and Woelfel.

Definition 4.4
Let $N(k, l, q)$ denote the number of non-isomorphic graphs whose cyclomatic number is q and which have $k - l$ inner edges and l leaf edges.

We are interested in estimating $N(k, l, q)$.

Lemma 4.5

1. $N(k, l, 0) \leq k^{2l-4}$
2. $N(k, l, q) \leq 1/2 \cdot k^{2q} \cdot (k - q)^{2l+4q-4}$

Proof.

1. Any tree with $k - l$ inner edges and $l \geq 2$ leaf edges can be constructed in the following way. We start with a path of length $k_2 \in [2, k - (l - 2)]$, called T_2. For $i = 3, \ldots, l$ one constructs T_i from T_{i-1} by taking a new path of length $k_i \geq 1$ such that $k_2 + \ldots + k_i \leq k - (l - i)$ and identifying one endpoint of this path with an arbitrary non-leaf vertex in T_{i-1}. Clearly, the length k_l of the last path is uniquely determined by $k_l = k - (k_2 + \ldots + k_{l-1})$. Thus we have to choose $l - 2$ lengths for our paths. For each of these path lengths we have fewer than k choices and thus there are fewer than k^{l-2} choices for the $l - 2$ paths. For each of these choices, there are fewer than k possible positions for the endpoint of our

path, because a tree with k edges contains $k+1$ vertices and there are at least two leaf vertices. This yields

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

2. We can evaluate $N(k,l,q)$ using $N(k,l,0)$. A graph with cyclomatic number q can be transformed into an acyclic graph by removing q cycle edges. After the deletion of these edges, the adjacent cycle edges might become leaf edges. We have $\binom{k}{2}^q$ choices for the endpoints of the removed edge, because a graph with cyclomatic number $q > 0$ and k edges has exactly $k - q + 1 \leq k$ vertices. Hence we can bound $N(k,l,q)$ as follows:

$$\begin{aligned} N(k,l,q) &= \binom{k}{2}^q \cdot N(k-q, l+2q, 0) \\ &\leq \left(\frac{k^2}{2}\right)^q \cdot N(k-q, l+2q, 0) \\ &\leq 1/2 \cdot k^{2q} \cdot N(k-q, l+2q, 0) \\ &\leq 1/2 \cdot k^{2q} \cdot (k-q)^{(2l+4q)-4}. \end{aligned}$$

□

We can extend the definition of $N(k,l,q)$ in the following way:

Definition 4.6
Let $N(k,l,c,s)$ denote the number of non-isomorphic graphs with $k-l$ inner edges, l leaf edges, c connected components and excess s.

The excess of the graph describes the size of the stash and is of interest if we want to analyze stash-based cuckoo hashing. We obtain the following upper bounds.

Lemma 4.7

1. $N(k,l,1,s) < (k-s)^{2l+6s+2}$
2. $N(k,l,c,s) < (k+c-s)^{2l+6s+8c-6}$

Proof.

1. $N(k,l,1,s)$ denotes the number of non-isomorphic connected graphs with $k-l$ inner edges, l leaf edges and excess s. Using Lemma 4.5, we can express $N(k,l,1,0)$ as follows:

$$\begin{aligned} N(k,l,1,0) &= N(k,l,0) + N(k,l,1) \\ &\leq k^{2l-4} + 1/2 \cdot k^2(k-1)^{2l} \\ &< k^{2l-4} + 1/2 \cdot k^{2l+2} \\ &= k^{2l+2}(k^{-6} + 1/2) \\ &\leq k^{2l+2}. \end{aligned} \tag{1}$$

   Moreover, note that we can delete s cycle edges from a graph with $s \geq 1$ excess edges to obtain a unicyclic graph. For choosing the endpoints of these edges, we have fewer than $(k-s)^{2s}$ possibilities ($(k-s)^2$ for each edge, because a connected graph with excess $s \geq 1$ has cyclomatic number $s+1$ and thus $k-s$ vertices). As each of the removed edges are on a cycle and the adjacent edges might become leaf edges, we gain at most $2s$ leaf edges. Thus, by Equation 1

$$\begin{aligned} N(k,l,1,s) &\leq (k-s)^{2s} \cdot N(k-s, l+2s, 1, 0) \\ &< (k-s)^{2s} \cdot (k-s)^{2l+4s+2} \\ &= (k-s)^{2l+6s+2}. \end{aligned}$$

2. We can express the difference between a connected graph and a graph with $c > 1$ connected components as follows. If we take a look at $N(k,l,2,s)$ then this is the number of non-isomorphic graphs with $k-l$ inner edges, l leaf edges, two connected components and excess s. If we insert an edge between these two components, the two components will be joined and the graph is connected. In this case, the number of edges will be incremented by one and the new edge is an inner edge. The new edge increases the excess of the graph by one, if it connects two cyclic components as in Figure 11 on page 21.

   The same aspects can be extended to the scenario of multiple connected components. In this case we join c connected components and thus the resulting graph G has excess between s and $s+c-1$. Prior to the inserting of these edges, the graph has k edges and excess s. Thus, we know by

$$\begin{aligned} \mathrm{ex}(G) &\leq \gamma(G) \\ &= m - n + c \\ &= k + c - n \end{aligned}$$

   that the graph contains at most $k+c-s$ vertices. There are fewer than $(k+c-s)^2$ choices for the endpoints of each edge being inserted into the graph. We obtain

the following upper bound for $N(k,l,c,s)$:

$$\begin{aligned} N(k,l,c,s) &\leq (k+c-s)^{2(c-1)} \cdot N(k+c-1,l,1,s+c-1) \\ &< (k+c-s)^{2c-2} \cdot (k+c-s-1)^{2l+6(s+c-1)+2} \\ &= (k+c-s)^{2c-2} \cdot (k+c-s-1)^{2l+6s+6c-6+2} \\ &< (k+c-s)^{2l+6s+8c-6}. \end{aligned}$$

□

With the knowledge of an upper bound of non-isomorphic graphs with a given number of inner resp. leaf edges, connected components and a given excess, we are now interested in the probability that a random bipartite graph is isomorphic to one of these graphs.

Let $K(T) = G(T, h_1, h_2), T \subseteq S$, denote the subgraph of G consisting of all edges for keys $x \in T$, disregarding isolated vertices.

Lemma 4.8
Let $T \subseteq U$, and $H = (V_H, E_H)$ be a bipartite graph, whose edges are uniquely labeled with the elements of T. If the values $h_i(x)$ are chosen fully randomly for all $x \in T, i \in \{1,2\}$, then the probability that $K(T)$ is isomorphic to H is

$$2^c \cdot m^{-|E_H|-\gamma(H)+c},$$

where c denotes the number of connected components of H.

Proof. Let $d_H(v) = d_v$. Consider a vertex v in H that is incident to d_v edges $x_1, \ldots, x_{d_v}$. If H is isomorphic to $K(T)$ then there exists $h_i(x), i \in \{1,2\}$, such that $h_i(x_1) = \ldots = h_i(x_{d_v})$. As we assume fully random hash functions, the probability for these $d_v - 1$ collisions is m^{-d_v+1}.

Let H' be a component of H. If H is bipartite then there is exactly one way to split $V_{H'} = V_{L'} \cup V_{R'}$, such that all vertices in L' are colored with one color and all vertices in R' with the other and all edges are incident to both of these colors. The probability that $K(T)$ is isomorphic to H is bounded by the probability that there exists a coloring in each connected component such that if v is colored with color i and v is incident to $x_1, \ldots, x_{d_v}$, then $h_i(x_1) = \ldots = h_i(x_{d_v})$. Clearly, $K(T)$ is isomorphic to H if all connected components are isomorphic. We can color the vertices in H in 2^c different ways, where c is the number of connected components in H, because every connected component can be colored in two different ways.

We will need the following two results for the analysis. In section 2.2 we proved that $\sum_{v \in V_H} d_v = 2|E_H|$ and $\gamma(H) = |E_H| - |V_H| + c$.

It follows

$$\begin{aligned}\Pr(K(T) \text{ is isomorphic to } H) &= 2^c \cdot \prod_{v \in V_H} m^{-d_v+1} \\ &= 2^c \cdot m^{\sum_{v \in V_H} -d_v+1} \\ &= 2^c \cdot m^{|V_H|-2|E_H|} \\ &= 2^c \cdot m^{|E_H|-\gamma(H)+c-2|E_H|} \\ &= 2^c \cdot m^{-|E_H|-\gamma(H)+c}.\end{aligned}$$

□

As we are interested to bound the probability that the cuckoo graph has excess at least $\hat{s}+1$ if the maximum stash size is $\hat{s}$, one should focus on a minimal subgraph that induces this excess[7].

Definition 4.9
An *excess-s core structure* of $G = G(S, h_1, h_2)$ is a subgraph G' of G with the following properties:

1. G' has excess exactly s.
2. G' has no leaf edges.
3. G' contains only components with at least two cycles.

Note that we are only interested in excess-s core structures for $s \geq 1$, because even if we do not incorporate a stash, cuckoo hashing would fail because of an excess-1 core structure in the graph. Moreover, our stash should be of small size. Thus, we assume that the excess is constant.

Lemma 4.10
Let $G = G(S, h_1, h_2)$ be the cuckoo graph. Let G have excess at least s. Then G contains an excess-s core structure.

Proof. We already know that we can decrease the excess by looking at a component C of G that contains more than one cycle. While $\text{ex}(C) > 0$, removing a cycle edge from C will decrease the excess by one. Thus we can obtain a graph with excess exactly s. The second property – a subgraph with no leaf edges – is easy to acquire by repeatedly removing leaf edges (while there are any left). Clearly, removing leaf edges preserves connectivity and does not decrease excess. All components in the remaining graph are at least unicyclic, because repeatedly removing leaf edges in a tree will remove all edges of the tree. Since a unicyclic component has excess 0, we can remove unicyclic components without changing the excess. □

[7] If the cuckoo graph has excess $\hat{s}+1$, more than $\hat{s}$ keys must be placed into the stash and a rehash occurs.

An example for an excess-s core structure of a cuckoo graph is provided in Section A.2, Figure 26 on page 81.

We can now proceed to the main theorem of this section, which is equivalent to Theorem 2.1 in [KMW08].

Theorem 4.11
Let $G = G(S, h_1, h_2)$ be the random cuckoo graph on key set $S \subseteq U$ and fully random hash functions h_1, h_2 from U to $[m]$, where $|S| = n$ and $m = (1+\varepsilon)n$ for an $\varepsilon \in (0, 1)$. Let $\mathrm{ex}(G)$ denote the excess of G. Then

$$\Pr(\mathrm{ex}(G) \geq s) = \mathcal{O}(n^{-s}).$$

We start with a short discussion. The presented theorem is the main result of our analysis. It states that we can improve the failure probability of cuckoo hashing on a given key set from $\mathcal{O}(n^{-1})$ to $\mathcal{O}(n^{-\hat{s}-1})$ by using a constant memory with $\hat{s}$ places and the described insertion algorithm, because the probability that more than $\hat{s}$ edges are placed into the stash is $\Pr(\mathrm{ex}(G) > \hat{s}) \leq \mathcal{O}(n^{-\hat{s}-1})$.

Proof. Let $G = G(S, h_1, h_2)$ and $\mathrm{ex}(G) \geq s$. Then G contains a subgraph $K(T), T \subseteq S$, of size k that forms an excess-s core structure. We already know that there are at most $N(k, 0, c, s)$ non-isomorphic graphs that form such an excess core structure. There are $\binom{n}{k}$ ways for choosing T and $k!$ ways to label a graph with k edges uniquely with the elements from T, hence there are at most $\binom{n}{k}k! \cdot N(k, 0, c, s) \leq n^k \cdot N(k, 0, c, s)$ non-isomorphic edge-labeled graphs for a given k. Using Lemma 4.8, the probability that a random bipartite multigraph H with k edges uniquely labeled with the keys from T is isomorphic to $K(T)$ is bounded by $2^c \cdot m^{-|E_H|-\gamma(H)+c}$ and $s = \gamma(H) - c$ if H forms an excess core structure (because all components in H are cyclic), yielding $2^c \cdot m^{-|E_H|-s}$. Furthermore, a graph with excess $s \geq 1$ contains at least $s+2$ edges. Every component of an excess core structure has at least one excess edge, thus the number of connected components is bounded by s.

Putting together all these observations, we can estimate the probability for the event that a random cuckoo graph G has excess at least s as follows.

$$
\begin{aligned}
\Pr(\mathrm{ex}(G) \geq s) &\leq \sum_{k=s+2}^{n} \sum_{c=1}^{s} \frac{2^c \cdot n^k \cdot N(k,0,c,s)}{m^{k+s}} \\
&< \sum_{k=s+2}^{n} \sum_{c=1}^{s} \frac{2^c \cdot n^k \cdot (k+c-s)^{6s+8c-6}}{m^{k+s}} \\
&= \sum_{k=s+2}^{n} \sum_{c=1}^{s} \frac{2^c \cdot n^k \cdot (k+c-s)^{6s+8c-6}}{((1+\varepsilon)n)^{k+s}} \\
&= \frac{1}{n^s} \sum_{k=s+2}^{n} \sum_{c=1}^{s} \frac{2^c \cdot (k+c-s)^{6s+8c-6}}{(1+\varepsilon)^{k+s}} \\
&\leq \frac{s2^s}{n^s} \sum_{k=s+2}^{n} \frac{(k+s-s)^{6s+8s-6}}{(1+\varepsilon)^{k+s}} \\
&= \frac{s2^s}{n^s} \sum_{k=s+2}^{n} \frac{k^{\mathcal{O}(1)}}{(1+\varepsilon)^{k+s}} \\
&\leq \frac{\mathcal{O}(1)}{n^s} \sum_{k=s+2}^{n} \frac{k^{\mathcal{O}(1)}}{(1+\varepsilon)^{k+s}} \\
&= \mathcal{O}(n^{-s})
\end{aligned}
$$

□

Using this result, we can directly state the failure probability of the insertion procedure in the stash-based scenario.

Corollary 4.12
Let $S \subseteq U$ with $|S| = n$. Let $m = (1+\varepsilon)n$ for an arbitrary constant $\varepsilon \in (0,1)$ and let the stash contain $\hat{s}$ memory cells to store keys.

Then calling the insertion procedure for all keys in S fails and triggers a rehash with probability $\mathcal{O}(n^{-\hat{s}-1})$.

Proof. If the stash can hold $\hat{s}$ items, a rehash event occurs if the cuckoo graph has an excess of at least $\hat{s}+1$. The probability for this is bounded by $\mathcal{O}(n^{-\hat{s}-1})$ and the corollary follows. □

With the analysis conducted during the last two sections, we can describe the size of the stash and provide an upper bound on the probability that the insertion of a key set yields a stash size of $\hat{s}$ or more. We have seen that the size of the stash is described by a property of the cuckoo graph $G(S, h_1, h_2)$ – its excess. The probability that a cuckoo graph has excess $\hat{s}+1$ or more is bounded by a polynomially small term $\mathcal{O}(n^{-\hat{s}-1})$, which dramatically reduces the failure probability of $\mathcal{O}(n^{-1})$ in standard cuckoo hashing. Our analysis provides an alternative approach to prove the impact on

the failure probability of the insertion algorithm when using a stash in cuckoo hashing. In contrary to the methodology used by Kirsch et al. in [KMW08] and Kutzelnigg in [Kut09], we base our analysis on a counting argument and graph isomorphism and one should be able to follow our reasoning without much knowledge in the problem domain. In contrast to the result of our approach, Kutzelnigg computed the constant in the $\mathcal{O}(n^{-\hat{s}-1})$ expression by using generating functions and the saddle point method, which is clearly an advantage of his method.

With the results obtained in the previous sections, we are now able to provide time bounds on the insert and delete operation. We will start with the analysis of the insertion algorithm.

4.3. Analysis of Insertions in Cuckoo Hashing with a Stash

We have seen in the preliminaries section that the insertion of a single element in cuckoo hashing is expected to run in amortized constant time. Conducting this analysis in the stash scenario adds a new condition. We have analyzed the probability that the insertion of a key x yields t evictions in the data structure. In the new scenario, we have to examine the same situation, but take into account that some keys have been put into the stash prior to the insertion of key x.

The last section told us that the stash size equals $\mathrm{ex}(G(S, h_1, h_2))$ using an idealized variant of the insertion algorithm. In practical scenarios, we often restrict the number of evictions during a single insertion to $\Theta(\log n)$, adding a small failure probability that a key that actually can be included is moved to the stash. In the stash-based scenario, we have to take into account that such insertion failures could happen multiple times. Obviously, it would be great to assume that such failures occur independently. However, this is not the case.

Let us focus on the small schematic example presented in Figure 16, consisting of one path of length exceeding the number *MaxLoop*. Now if keys x and y are inserted in this order, this path would cause two keys being moved to the stash. The example similarly holds for more than two keys.

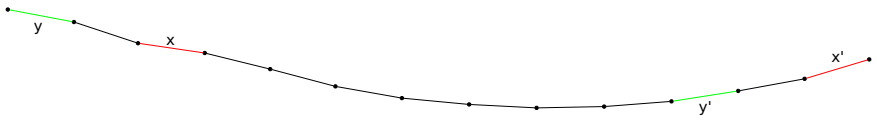


Figure 16: Insertions of two keys x and y. Edges colored in the same way as the key edges denote edges that would be moved to the stash, because the eviction took more than *MaxLoop* steps.

The insertion of the key x causes x' being moved to the stash. Afterwards, the insertion of a key y moves a key y' to the stash. The existence of one rather large structure can thus yield many failing keys. We can conclude that such failures are not

independent and it is hard to bound the probability of multiple insertions failing in this way.

Recall that s denotes the current stash load and $\hat{s}$ the maximal stash size. We consider s as a random variable. We can reuse Lemma 3.6 if the stash is empty and thus

$$\begin{aligned}\Pr(\#\text{evictions}_x \geq t \mid s = 0) &\leq \Pr(\#\text{evictions}_x \geq t) \\ &\leq 2(1+\varepsilon)^{-\lceil \frac{t-1}{3} \rceil}.\end{aligned}$$

Knowing this, we can recall our results for the value *MaxLoop*. Setting *MaxLoop* equal to $c\lceil \log_{1+\varepsilon} n \rceil + 1$ yields a failure probability of $2n^{-c/3}$. We are interested in $\Pr(s \leq \hat{s})$. If we set $\mathit{MaxLoop} = 3(\hat{s}+2)\lceil \log_{1+\varepsilon} n \rceil + 1$, the probability that there was one key moved to the stash because of a too long eviction chain is bounded by $n \cdot \mathcal{O}(n^{-\hat{s}-2}) = \mathcal{O}(n^{-\hat{s}-1})$ and thus the failure probability is dominated by the excess of the graph. We will hence focus on the case that all stash keys are put into the stash because bad edge sets occurred. We can analyze the general case as follows.

Lemma 4.13
Assume that a key x can be inserted successfully in the hash table and let $\#\text{evictions}_x$ denote the number of evictions in the table until the insertion of x succeeded. Then

$$\Pr(\#\text{evictions}_x \geq t \mid s \leq \hat{s}) \leq 2(1+\varepsilon)^{-\lceil \frac{t-1}{3} \rceil}(1 + \mathcal{O}(n^{-\hat{s}-1})).$$

Proof.

$$\begin{aligned}\Pr(\#\text{evictions}_x \geq t \mid s \leq \hat{s}) &= \frac{\Pr(\#\text{evictions}_x \geq t \cap s \leq \hat{s})}{\Pr(s \leq \hat{s})} \\ &\leq \frac{\Pr(\#\text{evictions}_x \geq t)}{\Pr(s \leq \hat{s})}.\end{aligned}$$

We can use the same argument as in Lemma 3.6. If a key x evicts more than t keys then there exists a simple path of length at least $\lceil \frac{t-1}{3} \rceil$ starting at $h_1(x)$ or $h_2(x)$, which is not influenced by the stash. It follows

$$\begin{aligned}\Pr(\#\text{evictions}_x \geq t \mid s \leq \hat{s}) &\leq \frac{2(1+\varepsilon)^{-\lceil \frac{t-1}{3} \rceil}}{1 - \mathcal{O}(n^{-\hat{s}-1})} \\ &= 2(1+\varepsilon)^{-\lceil \frac{t-1}{3} \rceil}(1 + \mathcal{O}(n^{-\hat{s}-1})).\end{aligned}$$

□

What is the expected time of a successful key insertion?

Lemma 4.14
Let x be a key that can be successfully inserted into the hash table.

Then the expected insertion time of x is constant under the condition that the stash holds at most $\hat{s}$ elements.

Proof.

$$\begin{aligned}
\mathrm{E}(\#\text{evictions}_x \mid s \leq \hat{s}) &= \sum_{t \geq 1} \Pr(\#\text{evictions}_x \geq t \mid s \leq \hat{s}) \\
&\leq \sum_{t \geq 1} 2(1+\varepsilon)^{-\lceil \frac{t-1}{3} \rceil}(1 + \mathcal{O}(n^{-\hat{s}-1})) \\
&= 2(1 + \mathcal{O}(n^{-\hat{s}-1})) \sum_{t \geq 1} (1+\varepsilon)^{-\lceil \frac{t-1}{3} \rceil} \\
&= \mathcal{O}(1).
\end{aligned}$$

□

Hence, we expect that the insertion algorithm inserts a key in constant time, if the key can be inserted without triggering a rehash or moving a key into the stash.

Note that the previous results match the results for standard cuckoo hashing. As rehashes are more unlikely in the stash-based scenario, we formulate the result on the amortized expected running time of the insert operation as a corollary. The line of argument in Theorem 3.8 still holds, using the mentioned value for *MaxLoop* and the fact that the probability for a stash insertion is bounded by $\mathcal{O}(n^{-1})$ and thus contributes $\mathcal{O}(\mathit{MaxLoop} \cdot n^{-1}) = o(1)$ to the expected running time of an insertion in this case.

Corollary 4.15
Assume that (h_1, h_2) behaves fully randomly, let $m = (1+\varepsilon)n$ and let $\hat{s}$ denote the maximum stash size. Set $\mathit{MaxLoop} = 3(\hat{s}+2)\lceil \log_{1+\varepsilon} n \rceil + 1$.

Then insertions can be carried out in amortized expected constant time.

The lookup and insert operations provide the same performance in the stash-based cuckoo hashing scenario as observed in standard cuckoo hashing. The missing operation is the delete operation. In Section 3.4 we observed that a naïve implementation of a deletion algorithm might change the size of the stash if we insert and remove a key x in contrary to x not being inserted at all. For $G = G(S, h_1, h_2)$, this yields problems with our analysis in the previous sections, because the size of the stash does not equal $\mathrm{ex}(G)$ anymore.

The next section will propose possible variants of the delete operation.

4.4. Analysis of Deletions in Cuckoo Hashing with a Stash

The main problem of the delete operation is the necessity to check if a stash key x can be included into the hash table again after a key y is deleted.

A key x is a candidate for reinsertion into the hash table, if removing the edge labeled y yields an acyclic component and x has at least one endpoint incident to a vertex in this component[8].

One way to check if a stash key x can be stored in the hash table after the deletion of a key y works as follows. Whenever the delete operation is invoked, we try to insert all keys before deleting y. We can observe an interesting fact in the sequence of nestless keys.

If y becomes nestless during the insertion of a stash key, we just abort and are done, because y is now deleted and a stash key is inserted in the hash table. If a stash key cannot be included, the insertion will just fail. If all insertions failed, we just remove y in the naïve way and leave the stash unmodified. We use an idealized insertion algorithm that aborts if a key is moved for a third time, because this is an indicator for a loop, as pointed out before. We have seen such algorithms in Chapter 3 and observed their expected constant running time. Thus, the expected time to obtain a possible key x that can be included successfully after y was removed is at most $\hat{s} \cdot \mathcal{O}(1)$. It is also important to look at the worst-case running time.

Clearly, the worst-case size of a component is not constant and the worst-case running time of the insertion trial is not constant in such a case. Thus, the delete operation would not preserve its constant worst-case running time using this approach. Is there a way to preserve it?

The solution is to delay the insertion of the stash keys. If the delete operation removes a non-stash key, it sets a flag that notifies the insertion algorithm to start the next insertion of a key with the insertion of all keys in the stash. This would preserve the worst-case constant running time of the delete operation, but has impact on the running time of the insertion algorithm. How can we estimate this impact?

If a key x is in the stash and we try to insert this key into the data structure, we know that a key has to be deleted in the component of x before x can be inserted successfully in the hash table. We know that the expected running time of a successful insertion is constant. Otherwise, the insertion will run for *MaxLoop* steps and put a key into the stash again. How likely is it that some random key is deleted from the hash table and a stash key can be included again?

We know that the expected size of the component of a stash key is constant. Furthermore, the delete operation has $\mathcal{O}(n)$ keys to choose from and the hash functions threw all these keys uniformly and independently into the hash table. Thus it is unlikely that the deletion of a random key removes an edge inside the connected component of a stash key and we simplify our task by assuming that the insertion will run *MaxLoop* steps. Furthermore, note that the insertion of a stash key x can never trigger a rehash event, because in the worst case one key is put into the stash at the now free position of x (this could be x itself).

Set $\mathit{MaxLoop} = 3(\hat{s} + 2)\lceil \log_{1+\varepsilon} n \rceil + 1$. Let X be the random variable counting all

[8] x has been moved to the stash to lower the excess of the graph. One way of lowering this excess is decomposing a component with two cycles into two unicyclic components, removing an edge on the simple path connecting these components. If a cycle edge y is removed in one of these unicyclic components, x can be included into the data structure again and has only one endpoint in the acyclic component.

eviction steps during the insertion of all s stash elements. It follows that

$$\begin{aligned}
\mathrm{E}(X) &= \sum_{i=1}^{\hat{s}} \mathrm{E}(X \mid s = i) \cdot \Pr(s = i) \\
&\leq \sum_{i=1}^{\hat{s}} i \cdot \ \mathit{MaxLoop} \cdot \Pr(s \geq i) \\
&= \mathit{MaxLoop} \cdot \mathcal{O}\Big(\frac{1}{n} + \frac{2}{n^2} + \frac{3}{n^3} + \ldots + \frac{\hat{s}}{n^{\hat{s}}}\Big) \\
&= \mathcal{O}(\mathit{MaxLoop} \cdot n^{-1}) \\
&= o(1), \text{ for } \mathit{MaxLoop} \in \Theta(\log n).
\end{aligned}$$

Hence, the trial insertion of all stash elements runs in an expected time bounded away from 1, adding a "constant" overhead to the insertion algorithm. We refine this solution further.

Clearly, the worst event that could happen is that we delay the insertion for so long that a rehash event is unnecessarily triggered. We can postpone the insertion trial such that we avoid this event, but do not enforce a subsequent insert operation to insert all stash keys after delete operations.

One has to keep in mind a downside of a key unnecessarily residing in the stash. An unsuccessful lookup must look in this table cell as well, adding some constant to each unsuccessful lookup operation. Depending on the number of lookup invocations during the operation of the data structure, this might have influence on the performance of the whole data structure. We are going to use the following approach:

The delete operation sets a flag that notifies the insertion procedure if a key has been removed from the hash table. The insert operation will evaluate this flag only in the case that it could not add a key to the hash table and wants to push it into the stash. Before it moves the key to the stash, the insertion of all keys currently residing in the stash is performed – expecting constant overhead, as seen above. Using this approach we guarantee that the stash has minimal size after each unsuccessful insert operation. If lookups often fail during the operation of our data structure, we could also consider the following heuristic refinement.

If s keys are in the stash, we expect that we can insert a stash key after n/s deletions. We can count the number of delete operations and try to insert all stash keys with the next insertion after every n/s-th deletion. Every deletion preserves its constant running time using this approach and we insert stash keys although no insert operation takes place to improve the running time of lookup operations.

4.5. Conclusion

In this chapter we analyzed the behavior and performance of stash-based cuckoo hashing. We found a way to retrieve the stash size for a given graph $G(S, h_1, h_2)$ and further bounded the probability that a given maximal stash size will be exceeded.

Moreover, we provided an algorithm for the delete operation that solved the problems

we observed with a naïve implementation. The performance of the different operations in the stash-based scenario matches the performance of these operations in standard cuckoo hashing.

In the whole chapter, we assumed that fully random hash functions are to our free disposal and used the randomness property excessively. Can we achieve randomness with simpler hash functions – possibly only using d-wise independent hash functions for some constant d? This question will be solved in the next chapter.

5. Cuckoo Hashing with a Stash and Realistic Hash Functions

In the previous analysis of cuckoo hashing with a stash we always assumed that fully random hash functions, which map keys from a universe U to a range $[m]$ uniformly and independently are available for free. As we have seen in the preliminaries, such hash functions are unsuitable in practice. An interesting approach to hashing is universal hashing and, more specifically, d-wise independent hashing, because these hash functions can be efficiently evaluated and provide provable, but limited, independence.

As we have seen in the previous analysis, we can often identify core structures that own a given property if the corresponding graph owns this property, e.g., a graph $G = G(S, h_1, h_2)$ with excess at least s always contains a subgraph H with the property that H has excess exactly s, only connected components with at least two cycles and only inner edges. We called such a subgraph an excess-s core structure. If we could assume full randomness on such a core structure, we could conduct our analysis with such hash functions that work fully randomly on core structures but not necessarily on the whole graph.

The idea of our approach is to choose pairs (h_1, h_2) of hash functions from a family of hash functions $\mathcal{H}$ and decide, if the graph $G(S, h_1, h_2)$ induced by the key set S and the pair of hash functions is good or bad. Informally spoken, a graph is bad if there exists a key set $T \subseteq S$ where h_1 and h_2 are not working fully randomly and the subgraph $K(T)$ forms an excess core structure. On the other hand, if a graph is not bad for a given (h_1, h_2), we can assume that the hash functions map all keys uniformly and independently to their ranges for all such core structures.

Clearly, the question whether a graph is good or bad depends enormously on the used family of hash functions. We are interested in a family that yields a high probability for a graph to behave in the good way.

This chapter will introduce a candidate for such a hash family and we will thoroughly study its behavior.

5.1. The Hash Function Families $\mathcal{R}$ and $\hat{\mathcal{R}}$

Dietzfelbinger and Meyer auf der Heide introduced and studied a new class of hash functions in [DadH90]. Their so-called class $\mathcal{R}$ was designed for constant evaluation time and functions from this class behave in some aspects like truly random hash functions. The hash functions included in $\mathcal{R}$ are defined in the following way:

Definition 5.1
Let $\mathcal{H}_m^d$ denote a class of d-wise independent hash functions with range $[m]$. Let $d \geq 2$ and $r, m \in \mathbb{N}$. For $f \in \mathcal{H}_m^d, g \in \mathcal{H}_r^d$ and $z = (z_0, z_1, \ldots, z_{r-1}) \in [m]^r$ the hash function $h_{f,g,z} : U \to [m]$ is defined by $x \mapsto (f(x) + z_{g(x)}) \mod m$. The hash class $\mathcal{R}_{r,m}^d$ is the family of all functions $h_{f,g,z}$.

The idea behind this class of hash functions is that we can choose the values for the random vector z independently and only keys that collide under g are mapped to

the same element in this vector. Furthermore, the hash function f compensates for collisions of the hash function g and, if the degree of independence is chosen correctly, can yield almost random behavior of the hash functions.

Clearly, we could use pairs of hash functions from $\mathcal{R}^d_{r,m} \times \mathcal{R}^d_{r,m}$ to construct the cuckoo graph $G(S, h_1, h_2)$, but for our analysis it will be useful to modify this class of hash functions in the following way.

Definition 5.2 ***(Dietzfelbinger and Woelfel [DW03])***
The family of hash functions $\hat{\mathcal{R}}^d_{r,m}$ 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 $z^{(1)}, z^{(2)} \in [m]^r$.

Note that we share the g-component in (h_1, h_2). Dietzfelbinger and Woelfel showed in [DW03] that for $(h_1, h_2) \in \hat{\mathcal{R}}^d_{r,m}$ the hash functions behave with a high probability like fully random hash functions inside the connected components of the graph $G(S, h_1, h_2)$.

Obviously, we have to observe the behavior of this class of hash functions in our stash-based cuckoo hashing scenario, where the excess of the graph might be shared over more than one connected component. The methodology is closely connected to the previous work of Dietzfelbinger and Woelfel. We will start by stressing the way of achieving full randomness on a set of keys using hash functions from $\hat{\mathcal{R}}^d_{r,m}$.

5.2. Achieving Full Randomness with $\hat{\mathcal{R}}$

Before we start with definitions and the important results, one must understand a special property of hash functions from class $\hat{\mathcal{R}}$.

Consider a set of elements $T \subseteq U$ for which $|g(T)| = |T| - l$ holds, which means that g distributes the keys in T well in the sense of a bounded number of collisions. We will soon prove that there are at most $2l$ keys in T that collide under g. If we look at the z vector of a hash function in $\hat{\mathcal{R}}^d_{r,m}$ and choose the offsets of the indices with colliding keys in T under g randomly, the hash function $h_{f_i,g,z^{(i)}}$ still behaves randomly on these keys, if we choose f to be $2l$-wise independent. The other keys will then be mapped randomly because of the fully random offsets. f compensates for collisions in g and this is the key idea behind our construction – choose independent random offsets in z and use f to compensate for those keys that collide under g. We will now provide this result in a formal way.

Definition 5.3
Let $S \subseteq U$. For $T \subseteq S$, the set[9] $R^*(T)$ consists of those hash function pairs $(h_1, h_2) \in \hat{\mathcal{R}}^{2l}_{r,m}$ whose g-component satisfies $|g(T)| \geq |T| - l$.

[9] Note that $R^*(T)$ depends on l. We will omit this dependence, because we assume that l is fixed in the subsequent sections.

Theorem 5.4
Let $T \subseteq U$. Sample (h_1, h_2) from $R^*(T)$ randomly.

Then all $(h_1(x), h_2(x)), x \in T$, are uniformly and independently distributed in $[m]^2$.

Proof. If $(h_1, h_2) \in R^*(T)$, g distributes T rather well in $[m]$, because we required $|g(T)| \geq |T| - l$ and thus $|g(T)| = |T| - l'$ for some $l' \leq l$. We imagine the hash function g to throw the keys into buckets numbered 0 to $r - 1$. Clearly, there exist exactly $|T| - l'$ filled buckets and there are at most l' buckets containing more than one element. We identify the following subsets.

Let T_1 contain exactly one element of every occupied bucket and denote with T_1' the subset of those elements among T_1 that reside in buckets with at least two elements. Let T_2 denote the keys in $T - T_1$. Note that these keys collide under g and there are exactly l' many of them[10]. See Figure 17 for an example.

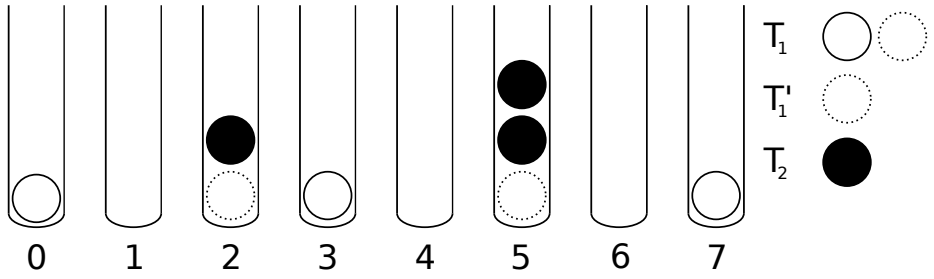


Figure 17: The sets T_1, T_1' and T_2 for $|T| = 8$ and $|g(T)| = 5$.

How are the keys in T_2 distributed over the buckets? In the worst case, every such key is residing in a bucket with exactly two elements. As the number of colliding keys is given by $|T_1'| + |T_2|$ and $|T_1'|$ equals $|T_2|$ in this case, we can estimate the total number of colliding keys to be at most $2l'$.

Recall that our hash functions were defined by $x \mapsto (f_i(x) + z^{(i)}_{g(x)}) \mod m$, where $z^{(i)}$ were random vectors containing r elements. We now choose $f^{(1)}$, $f^{(2)}$ and the random offsets for $z^{(1)}$, $z^{(2)}$ in a particular order.

We start by choosing random offsets for the indices $j \in g(T_2)$ of colliding keys in $z^{(1)}_j$ resp. $z^{(2)}_j$. Assume that these offsets are fixed. Observe that h_1 and h_2 distribute the keys $x \in T_1' \cup T_2$ uniformly and independently in $[m]$, because f_1 and f_2 are $2l$-wise independent. All keys in $T_1 - T_1'$ do not collide under g with another key in T and we can choose these offsets independently from each other. Thus, all keys in T are uniformly and independently distributed in $[m]$ for each of the hash functions h_1 and h_2. □

[10] We can imagine T_2 by removing exactly one element from every occupied bucket. The elements left in the buckets are the keys in T_2 and there are exactly $|T| - (|T| - l')$ many of them.

Furthermore, we are interested in how likely it is that $|g(T)| = |T| - l$ for a set $T \subseteq U$ of size k.

Lemma 5.5
Let $T \subseteq U$ with $|T| = k$. Then the probability that $|g(T)| = |T| - l$ is bounded by

$$\Big(\frac{k^2 l}{r}\Big)^l.$$

Proof. If $|g(T)| = |T| - l$, we can identify subsets T_1, T_1' and T_2 as above. T_2 contains exactly l elements and T_1' contains at most l elements and hence there are fewer than k^{2l} choices for the elements in T_1' and T_2. The elements in T_2 collide with every key in T_1', and, as g is $2l$-wise independent, the probability for these collisions is bounded by $(\frac{|T_1'|}{r})^{|T_2|} \leq (l/r)^l$. Thus, the probability of the event that $|g(T)| = |T| - l$ is bounded by $(k^2 l/r)^l$. □

Especially Theorem 5.4 will be the main result behind the line of argument in the next section. While $(h_1, h_2) \in \hat{\mathcal{R}}^{2l}_{r,m}$ and the g-component satisfies $|g(T)| \geq |T| - l$ for $T \subseteq U$, (h_1, h_2) behaves fully randomly on the keys in T. It remains to estimate how likely it is that a pair of hash function is a member of $R^*(T)$ for graph classes of our interest.

5.3. Hash Class $\hat{\mathcal{R}}$ in Cuckoo Hashing with a Stash

As indicated in the introduction to this chapter, we divide the cuckoo graphs induced by the hash functions chosen from $\hat{\mathcal{R}}^{d}_{r,m}$ for a given set S of keys into good and bad graphs. Let $\hat{s}$ be a constant denoting the maximum stash size. We call a graph *bad* if there exists a subset T of the key set, such that the subgraph $K(T) = G(T, h_1, h_2)$ forms an excess core structure with excess between 1 and $\hat{s} + 1$, and (h_1, h_2) does not distribute the keys from T uniformly and independently. We will only focus on excess core structures with an excess mentioned above[11] and thus omit the exact excess values in the following considerations.

Definition 5.6
Let $S \subseteq U$, and let $l \in \mathbb{N}$.

1. For $T \subseteq S$, the set $R^*(T)$ consists of those hash function pairs (h_1, h_2) whose g-component satisfies $|g(T)| \geq |T| - l$.
2. $G = G(S, h_1, h_2)$ is l-bad if there exists a subset $T \subseteq S$ such that $K(T)$ is an excess core structure and $(h_1, h_2) \notin R^*(T)$.
3. $R(S) \subseteq \hat{\mathcal{R}}^{2l}_{r,m}$ is the set of those hash function pairs (h_1, h_2) for which the graph $G(S, h_1, h_2)$ is not l-bad.

[11] We are especially interested in full randomness on excess-$(\hat{s}+1)$ core structures, as we need them to state the failure probability of stash-based cuckoo hashing.

As proven in the previous section, hash functions in $R^*(T)$ satisfy exactly our goal for the hash functions we want to use. If a hash function pair (h_1, h_2) is in $R^*(T)$, then (h_1, h_2) behaves fully randomly on the keys in T. From the definition it follows directly that a pair of good hash functions chosen from $R(S)$ is member of $R^*(T)$ for every excess core structure $T \subseteq S$.

Informally spoken, we can now choose a hash function from $\hat{\mathcal{R}}^{2l}_{r,m}$ and if this hash function is fortunately a member of $R(S)$ for our set S of keys, we can assume that the pair of hash functions works perfectly randomly on all those subsets T of S that yield an excess core structure. It remains to answer if it is likely to happen that we pick a hash function pair (h_1, h_2) from $R(S)$. We will proceed with the following basic idea.

If $(h_1, h_2) \in \hat{\mathcal{R}}^{2l}_{r,m}$ is not in $R(S)$, we know that the graph is l-bad and contains a subgraph $K(T)$ that forms an excess core structure and $|g(T)| < |T| - l$. We also know that (h_1, h_2) works fully randomly on key sets T' with $|g(T')| \geq |T'| - l$. Thus, removing keys from T in a particular way until it yields such a subset T' seems like an appropriate way to extract a subgraph where (h_1, h_2) works fully randomly, although $G(S, h_1, h_2)$ is l-bad.

We will see that we can extract subgraphs of l-bad graphs in such a way. On these structures, $(h_1, h_2) \notin R(S)$ will still work fully randomly and hence we can use the methodology from the previous chapter to count such structures and estimate the probability of the event that a graph is l-bad. We will first examine such subgraphs of l-bad graphs. As there are at most $2l$ keys that collide if $|g(T)| \geq |T| - l$, we will call such a subgraph $K(T)$ an $2l$-reduced subgraph.

Lemma 5.7
Let $G = G(S, h_1, h_2)$ with $(h_1, h_2) \in \hat{\mathcal{R}}^{2l}_{r,m}$. If G is l-bad, then there exists a subset $T \subseteq S$ such that $|g(T)| = |T| - l$ and $K(T)$ has the following properties:

1. There is one connected component in $K(T)$ that has at most $2l$ leaf and cycle edges.
2. All other connected components contain only inner edges.
3. There are at most $2l$ connected components.

Proof. If G is l-bad, there exists a subset $T \subseteq S$ with $|g(T)| < |T| - l$ and $K(T)$ forms an excess core structure. Fix such a T and mark the edge $(h_1(x), h_2(x))$ in $K(T)$ for $x \in T$, if there is another key $x' \in T, x \neq x'$, and $g(x) = g(x')$ (a key x' that collides with x under g).

We start by removing all connected components which do not contain a marked edge. Furthermore, we remove arbitrary connected components while $|g(T)| \leq |T| - l$ holds. Thus, removing another connected component yields $|g(T)| > |T| - l$ and from now on we focus on a connected component of $K(T)$. We choose an arbitrary connected component $K(T')$ in $K(T)$. Note that removing keys in T' will also delete these keys in T.

We start by iteratively removing edges from $K(T')$ while the remaining subset T of keys satisfies $|g(T)| \leq |T| - l$ and we do not destroy connectivity in $K(T')$.

When this process stops, all remaining leaf and cycle edges must be marked in this component, because otherwise we could remove such an edge without breaking our constraints on the process[12]. Furthermore, $|g(T)| = |T| - l$, because we could remove a marked edge otherwise. As we have seen before, there are at most $2l$ keys that collide under g if $|g(T)| = |T| - l$ and hence there are at most $2l$ leaf and cycle edges in $K(T')$. An example of this process can be found in Section A.4 on page 87.

Taking a look at the global structure of T tells us that $K(T')$ is the required component satisfying the first constraint and all other connected components contain only inner edges, for the connected components of excess core structures contain only inner edges to begin with. Moreover, $K(T)$ contains at most $2l$ connected components, because there are at most $2l$ marked edges and each component contains at least one. The lemma follows. □

As there is at most one component that contains leaf edges, we will call this component the *leaf component* of the $2l$-reduced subgraph. Furthermore, we can count such reduced subgraphs to estimate the probability that for a key set S and a hash function pair (h_1, h_2) from $\hat{\mathcal{R}}^{2l}_{r,m}$ the graph $G = G(S, h_1, h_2)$ is l-bad.

We recall that the value $N(k, l, c, s)$ denotes the number of non-isomorphic graphs that contain k edges, l leaf edges, c connected components and excess s. Furthermore, the probability that $K(T)$ is isomorphic to a bipartite random graph H uniquely labeled with the keys of T is $2^c \cdot m^{-|E_H|-\gamma(H)+c}$.

Using these facts, we can prove the following theorem:

Theorem 5.8
Let $\varepsilon \in (0, 1)$ and $l \geq 1$ be fixed. Let $m = (1 + \varepsilon)n$ for any set $S \subseteq U$ of size n. Let $R(S)$ be defined as above and $(h_1, h_2) \in \hat{\mathcal{R}}^{2l}_{r,m}$.
Then

$$\Pr((h_1, h_2) \in R(S)) \geq 1 - \mathcal{O}(n \cdot r^{-l}).$$

Proof. Note that $(h_1, h_2) \in R(S)$ is equivalent to $G(S, h_1, h_2)$ *not being* l*-bad.* We have identified $2l$-reduced subgraphs $K(T), |T| = k$, for l-bad graphs.

If $K(T)$ is an $2l$-reduced subgraph of an l-bad graph then $|g(T)| = |T| - l$. By Lemma 5.5 it follows that the probability for this is bounded by $(k^2 l/r)^l$.

Let H be a bipartite connected graph with k edges labeled with the keys of T, at most $2l$ connected components, a leaf component with at most $2l$ leaf and cycle edges, no other connected component with leaf edges and excess s of at most $\hat{s} + 1$. There are fewer than n^k ways to choose $T \subseteq S$ with $|T| = k$ and to label the edges of H in a unique way with elements from T. If $G = G(S, h_1, h_2)$ is l-bad, $K(T)$ is isomorphic to such a bipartite graph H and $|g(T)| = |T| - l$. If the last equation holds, (h_1, h_2) distributes T uniformly and randomly and thus we can reuse our observations regarding the probability that such graphs are isomorphic. H has at most one acyclic component and hence the excess s of H is at most $\gamma(H) - c + 1$ and $-|E_H| - \gamma(H) + c \leq -|E_H| - s + 1$.

[12] Removing a non-marked edge subtracts exactly one from both sides of the inequality $|g(T)| \leq |T| - l$.

Summing over all possible values of k we obtain the following upper bound on the probability that $G = G(S, h_1, h_2)$ is l-bad. Note that the estimated upper bounds (at most $2l$ leaf edges, at most excess $\hat{s}+1$, at most $2l$ connected components) yield a lot of summations in the inequality, but we can resolve them easily by estimating maximal values.

$$\begin{aligned}
\Pr(G \text{ is } l\text{-bad}) &\leq \sum_{k=2}^{n}\sum_{s=0}^{\hat{s}+1}\sum_{l_a=0}^{2l}\sum_{c=1}^{2l}\frac{2^c\cdot n^k\cdot (k^2l/r)^l\cdot N(k,l_a,c,s)}{m^{k+s-1}}\\
&< \sum_{k=2}^{n}\sum_{s=0}^{\hat{s}+1}\sum_{l_a=0}^{2l}\sum_{c=1}^{2l}\frac{2^c\cdot n^k\cdot (k^2l/r)^l\cdot (k+c-s)^{2l_a+6s+8c-6}}{m^{k+s-1}}\\
&\leq 2^{2l}2l\sum_{k=2}^{n}\sum_{s=0}^{\hat{s}+1}\sum_{l_a=0}^{2l}\frac{n^k\cdot (k^2l/r)^l\cdot (k+2l-s)^{2l_a+6s+16l-6}}{m^{k+s-1}}\\
&= \frac{1}{r^l}2^{2l}2l\cdot l^l\sum_{k=2}^{n}\sum_{s=0}^{\hat{s}+1}\sum_{l_a=0}^{2l}\frac{n^k\cdot k^{2l}\cdot (k+2l-s)^{2l_a+6s+16l-6}}{m^{k+s-1}}\\
&= \frac{1}{r^l}2^{2l+1}l^{l+1}\sum_{k=2}^{n}\sum_{s=0}^{\hat{s}+1}\sum_{l_a=0}^{2l}\frac{n^k\cdot k^{2l}\cdot (k+2l-s)^{2l_a+6s+16l-6}}{m^{k+s-1}}\\
&\leq \frac{1}{r^l}2^{2l+1}l^{l+1}(2l+1)\sum_{k=2}^{n}\sum_{s=0}^{\hat{s}+1}\frac{n^k\cdot k^{2l}\cdot (k+2l-s)^{20l+6s-6}}{m^{k+s-1}}\\
&\leq \frac{1}{r^l}2^{2l+1}l^{l+1}(2l+1)(\hat{s}+2)\sum_{k=2}^{n}\frac{n^k\cdot k^{2l}\cdot (k+2l)^{20l+6(\hat{s}+1)-6}}{m^{k-1}}\\
&= \frac{n}{r^l}2^{2l+1}l^{l+1}(2l+1)(\hat{s}+2)\sum_{k=2}^{n}\frac{k^{2l}\cdot (k+2l)^{20l+6\hat{s}}}{(1+\varepsilon)^{k-1}}\\
&< \frac{n}{r^l}2^{2l+1}l^{l+1}(2l+1)(\hat{s}+2)\sum_{k=2}^{n}\frac{(k+2l)^{22l+6\hat{s}}}{(1+\varepsilon)^{k-1}}\\
&\leq \frac{n}{r^l}2^{2l+1}l^{l+1}(2l+1)(\hat{s}+2)\sum_{k=2}^{n}\frac{(3k)^{22l+6\hat{s}}}{(1+\varepsilon)^{k-1}}\\
&= \frac{n}{r^l}2^{2l+1}l^{l+1}(2l+1)(\hat{s}+2)\sum_{k=2}^{n}\frac{(3k)^{\mathcal{O}(1)}}{(1+\varepsilon)^{k-1}}\\
&= \frac{n}{r^l}\mathcal{O}(1)\sum_{k=2}^{n}\frac{k^{\mathcal{O}(1)}}{(1+\varepsilon)^{k-1}}\\
&= \mathcal{O}\Big(\frac{n}{r^l}\Big).
\end{aligned}$$

□

The considered class of hash functions $\hat{\mathcal{R}}^{2l}_{r,m}$ provides us with an important property. Depending on how we choose the values l and r, specific hash functions from this class work fully randomly on subsets of the set S of keys from a universe U. The next theorem provides us with a specific way of choosing these two parameters.

Theorem 5.9
Let $S \subseteq U, |S| = n$, and $(h_1, h_2) \in \hat{\mathcal{R}}^{2l}_{r,m}$. Let $r = n^\beta, \frac{1}{2} < \beta < 1$ and $\hat{s}$ be constant.

If we choose $l = 2(\hat{s} + 2)$ then the probability that the excess of the graph $G(S, h_1, h_2)$ is greater than $\hat{s}$ is bounded by $\mathcal{O}(n^{-\hat{s}-1})$.

Proof. If $l = 2(\hat{s} + 2)$, then the probability that $G = G(S, h_1, h_2)$ is l-bad is bounded by $\mathcal{O}(n/n^{2(\hat{s}+2)\beta})$. For $\frac{1}{2} < \beta < 1$ this is less than $\mathcal{O}(1/n^{\hat{s}+1})$. If G is not l-bad, (h_1, h_2) works fully randomly on all subsets of S that yield an excess core structure for excess between 1 and $\hat{s}+1$. Following our reasoning in the proof of Theorem 4.11, every graph with excess at least $\hat{s}+1$ contains an excess-$(\hat{s}+1)$ core structure. Thus, if (h_1, h_2) induces a non-l-bad graph, we can express the probability that the excess is at least $\hat{s}+1$ like in the fully random case. We observed that this probability is bounded by $\mathcal{O}(n^{-\hat{s}-1})$. Note that this dominates the probability that $G(S, h_1, h_2)$ is l-bad. The theorem follows. □

Using this result we can directly state the failure probability of the idealized insertion procedure.

Corollary 5.10
Let $S \subseteq U, |S| = n$, and $(h_1, h_2) \in \hat{\mathcal{R}}^{2l}_{r,m}$. Let $r = n^\beta, \frac{1}{2} < \beta < 1$, $\hat{s}$ denote the maximum stash size and $l = 2(\hat{s} + 2)$. Let $m = (1 + \varepsilon)n$ for an arbitrary constant $\varepsilon \in (0, 1)$.

Then calling the insertion procedure for all keys in S fails and triggers a rehash with probability $\mathcal{O}(n^{-\hat{s}-1})$.

5.4. Conclusion

We proved that hash functions from the hash class $\hat{\mathcal{R}}^{2l}_{r,m}$ yield the same failure probability as fully random hash functions if the parameters r and l are chosen appropriately.

The reader might recall that this was not the only important result. We are clearly interested in the running time of the delete, insert and lookup operation when using hash functions from the class $\hat{\mathcal{R}}$. Our observations regarding this hash class so far focused on excess core structures and if one recalls the analysis of, e.g., the insert operation, we need a different result to conduct it successfully.

Dietzfelbinger and Woelfel discovered in [DW03] that hash functions from $\hat{\mathcal{R}}$ work with high probability fully randomly inside the connected components of a cuckoo graph. Furthermore, the upper bound they found matches the $\mathcal{O}(n \cdot r^{-l})$ bound proved for full randomness on excess core structures. Full randomness inside connected components is exactly the result we need to conduct the analysis of the insert and delete operation, for the insertion or deletion of a key is only of interest to the keys in its connected component.

In the next chapter, we will describe an abstract result on the randomness of hash functions from the class $\hat{\mathcal{R}}$. We will use it to prove full randomness on excess core structures and inside connected components.

6. A Generic Framework for Hash Class $\hat{\mathcal{R}}$

The present chapter is devoted to a more general result on the randomness properties of hash functions from the class $\hat{\mathcal{R}}^{2l}_{r,m}$ on arbitrary classes of bipartite multigraphs. Before getting down to the details, it seems appropriate to give a brief, intuitive and somewhat imprecise description of the approach we will use in this chapter.

As we have seen in the previous chapter, we try to obtain randomness properties of pairs of hash functions in $\hat{\mathcal{R}}^{2l}_{r,m}$, especially the likelihood of choosing a *good* pair of hash functions. We call a pair of hash functions good, if it works independently and uniformly on all "interesting" subsets of our key set S. Moreover, we will use the same methodology which was already applied during the previous chapter to obtain a reduced subgraph for an arbitrary graph and describe requirements on such reductions. On these subgraphs, even bad pairs of hash functions will work in the desired fully random way and we can count them to obtain an upper bound on the event that a randomly chosen pair of hash functions from $\hat{\mathcal{R}}^{2l}_{r,m}$ is bad. This chapter is based on an unpublished manuscript by Philipp Woelfel [Woe02].

6.1. The Framework

Let $\mathcal{G}_k$ denote the class of bipartite multigraphs (L, R, E) with k edges and both L and R copies of $[m]$. Let $\mathcal{G}$ denote the set of all these graph classes, formally

$$\mathcal{G} = \bigcup_{k=0}^{\infty} \mathcal{G}_k.$$

Let $\mathcal{G}^*$ denote the class of graphs in $\mathcal{G}$ where in each graph an arbitrary number of edges can be *marked*.

Definition 6.1
Let $l \in \mathbb{N}$ and $\delta : \mathbb{N} \to [0, 1]$. A class of graphs $\mathcal{D} \subseteq \mathcal{G}$ is $(2l, \delta)$-*reducible* if there exists a *reduction net* $\mathcal{C} \subseteq \mathcal{G}^*$ with the following properties:

1. If we mark arbitrary edges in $G \in \mathcal{D}$, then we can remove unmarked edges such that we obtain a graph in $\mathcal{C}$.
2. If $G \in \mathcal{C}$ contains marked edges, then we can remove one marked edge and possibly further unmarked edges such that we obtain a graph in $\mathcal{C}$.
3. If $G \in \mathcal{C}$ contains marked edges and we unmark an arbitrary marked edge, then we can remove unmarked edges such that we obtain a graph in $\mathcal{C}$.
4. The probability that in an independently and uniformly chosen graph with k edges we have to mark at most $2l$ edges such that the result is isomorphic to a graph in $\mathcal{C}$ is bounded by[13] $\delta(k)$.

We should step back and try to apply this definition to the analysis conducted in the last chapter. The set $\mathcal{D}$ contained all graphs with at most n edges that formed an excess core structure with excess between 1 and $\hat{s}+1$. The reduction net $\mathcal{C}$ contained all graphs with $k \leq n$ edges, one *leaf component* whose cycle and leaf edges are all marked, and all other components contained at least one marked edge and no leaf edges.

The first property is important to be able to actually reduce an arbitrary graph in $\mathcal{D}$ to the reduction net. The second and third property of the reduction are important to be able to remove edges until we obtain a graph on which even bad pairs of hash functions behave fully randomly. Note that Property 1 is used exactly once. Property 2 and 3 are used repeatedly until even a bad hash function pair works fully randomly.

The last property allows us to obtain the probability that a cuckoo graph $G(S, h_1, h_2)$ is "bad". As before, if the graph is bad we will extract some subset $T \subseteq S$ on which the hash functions behave fully randomly. We will argue that if $G(S, h_1, h_2)$ is bad, it suffices to mark at most $2l$ edges in $K(T)$ such that $K(T)$ is isomorphic to some graph in $\mathcal{C}$ and then reuse our approach of counting all such graphs and multiplying the result with the probability that $K(T)$ is isomorphic to them to obtain $\delta(k)$. Speaking in the language of the previous chapter, "marking at most $2l$ edges to obtain a graph isomorphic to a graph in $\mathcal{C}$" is in some sense equivalent to "$K(T)$ has at most $2l$ leaf and cycle edges in the leaf component, and at most $2l-1$ other connected components that contain only inner edges."

We will refine Definition 5.6 of the last chapter to define when we call a graph $G(S, h_1, h_2)$ l-bad in the new scenario.

Definition 6.2
Let $S \subseteq U$, $l \in \mathbb{N}$, and $\mathcal{D} \subseteq \mathcal{G}$.

1. For $T \subseteq S$, the set $R^*(T)$ consists of those hash function pairs $(h_1, h_2) \in \hat{\mathcal{R}}^{2l}_{r,m}$ whose g-component satisfies $|g(T)| \geq |T| - l$.
2. $G = G(S, h_1, h_2)$ is l-bad if there exists a subset $T \subseteq S$ such that $K(T) \in \mathcal{D}$ and $(h_1, h_2) \notin R^*(T)$.

We will now formulate the central theorem on the behavior of hash functions in $\hat{\mathcal{R}}^{2l}_{r,m}$ on $(2l, \delta)$-reducible graphs $\mathcal{D} \subseteq \mathcal{G}$. Recall that Theorem 5.4 already stated that hash functions in $R^*(T)$ behave fully randomly on T.

[13] Note that $\delta(k)$ will clearly depend on m, but as all of our observations consider a bipartite multigraph with copies of $[m]$ on both sides, we omit this dependency.

Theorem 6.3
Let $\mathcal{D} \subseteq \mathcal{G}$ be a $(2l, \delta)$-reducible class of multigraphs. Let $S \subseteq U$ denote the key set containing n keys.

Then the probability that a randomly chosen pair of hash functions (h_1, h_2) from $\hat{\mathcal{R}}^{2l}_{r,m}$ yields an l-bad graph $G(S, h_1, h_2)$ is at most

$$\left(\frac{l}{r}\right)^l \sum_{k=0}^{n} \binom{n}{k} k^{2l} \delta(k).$$

Proof. Let $\mathcal{D} \subseteq \mathcal{G}$ be $(2l, \delta)$-reducible.

If (h_1, h_2) yields an l-bad graph $G(S, h_1, h_2)$, there exists $T \subseteq S$ with $|g(T)| < |T| - l$ and $K(T) \in \mathcal{D}$. Now mark all edges in $K(T)$ that collide[14] under g. Clearly, $i > l$ edges are marked in this way.

As $\mathcal{D}$ is $(2l, \delta)$-reducible, there is a $T' \subseteq T$ which yields a subgraph $K(T') \in \mathcal{C}$ with i marked edges. Using Property 2 of $(2l, \delta)$-reducible graphs, we can iteratively remove marked edges (and might subsequently remove unmarked edges) and can always get back into $\mathcal{C}$. Note that removing a marked edge might unmark another edge automatically (if only two keys collide under g and one of these keys is removed), but Property 3 says that we can get back into $\mathcal{C}$ by removing further unmarked edges.

We remove edges until $|g(T')| = |T'| - l$. We have already seen that in this case at most $2l$ keys collide under g and the hash functions behave fully randomly on T'. Thus, we have to mark at most $2l$ edges in the random graph $K(T')$ to obtain a graph isomorphic to a graph in $\mathcal{C}$. The probability for this is bounded by $\delta(k)$.

Let $|T'| = k$. There are exactly $\binom{n}{k}$ ways to choose T' from S. Lemma 5.5 says that the probability of $|g(T')| = |T'| - l$ is bounded by $(k^2 l/r)^l$. Thus, we can obtain the following upper bound for the probability that a randomly chosen $(h_1, h_2) \in \hat{\mathcal{R}}^{2l}_{r,m}$ yields an l-bad graph:

$$(l/r)^l \sum_{k=0}^{n} \binom{n}{k} k^{2l} \delta(k).$$

□

This theorem arms us to state the following corollary.

Corollary 6.4
Let $C : \mathbb{N} \to \mathbb{R}$. If l is constant and

$$\delta(k) \leq C(n) \frac{k! \cdot k^{\mathcal{O}(1)}}{((1+\varepsilon)n)^k},$$

then $G(S, h_1, h_2), |S| = n$, is l-bad with probability $\mathcal{O}(C(n)/r^l)$.

[14] More formally, an edge x is marked if there exists a key $x' \in T$ such that $g(x) = g(x')$ and $x \neq x'$.

Proof. Let l be constant and $\delta(k)$ satisfy the inequality above. Then

$$\begin{aligned}
\Pr(G(S,h_1,h_2) \text{ is } l\text{-bad}) &\leq \Big(\frac{l}{r}\Big)^l \sum_{k=0}^{n} \binom{n}{k} k^{2l}\delta(k) \\
&\leq \Big(\frac{l}{r}\Big)^l \sum_{k=0}^{n} \binom{n}{k} k^{2l} C(n) \frac{k! k^{\mathcal{O}(1)}}{((1+\varepsilon)n)^k} \\
&\leq C(n)\Big(\frac{l}{r}\Big)^l \sum_{k=0}^{n} n^k k^{2l} \frac{k^{\mathcal{O}(1)}}{((1+\varepsilon)n)^k} \\
&\leq C(n)\Big(\frac{l}{r}\Big)^l \sum_{k=0}^{n} n^k \frac{k^{\mathcal{O}(1)}}{((1+\varepsilon)n)^k} \\
&\leq C(n)\Big(\frac{l}{r}\Big)^l \sum_{k=0}^{n} \frac{k^{\mathcal{O}(1)}}{(1+\varepsilon)^k} \\
&= \mathcal{O}\Big(\frac{C(n)}{r^l}\Big).
\end{aligned}$$

□

We are now ready to apply this framework to the case of full randomness on excess core structures and inside connected components.

6.2. Example 1: Full Randomness on Excess Core Structures

This section will apply the developed generic framework to prove the result obtained in the previous chapter one more time. Let $\hat{s}$ be the constant denoting the maximum stash size. Let $\mathcal{D}$ denote the class of multigraphs with at most n edges that form an excess core structure with excess between 1 and $\hat{s}+1$. If we consider the $(2l,\delta)$-reducibility of these graphs, we use the following reduction net.

Definition 6.5
Let $\mathcal{C}$ denote the class of marked multigraphs with at most n edges and the following properties:

1. There is one connected component that has only marked cycle and leaf edges — the leaf component.
2. All other connected components contain at least one marked edge and no leaf edges.

We will prove that $\mathcal{D}$ is $(2l,\delta)$-reducible. The important parts of the reduction step are already known from the previous chapter so that we can keep the proof short.

Theorem 6.6
Let $\mathcal{D}$ denote the class of multigraphs with at most n edges that form an excess core structure with excess between 1 and $\hat{s}+1$. Then $\mathcal{D}$ is $(2l, \delta)$-reducible.

Proof. We will use the reduction net $\mathcal{C}$ as given above. $\mathcal{D}$ and $\mathcal{C}$ have to provide the following properties:

1. If we mark arbitrary edges in $G \in \mathcal{D}$, then we can remove unmarked edges such that we obtain a graph in $\mathcal{C}$.
2. If $G \in \mathcal{C}$ contains marked edges, then we can remove one marked edge and possibly further unmarked edges such that we obtain a graph in $\mathcal{C}$.
3. If $G \in \mathcal{C}$ contains marked edges and we unmark an arbitrary marked edge, then we can remove unmarked edges such that we obtain a graph in $\mathcal{C}$.
4. The probability that in an independently and uniformly chosen graph with k edges we have to mark at most $2l$ edges such that the result is isomorphic to a graph in $\mathcal{C}$ is bounded by $\delta(k)$.

We now prove that each of these properties is provided:

1. Let $G \in \mathcal{D}$ and mark t arbitrary edges. Remove all unmarked components. Now choose an arbitrary component and iteratively remove unmarked leaf and cycle edges until all leaf and cycle edges are marked. This yields a graph in $\mathcal{C}$ with t marked edges.
2. Let $G \in \mathcal{C}$. Assume that $t > 0$ edges are marked. Remove an arbitrary marked leaf or cycle edge in the leaf component and iteratively remove unmarked leaf edges until the graph is once again in $\mathcal{C}$. It could also happen that we remove the leaf component in this way (if it contains only one marked edge) and in this case we choose an arbitrary connected component to become the new leaf component and remove unmarked leaf and cycle edges there.
3. Let $G \in \mathcal{C}$. Unmark an arbitrary marked edge. On the one hand, we could unmark a leaf or cycle edge in the leaf component. We can iteratively remove unmarked leaf and cycle edges until the graph is once again in $\mathcal{C}$. On the other hand, we could unmark an edge in another component. If this component does not contain a marked edge anymore, we just remove the whole component and the resulting graph is in $\mathcal{C}$.
4. We can reuse the value $N(k, l_a, c, s)$ and, as the graph is drawn uniformly and randomly, our observations regarding the probability that such a graph is isomorphic to a bipartite multigraph. Let $T \subseteq S, |T| = k$. Let H be a bipartite random graph that has at most $2l$ connected components, at most $2l$ leaf and cycle edges in the leaf component and excess at most $\hat{s}+1$. There are $k!$ ways to label H uniquely with the keys from T and the probability that it is isomorphic

to a randomly chosen graph with k edges labeled uniquely with the keys from T is $2^c \cdot m^{-k-\gamma(H)+c} \leq 2^c \cdot m^{-k-s+1}$, because H contains at most one acyclic component.

Putting all of these together yields[15]

$$\begin{aligned}\delta(k) &\leq \sum_{s=0}^{\hat{s}+1} \sum_{l_a=0}^{2l} \sum_{c=1}^{2l} k! \frac{2^c N(k, l_a, c, s)}{m^{k+s-1}} \\ &\leq \mathcal{O}(1) \cdot \frac{k! \cdot k^{\mathcal{O}(1)}}{((1+\varepsilon)n)^{k-1}} \\ &\leq \mathcal{O}(1) \cdot n \cdot \frac{k! \cdot k^{\mathcal{O}(1)}}{((1+\varepsilon)n)^{k}}.\end{aligned}$$

Thus, $\mathcal{D}$ is $(2l, \delta)$-reducible. □

Note that applying Corollary 6.4 yields $C(n) = n$ and we can state:

Corollary 6.7
Let $\mathcal{D}$ denote the set of all multigraphs with at most n edges that form an excess core structures with excess between 1 and $\hat{s}+1$. Then $(h_1, h_2) \in \hat{\mathcal{R}}^{2l}_{r,m}$ works fully randomly on a graph in $\mathcal{D}$ with probability $\mathcal{O}(n \cdot r^{-l})$.

This result was already known to us and we should focus on something new. We will take a look on the behavior inside connected components to conduct our analysis of the performance of the different operations in stash-based cuckoo hashing.

6.3. Example 2: Full Randomness on Connected Graphs

Let $\mathcal{D}$ denote the class of connected multigraphs with at most n edges. To establish the $(2l, \delta)$-reducibility of these graphs, we use the following reduction net.

Definition 6.8
Let $\mathcal{C}$ denote the class of all connected marked multigraphs with at most n edges whose leaf and cycle edges are all marked.

The attentive reader might immediately see the connection to the previous reduction net. As we are dealing with connected graphs now, we can reduce it to leaf components.

We will prove that $\mathcal{D}$ is $(2l, \delta)$-reducible.

Theorem 6.9
Let $\mathcal{D}$ denote the class of connected multigraphs with at most n edges. Then $\mathcal{D}$ is $(2l, \delta)$-reducible.

[15] Details of the calculation omitted, as they can be observed in the proof of Theorem 5.8.

Proof. We will use the reduction net $\mathcal{C}$ as given above. $\mathcal{D}$ and $\mathcal{C}$ have to provide the following properties:

1. If we mark arbitrary edges in $G \in \mathcal{D}$, then we can remove unmarked edges such that we obtain a graph in $\mathcal{C}$.
2. If $G \in \mathcal{C}$ contains marked edges, then we can remove one marked edge and possibly further unmarked edges such that we obtain a graph in $\mathcal{C}$.
3. If $G \in \mathcal{C}$ contains marked edges and we unmark an arbitrary marked edge, then we can remove unmarked edges such that we obtain a graph in $\mathcal{C}$.
4. The probability that in an independently and uniformly chosen graph with k edges we have to mark at most $2l$ edges such that the result is isomorphic to a graph in $\mathcal{C}$ is bounded by $\delta(k)$.

We now prove that each of these properties is provided:

1. Let $G \in \mathcal{D}$ and mark t arbitrary edges. Iteratively remove leaf and cycle edges until all leaf and cycle edges are marked. This yields a graph in $\mathcal{C}$ with t marked edges.
2. Let $G \in \mathcal{C}$. Assume that $t > 0$ edges are marked. Remove an arbitrary marked leaf or cycle edge and iteratively remove unmarked leaf edges until the graph is again in $\mathcal{C}$.
3. Let $G \in \mathcal{C}$. Unmark an arbitrary marked edge. If we unmark a leaf or cycle edge, we iteratively remove unmarked leaf and cycle edges until the graph is in $\mathcal{C}$.
4. We can reuse the value $N(k, l_a, 1, s)$ and our observations regarding graph isomorphism hold, as we sample the graph randomly. Let $T \subseteq S, |T| = k$. Let H be a bipartite random graph that has at most $2l$ leaf and cycle edges and excess at most $\hat{s} + 1$. There are $k!$ ways to label H uniquely with the keys from T and the probability that it is isomorphic to a randomly chosen connected graph with k edges labeled uniquely with the keys from T is $2 \cdot m^{-k-\gamma(H)+1} \leq 2 \cdot m^{-k-s+1}$ as H could be acyclic.

$$\begin{aligned}
\delta(k) &\leq \sum_{s=0}^{\hat{s}+1} \sum_{l_a=0}^{2l} k! \frac{2 \cdot N(k, l_a, 1, s)}{m^{k+s-1}} \\
&\leq \mathcal{O}(1) \cdot \frac{k! \cdot k^{\mathcal{O}(1)}}{((1+\varepsilon)n)^{k-1}} \\
&\leq \mathcal{O}(1) \cdot n \cdot \frac{k! \cdot k^{\mathcal{O}(1)}}{((1+\varepsilon)n)^{k}}.
\end{aligned}$$

□

Note that applying Corollary 6.4 yields $C(n) = n$ and thus we can state:

Corollary 6.10
Let $\mathcal{D}$ denote the set of all connected multigraphs with at most n edges. Then $(h_1, h_2) \in \hat{\mathcal{R}}^{2l}_{r,m}$ works fully randomly on a graph in $\mathcal{D}$ with probability $\mathcal{O}(n \cdot r^{-l})$.

Thus, we observed that hash functions from the class $\hat{\mathcal{R}}$ work with high probability fully randomly on connected graphs as well. The full randomness on excess core structures showed us that the failure probability of the idealized insertion algorithm matches the failure probability observed in the case of fully random hash functions. We are now able to reuse our results observed in the case of too long eviction chains, as those only occur in connected components of a graph and we can assume full randomness of the hash functions with high probability. As the performance observations for the lookup and delete operation hold, we will focus on the performance of insertions. The main problem of the following analysis is that we can only assume full randomness if we choose a good pair of hash functions from $\hat{\mathcal{R}}$ and every rehash could choose a new, bad pair. Thus, we will use a slightly different approach to prove that insertions run in amortized expected constant time. The idea comes from [DW03].

The full randomness inside connected components makes it possible for us to reuse our results on the probability of a long simple path in a connected component. Thus, if our stash has maximum size $\hat{s}$, we set $\mathit{MaxLoop} = 3(\hat{s} + 2)\lceil \log_{1+\varepsilon} n \rceil + 1$ to make failing insertions due to long simple paths in the graph dominated by the probability that a rehash occurs because the cuckoo graph has excess at least $\hat{s} + 1$. Thus, we can concentrate on "real" failing insertions if the hash functions behave in a good way.

If we sample (h_1, h_2) randomly from $\hat{\mathcal{R}}^{2l}_{r,m}$, we require that (h_1, h_2) works fully randomly in both cases: on excess core structures and inside the connected components of the graph. If (h_1, h_2) does not satisfy one of these assumptions, we always assume that our analysis fails and the worst things happen. The probability that a randomly chosen pair of hash functions does not fulfill one of those requirements is given by $\mathcal{O}(n \cdot r^{-l})$. We set $r = n^{\beta}$, $\frac{1}{2} < \beta < 1$ and $l = 2(\hat{s}+2)$ and thus can assume that a hash pair does work in the good way for both cases with probability at least $1 - \mathcal{O}(n^{-\hat{s}-1})$.

We consider a phase of ρn operations on the data structure, where $\rho > 0$ is constant and $n \geq |S|$ with $S \subseteq U$ denotes the set of those keys that are stored in the tables at any time during the phase. The table is filled with some keys according to a pair of hash functions $(h_1, h_2) \in \hat{\mathcal{R}}^{2l}_{r,m}$. We will disregard deletions and lookups, as we are only interested to show that the insertions are expected to take time $\mathcal{O}(n)$.

The phase is divided into subphases. The start of the first subphase is at the beginning of the phase and a new subphase starts if a rehash event occurs and we choose new hash functions. We set *MaxLoop* as seen above.

We start by stressing certain observations regarding a subphase. A subphase ends prematurely, if the insertion procedure cannot insert a key and moves it to an already full stash. The last subphase is the subphase that succeeds in building the data structure defined by the ρn operations.

We are looking for answers to the following three questions to obtain a time bound for a single arbitrary insertion operation:

1. What is the expected number of subphases?
2. How long does a subphase take in the worst case?
3. What is the expected time in the last succeeding subphase?

Lemma 6.11
Let $\hat{s}$ denote the maximum stash size. Then the expected number of subphases is at most $1 + \mathcal{O}(n^{-\hat{s}-1})$.

Proof. Focus a prematurely ending subphase. This phase ended, because the stash was already full when an insert operation tried to move an additional key into the stash.

We already argued that we can focus on failing insertions due to bad edge sets with probability $1-\mathcal{O}(n^{-\hat{s}-1})$, because (h_1, h_2) works fully randomly inside connected components with this probability.

If $G(S, h_1, h_2)$ is not l-bad regarding excess core structures, the failure probability matches the observations in the fully random case and thus equals $\mathcal{O}(n^{-\hat{s}-1})$. The probability that (h_1, h_2) is l-bad is bounded by $\mathcal{O}(n^{-\hat{s}-1})$. Note that the argument holds recursively for other failing subphases and the lemma follows. □

Furthermore, we have to inspect the worst-case running time of a subphase.

Lemma 6.12
The total time spent during a subphase is $\mathcal{O}(n \log n)$ in the worst case.

Proof. Note that there are never more than $\mathcal{O}(n)$ keys in the table. An insertion takes time $\mathit{MaxLoop} = \Theta(\log n)$ in the worst case and there are, even in the case of a rehash at the beginning of the subphase, at most $\mathcal{O}(n)$ insert operations. Hence no subphase takes longer than $\mathcal{O}(n \log n)$ in the worst case, even if (h_1, h_2) behaves bad. □

Lemma 6.13
The expected time spent in the last subphase is $\mathcal{O}(n)$.

Proof. If a bad pair of hash functions is used in the last subphase, we expect that we spend $\mathcal{O}(n \log n \cdot n^{-\hat{s}-1}) = o(n)$ time in this subphase. If the pair of hash functions is good it follows by Lemma 4.14 that each successful insertion runs in expected constant time. A failing insertion takes $\Theta(\log n)$ steps, but only occurs with probability bounded by $\mathcal{O}(n^{-1})$ and thus we expect that we spend $o(1)$ time in failing insertions. It follows that we expect to spend time $\mathcal{O}(n) \cdot \mathcal{O}(1) + o(n) = \mathcal{O}(n)$ in the last subphase and the lemma follows. □

So we expect to spend $\mathcal{O}(n \log n) \cdot \mathcal{O}(n^{-\hat{s}-1}) = o(n)$ in prematurely ending subphases and $\mathcal{O}(n)$ in the last successful subphase. An arbitrary insertion takes us $(\mathcal{O}(n) + o(n))/n$ amortized. It follows that an arbitrary insertion runs in amortized expected constant time. We formulate the following corollary to summarize our results on the performance of the different operations.

Corollary 6.14
Let $m = (1+\varepsilon)n$, $r = n^{\delta}, 1/2 < \delta < 1$, and $\hat{s}$ denote the maximum stash size. Assume that (h_1, h_2) is chosen randomly from $\hat{\mathcal{R}}_{r,m}^{4(\hat{s}+2)}$. Assume further that a phase involving at most n keys and ρn operations is run as described above.

Then insertions can be carried out in amortized expected constant time. Lookup and delete operations can be carried out in worst-case constant time.

6.4. Conclusion

With $\hat{\mathcal{R}}_{r,m}^{2l}$ we obtained a suitable hash class for cuckoo hashing with a stash. We have shown that the failure probability and the performance of the insert, delete and lookup operation is similar to the performance observed in the case of fully random hash functions, if we choose the parameters l and r in an appropriate way and proposed suitable parameter settings. Furthermore, we suggested a general framework for working with hash class $\hat{\mathcal{R}}$ and it is an interesting task to find new examples to which we can apply it.

7. Experimental Evaluation

In the previous chapters we thoroughly studied the theoretical behavior of stash-based cuckoo hashing. We provided an upper bound for the probability that cuckoo hashing with a stash of size $\hat{s}$ will fail to insert all elements in a key set S. We conducted our analysis assuming fully random hash functions being available for free first and later provided a family of realistic hash functions that achieved an almost random behavior.

In this chapter we will study the behavior of cuckoo hashing with an additional stash in a practical scenario. We will not only concentrate on a stash in standard cuckoo hashing, but also consider two popular refinements of cuckoo hashing – d-ary and blocked cuckoo hashing. If the reader is not familiar with these variants, he will find a description featuring the main results of both in Section A.3 in the appendix. We used the Java programming language in version 6 and the Java standard random number generator for the implementation.

7.1. Standard Cuckoo Hashing

The insertion algorithm is implemented in its idealized version, hence the insertion of an element only fails if its insertion yields a loop. Furthermore, we do not restrict the size of the stash to a maximum value to obtain the exact required stash size. The keys are drawn randomly from the universe of positive integers in $[10^7]$ and the key sets do not contain duplicates. We use cubic hash functions[16] to obtain the hash values of the keys. Moreover, we utilize two tables of size m each, where $m \in \{500, 5000, 50000, 500000\}$ and choose a random key set of $(1-\delta)m$ keys, where $\delta \in \{0.2, 0.15, 0.1, 0.06, 0.04, 0.02, 0.01\}$. The load factor of the hash table is thus between 40% and 49.5%, which is close to the analyzed maximum load of 50%. For every set of parameters (m, δ) we conduct 10^6 trials. We counted the exact stash size after each insertion trial. In the following tables, we will present the results of our experiments. The number inside the table cell shows the number of insertion trials that ended on the stash size given in the top row for a combination of m and δ on the left side.

[16] $h : x \mapsto (ax^3 + bx^2 + cx + d) \mod p \mod m$, where $a, b, c, d \in U$ are randomly chosen and $p > m$ is a prime. Note that this class is 4-wise independent.

m	stash:	0	1	2	3	4	5	6	7	8	9	>
500	δ =0.20	992334	7320	321	24	1	–	–	–	–	–	–
500	δ =0.15	984621	14199	1061	104	12	3	–	–	–	–	–
500	δ =0.10	967520	28447	3470	477	69	13	4	–	–	–	–
500	δ =0.06	941902	47876	8225	1570	342	70	12	3	–	–	–
500	δ =0.04	921292	62495	12431	2898	684	153	34	11	2	–	–
500	δ =0.02	895410	79125	18805	4869	1315	347	93	28	6	1	1
500	δ =0.01	878845	89236	23097	6393	1761	488	128	37	13	2	–
5000	δ =0.20	998768	1217	14	1	–	–	–	–	–	–	–
5000	δ =0.15	996807	3122	70	1	–	–	–	–	–	–	–
5000	δ =0.10	990034	9400	529	29	6	1	1	–	–	–	–
5000	δ =0.06	971083	25372	2946	491	90	11	6	–	1	–	–
5000	δ =0.04	948729	42279	7091	1481	327	71	17	4	1	–	–
5000	δ =0.02	908540	68666	16434	4482	1314	375	136	41	8	3	1
5000	δ =0.01	877841	86616	24086	7584	2542	883	284	116	33	8	7
50000	δ =0.20	999868	132	–	–	–	–	–	–	–	–	–
50000	δ =0.15	999641	358	1	–	–	–	–	–	–	–	–
50000	δ =0.10	998582	1405	12	1	–	–	–	–	–	–	–
50000	δ =0.06	993672	6072	237	19	–	–	–	–	–	–	–
50000	δ =0.04	983056	15493	1279	153	15	2	1	–	1	–	–
50000	δ =0.02	947400	43054	7462	1539	401	102	27	13	2	–	–
50000	δ =0.01	904132	70955	17374	5116	1594	560	175	56	27	7	4
500000	δ =0.20	999989	11	–	–	–	–	–	–	–	–	–
500000	δ =0.15	999951	49	–	–	–	–	–	–	–	–	–
500000	δ =0.10	999845	155	–	–	–	–	–	–	–	–	–
500000	δ =0.06	999212	782	6	–	–	–	–	–	–	–	–
500000	δ =0.04	997339	2607	52	1	1	–	–	–	–	–	–
500000	δ =0.02	984289	14422	1135	131	21	2	–	–	–	–	–
500000	δ =0.01	949607	41384	6919	1543	402	116	25	2	1	1	–

Figure 18: Stash sizes required for cuckoo hashing using two tables of size m each, $|S| = (1 - \delta)m$ keys from the universe $U = [10^7]$ and cubic hash functions. Results were measured over a sample size of 10^6.

In general we can see in Figure 18 that in the majority of the cases no stash is required to insert all elements into the hash table without failing for a single element[17]. Nevertheless, we can identify two very useful applications of a stash.

In the example of small table sizes, a stash proves to be useful even in the situation of a rather small load of 40%. Using a stash of size 5 yields a success rate of 100% for table sizes starting at 500. On the other hand, the closer we get to a load of 50%, the more likely a failure becomes. If we used a stash of size 9, the worst success probability would have been 99.9993%.

Clearly, those two cases do not exclude each other and can be combined to achieve

[17]The lowest success rate is 87.78% on $(5000, 0.01)$ for the parameters (m, δ).

higher success rates. This shows that the application of a small-sized additional memory decreases the failure probability in practical scenarios, allowing us to almost completely avoid rehashes in the data structure with small cost.

A load close to 50% is in most cases not desirable. Two well-known refinements of standard cuckoo hashing to achieve a higher load of the hash table are d-ary cuckoo hashing and blocked cuckoo hashing. Kirsch et al. presented in [KMW08] a short practical evaluation of d-ary cuckoo hashing and we will conduct an evaluation in the same style as seen above for both variants in the next two sections.

7.2. d-ary Cuckoo Hashing

We focus our experiments on a variant of d-ary cuckoo hashing described by Fotakis et al. in [FPSS03]. The failure probability of d-ary cuckoo hashing is given by $\mathcal{O}(n^{4-2d})$. Thus, the failure probability decreases by a factor n^{-2} each time we increase d and hence we expect that the impact of the stash becomes smaller the larger we choose d. In [KMW08], Kirsch et al. showed that a stash of maximum size $\hat{s}$ reduces the failure probability to $\mathcal{O}(n^{(1-d)(\hat{s}+1)})$. This reduced failure probability shows in the experiments as well, as for n large enough even a stash size of 1 will become unnecessary.

For $d \geq 2$ we use d separated tables of size $\lceil m/d \rceil$ and each hash function is in charge of one table. Furthermore, we will use a random walk based insertion algorithm instead of a breadth-first-search to avoid the necessary bookkeeping. The parameter setting matches the one used in the previous section.

The keys are drawn randomly from the universe of positive integers in $[10^7]$ and the key sets do not contain duplicates. We use cubic hash functions to obtain the hash values of the keys. Moreover, we choose a random key set of $(1 - \delta)m$ keys, where δ depends on d. Each insertion runs at most $2n + 1$ steps before we declare it a failure, where n is the number of elements in the hash table prior to the insertion of the element. As the failure rate of d-ary cuckoo hashing is much better than standard cuckoo hashing, we reduced the number of different values of δ and focused on values close to the theoretical thresholds for the load factor observed by Dietzfelbinger et al. in [DGM+09]. For every set of parameters (m, d, δ) we conducted 10^5 trials. We will start with 3-ary cuckoo hashing.

m	stash:	0	1	2	3	4	5	6	7	8	9	>
501	δ =0.12	98831	760	220	104	38	22	12	6	5	2	–
501	δ =0.11	95715	2389	923	434	252	131	78	41	14	13	10
501	δ =0.10	86576	6335	2857	1643	1021	665	397	217	134	64	91
501	δ =0.09	68729	11062	6365	4411	3059	2138	1502	1016	673	418	627
5001	δ =0.12	100000	–	–	–	–	–	–	–	–	–	–
5001	δ =0.11	100000	–	–	–	–	–	–	–	–	–	–
5001	δ =0.10	99991	6	1	1	–	1	–	–	–	–	–
5001	δ =0.09	94042	1679	916	553	476	378	288	253	224	187	1004
50001	δ =0.12	100000	–	–	–	–	–	–	–	–	–	–
50001	δ =0.11	100000	–	–	–	–	–	–	–	–	–	–
50001	δ =0.10	100000	–	–	–	–	–	–	–	–	–	–
50001	δ =0.09	100000	–	–	–	–	–	–	–	–	–	–

Figure 19: Stash sizes required for 3-ary cuckoo hashing with $|S| = (1-\delta)m$ keys from the universe $U = [10^7]$ and cubic hash functions. Results were measured over a sample size of 10^5.

Note that the theoretical threshold for the table load of 3-ary cuckoo hashing is 91.79% and thus $\delta = 0.09$ is very close to this threshold. The failure probability of 3-ary cuckoo hashing is $\mathcal{O}(n^{-2})$ and thus the required stash sizes decrease rapidly compared to standard cuckoo hashing. As we can see in Figure 19, even in the case of a load of 91%, there is no benefit when using a stash in the case of large tables. In the case of small tables ($m = 501$), the stash is a valuable extension, especially if the load is close to the theoretical maximum. A stash size of 9 seems to be suitable to achieve success rates of over 99% in this case, which is much better than 68.7% without a stash. In the case of $m = 5001$, a stash size of 9 increases the success rate from 94% to 99% if the load is close to the theoretical maximum. If the load factor is smaller, there is almost no impact of an added stash.

How does this impact change for higher values of d? Let us take a look at 4-ary cuckoo hashing.

m	stash:	0	1	2	3	4	5	6	7	8	9	>
500	δ =0.05	99388	538	58	12	1	2	–	–	–	–	1
500	δ =0.04	93655	4774	1046	333	118	42	19	8	3	–	2
500	δ =0.03	66703	17767	7888	3895	1965	975	448	218	90	25	26
5000	δ =0.05	100000	–	–	–	–	–	–	–	–	–	–
5000	δ =0.04	100000	–	–	–	–	–	–	–	–	–	–
5000	δ =0.03	99148	512	177	58	39	20	18	9	9	3	7
50000	δ =0.05	100000	–	–	–	–	–	–	–	–	–	–
50000	δ =0.04	100000	–	–	–	–	–	–	–	–	–	–
50000	δ =0.03	100000	–	–	–	–	–	–	–	–	–	–

Figure 20: Stash sizes required for 4-ary cuckoo hashing with $|S| = (1-\delta)m$ keys from the universe $U = [10^7]$ and cubic hash functions. Results were measured over a sample size of 10^5.

Note that the failure probability of 4-ary cuckoo hashing is $\mathcal{O}(n^{-4})$ and the theoretical threshold is 97.68%. Figure 20 depicts the results for 4-ary cuckoo hashing. As in the case of 3-ary cuckoo hashing, a stash seems unnecessary for big tables. In the case of small tables, a stash size of 9 seems to be a good choice again to achieve a success rate of approximately 99.97% for $\delta = 0.03$. For a medium-sized table, a stash is only required to lower the failure rate if the load is close to the threshold.

m	stash:	0	1	2	3	4	5	6	7	8	9	>
500	δ =0.03	99814	177	4	–	–	–	–	–	–	–	5
500	δ =0.02	94424	4945	527	86	11	4	1	–	–	–	2
500	δ =0.01	47634	30302	13956	5515	1804	563	162	50	10	3	1
5000	δ =0.03	100000	–	–	–	–	–	–	–	–	–	–
5000	δ =0.02	100000	–	–	–	–	–	–	–	–	–	–
5000	δ =0.01	87584	7276	2483	1184	615	344	203	124	79	40	68
50000	δ =0.03	100000	–	–	–	–	–	–	–	–	–	–
50000	δ =0.02	100000	–	–	–	–	–	–	–	–	–	–
50000	δ =0.01	100000	–	–	–	–	–	–	–	–	–	–

Figure 21: Stash sizes required for 5-ary cuckoo hashing with $|S| = (1-\delta)m$ keys from the universe $U = [10^7]$ and cubic hash functions. Results were measured over a sample size of 10^5.

In the case of 5-ary cuckoo hashing, the theoretical threshold is 99.24%. Note that $\delta = 0.01$ is really close to this value and hence the failure rates without an added stash are higher than the previously observed rates for 3- and 4-ary cuckoo hashing, even though the failure rate is down to $\mathcal{O}(n^{-6})$. Nevertheless, we can see in Figure 21 that our observations are again valid for this case.

In the case of big tables, a stash is practically useless. For small tables and a stash size of 9, only 1 out of 100.000 constructions fail if we operate close to the theoretical maximum. In the case of a medium-sized table, a stash is again useful when we operate close to the theoretical threshold.

In conclusion, a stash is especially useful if the table size is small. Even close to the theoretical threshold, an added stash makes it extremely unlikely that the data structure cannot be built successfully. In contrast to the case of standard cuckoo hashing, the stash becomes unnecessary if we use big tables, which is due to the already improved failure probability of $\mathcal{O}(n^{4-2d})$ for $d \geq 3$ – compared to $\mathcal{O}(n^{-1})$ in standard cuckoo hashing.

We will take a look at another candidate for a higher load of the hash table – blocked cuckoo hashing – next.

7.3. Blocked Cuckoo Hashing

Dietzfelbinger and Weidling [DW07] introduced blocked cuckoo hashing as an alternative approach to achieve a higher load of the hash table compared to standard cuckoo hashing. The failure probability of blocked cuckoo hashing, where each bucket can store d elements, is given by $\mathcal{O}(n^{1-d})$. Thus the failure probability for $d = 2$ matches the failure probability of standard cuckoo hashing, but decreases by a factor of n^{-1} each time we increase d and hence we expect that the impact of the stash becomes smaller the larger we choose d. In [KMW08], Kirsch et al. showed that a stash of maximum size $\hat{s}$ reduces the failure probability to $\mathcal{O}(n^{(1-d)(\hat{s}+1)})$. This reduced failure probability shows in the experiments as well, as for n large enough even a stash size of 1 will become unnecessary.

Let $m = (1 + \varepsilon)n$ denote the table size and assume that $m \mod d = 0$, otherwise increase m. We divide the table into m/d blocks and each of the two hash functions maps a key from the universe to a block. We use a random walk approach to insert keys into the hash table to avoid the bookkeeping of a breadth-first-search. The parameter settings are similar to the evaluation of d-ary cuckoo hashing. We obtained the following experimental results for a block size of 2.

m	stash:	0	1	2	3	4	5	6	7	8	9	>
500	δ =0.15	97917	1079	446	245	136	81	46	23	14	5	8
500	δ =0.14	94905	2317	1079	661	405	257	156	92	58	31	39
500	δ =0.13	88601	4384	2380	1658	1054	745	452	286	173	120	147
500	δ =0.12	77545	7121	4378	3196	2331	1779	1205	795	624	390	636
500	δ =0.11	61422	9706	6715	5335	4269	3265	2615	2042	1460	1082	2089
5000	δ =0.15	99988	12	–	–	–	–	–	–	–	–	–
5000	δ =0.14	99984	16	–	–	–	–	–	–	–	–	–
5000	δ =0.13	99977	18	3	–	1	1	–	–	–	–	–
5000	δ =0.12	99282	234	99	65	61	46	38	23	32	25	95
5000	δ =0.11	83531	2967	1792	1488	1185	1010	911	769	749	621	4977
50000	δ =0.15	100000	–	–	–	–	–	–	–	–	–	–
50000	δ =0.14	99999	1	–	–	–	–	–	–	–	–	–
50000	δ =0.13	99997	3	–	–	–	–	–	–	–	–	–
50000	δ =0.12	99998	2	–	–	–	–	–	–	–	–	–
50000	δ =0.11	99959	17	2	1	1	3	3	–	2	–	12

Figure 22: Stash sizes required for blocked cuckoo hashing with $d = 2$ and $|S| = (1-\delta)m$ keys from the universe $U = [10^7]$ and cubic hash functions. Results were measured over a sample size of 10^5.

In Figure 22, the following can be observed. In the case of small tables, the addition of a stash of size 9 is extremely useful, especially if we operate close to a load of 89%. We increase the success rate from 61.42% to almost 98% in this way. The same holds for the case of a medium-sized table with 5000 places. In the case of bigger tables, a stash is almost unnecessary, because the success rate without a stash is already at least 99.96%. Let us see how this impact changes, if we allow the buckets to hold more items.

m	stash:	0	1	2	3	4	5	6	7	8	9	>
501	δ =0.09	99934	45	13	5	2	1	–	–	–	–	–
501	δ =0.08	99591	255	84	38	18	8	4	2	–	–	–
501	δ =0.07	98426	876	341	179	78	44	29	10	11	6	–
501	δ =0.06	92298	3607	1704	923	598	352	226	121	70	52	49
501	δ =0.05	76797	8430	4830	3255	2250	1537	1073	692	433	304	399
5001	δ =0.09	100000	–	–	–	–	–	–	–	–	–	–
5001	δ =0.08	100000	–	–	–	–	–	–	–	–	–	–
5001	δ =0.07	99999	1	–	–	–	–	–	–	–	–	–
5001	δ =0.06	100000	–	–	–	–	–	–	–	–	–	–
5001	δ =0.05	98127	645	311	204	171	112	69	75	46	56	184
50001	δ =0.09	100000	–	–	–	–	–	–	–	–	–	–
50001	δ =0.08	100000	–	–	–	–	–	–	–	–	–	–
50001	δ =0.07	100000	–	–	–	–	–	–	–	–	–	–
50001	δ =0.06	100000	–	–	–	–	–	–	–	–	–	–
50001	δ =0.05	100000	–	–	–	–	–	–	–	–	–	–

Figure 23: Stash sizes required for blocked cuckoo hashing with $d = 3$ and $|S| = (1-\delta)m$ keys from the universe $U = [10^7]$ and cubic hash functions. Results were measured over a sample size of 10^5.

In Figure 23, we can see that a stash is again very useful in the case of small tables, especially when we want to operate close to a load of 95%. Using a stash of size 9 increases the success rate from 76.8% to 99.6%. In the case of a medium-sized table, the stash helps us again if we operate at a load of 95%. If the table becomes bigger, there is no need for a stash.

m	stash:	0	1	2	3	4	5	6	7	8	9	$>$
500	δ =0.06	99963	28	6	3	–	–	–	–	–	–	–
500	δ =0.05	99592	264	88	31	16	7	2	–	–	–	–
500	δ =0.04	96822	1767	690	358	177	85	53	25	14	4	5
500	δ =0.03	84076	6962	3615	2186	1288	812	471	269	157	79	85
5000	δ =0.06	100000	–	–	–	–	–	–	–	–	–	–
5000	δ =0.05	100000	–	–	–	–	–	–	–	–	–	–
5000	δ =0.04	100000	–	–	–	–	–	–	–	–	–	–
5000	δ =0.03	99944	29	10	4	1	3	1	3	1	1	3
50000	δ =0.06	100000	–	–	–	–	–	–	–	–	–	–
50000	δ =0.05	100000	–	–	–	–	–	–	–	–	–	–
50000	δ =0.04	100000	–	–	–	–	–	–	–	–	–	–
50000	δ =0.03	100000	–	–	–	–	–	–	–	–	–	–

Figure 24: Stash sizes required for blocked cuckoo hashing with $d = 4$ and $|S| = (1-\delta)m$ keys from the universe $U = [10^7]$ and cubic hash functions. Results were measured over a sample size of 10^5.

In Figure 24, the results for blocked cuckoo hashing with block size 4 are depicted. For $d = 4$ the failure probability of blocked cuckoo hashing is $\mathcal{O}(n^{-3})$ and obviously, the impact of the added stash is low, even in the case of a medium-sized table operating at a load of 97%. In the case of small tables, the stash is again extremely useful. A stash size of 9 increases the success rate at a load of 97% from 84.08% to 99.94%.

m	stash:	0	1	2	3	4	5	6	7	8	9	>
500	δ =0.05	99996	3	–	–	–	–	–	–	–	–	1
500	δ =0.04	99920	59	13	5	1	1	–	–	–	–	1
500	δ =0.03	98814	736	275	92	43	21	10	6	2	1	–
500	δ =0.02	89290	5583	2436	1344	631	358	169	89	56	19	25
5000	δ =0.05	100000	–	–	–	–	–	–	–	–	–	–
5000	δ =0.04	100000	–	–	–	–	–	–	–	–	–	–
5000	δ =0.03	100000	–	–	–	–	–	–	–	–	–	–
5000	δ =0.02	99991	2	6	–	–	–	–	1	–	–	–
50000	δ =0.05	100000	–	–	–	–	–	–	–	–	–	–
50000	δ =0.04	100000	–	–	–	–	–	–	–	–	–	–
50000	δ =0.03	100000	–	–	–	–	–	–	–	–	–	–
50000	δ =0.02	100000	–	–	–	–	–	–	–	–	–	–

Figure 25: Stash sizes required for blocked cuckoo hashing with $d = 5$ and $|S| = (1-\delta)m$ keys from the universe $U = [10^7]$ and cubic hash functions. Results were measured over a sample size of 10^5.

For $d = 5$ the failure probability of blocked cuckoo hashing is $\mathcal{O}(n^{-4})$. In Figure 25, we can see that for medium and large tables, adding a stash seems unnecessary, even if we operate at a load of 98%. In the case of small tables, the stash is again extremely useful. A stash size of 9 increases the success rate at a load of 97% from 89.29% to 99.98%.

7.4. Conclusion

In this chapter, we have evaluated the practical impact when a stash is added to different types of cuckoo hashing. Firstly, we observed that using a stash in standard cuckoo hashing allows us to use high loads close to 50% without expecting a rehash event. Furthermore, a stash helps us in the case of smaller tables.

Secondly, enhancing the data structure by using a stash for d-ary and blocked cuckoo hashing improves the situation when using small tables and operating close to the thresholds of these two refinements of cuckoo hashing. In contrast to standard cuckoo hashing, the failure probability of those data structures is so low that the addition of a stash seems to be unnecessary for bigger tables.

We can conclude that in practical scenarios a stash size of only 9 guarantees tremendously improved success rates.

8. Conclusion

This thesis considered an approach to lower the failure probability of cuckoo hashing. We gave new proofs that the addition of a stash to cuckoo hashing indeed reduces the failure probability dramatically. They are, at least, less technically involved and only use basic counting arguments as well as simple probability theory. Moreover, we analyzed the performance of the different operations on the data structure. In particular, we obtained an approach for the delete operation that yields both: a worst-case constant running time and a small expected constant overhead in some subsequent insert operation.

Furthermore, we provided a class of realistic hash functions that yields the same performance as fully random hash functions when used in cuckoo hashing with a stash. We thus answered an open question raised by Kirsch et al. in [KMW08]. We further presented a generic framework of the hash class $\hat{\mathcal{R}}$ that is based on the work in [Woe02]. One open question in this area is, if it is possible to find new examples to which we can apply the proposed framework.

In the end, we evaluated if the addition of a stash is useful in practical scenarios. We observed that adding a stash is reasonable to achieve a reduced failure probability in the case of small table sizes and high table load close to the theoretical maximum. In the case of d-ary and blocked cuckoo hashing, the stash was especially helpful if the tables were small. In the case of bigger tables, the failure probability of the variants without a stash is already low enough to achieve a success rate that renders the addition of a stash practically useless.

An interesting recent development is the work of Arbitman et al. [ANS09], which reuses the approach of a stash in a general fashion. They incorporate a queue which buffers some elements (stash elements, but also elements whose insertion takes long time) and insert only in subsequent constant steps. Further, the same work group introduced a new concept in [ANG10], stating: *"We construct the first dynamic dictionary that enjoys the best of both worlds: we present a two-level variant of cuckoo hashing that stores n elements using* $(1+\varepsilon)n$ *memory words, and guarantees constant-time operations in the worst case with high probability."* — which is clearly a great development.

In conclusion it can be said that a stash is a simple and efficient way to lower the failure probability of cuckoo hashing, also in the refined versions of d-ary and blocked cuckoo hashing. It is easy to implement and can be supported by hardware in high-performance embedded scenarios, combining the worst-case guarantees of the lookup operation with an adaptable failure probability of the insert operation.

A. Appendix

A.1. An Alternative Proof for the Expected Constant Size of Components in the Cuckoo Graph

In this section, we present an alternative proof of Lemma 3.9. Note that it is inaccurate, because we cannot assume an independent random experiment in each round.

Lemma A.1
Let $G = G(S, h_1, h_2)$ denote the cuckoo graph for the key set $S \subseteq U$ of size n and fully random hash functions. Let $m = (1 + \varepsilon)n$. Let $x \in S$ denote an edge of G.
Then the expected size of the connected component of x is constant.

Proof. Fix the edge x that is incident with two vertices u and v in the graph. We only concentrate our reasoning on the vertex u. As the hash functions behave fully randomly, the probability that a node v is a neighbor of u is $1/m$. The whole process is binomially distributed with parameters n and $1/m$. Hence, the expected number of neighbors of u is n/m. We denote the (multi-)set of neighbors of u with $N(u)$. Note that if we count the neighbors of u we obtain the number of edges incident with u.

Moreover, we fix such a vertex v in $N(u)$. For v it holds the same line of argument to express the expected size of $N(v)$. Furthermore, we are only interested in edges we did not count already. Clearly, there are at most $|N(v)| - 1$ edges incident with v we did not discover in the first place, because v was discovered by u and hence we already counted uv. To obtain the expected size of the component of the edge x, we have to bound the expected numbers of edges we discover by obtaining all neighbors of a vertex and recursively obtaining all neighbors of these neighbors.

Let $S(x)$ denote the size of the component containing x. The expected number of edges in the component of $x = uv$ is bounded by

$$|N(u)| + \sum_{v \in N(u)} |N(v)| + \ldots,$$

summing over all the neighbor sets of vertices we discover during this process. As each discovered vertex v has at most $|N(v)| - 1$ undiscovered neighbors, we can assume that the number of undiscovered neighbors is dominated by a binomial distribution with parameters n and $1/m$ for every vertex we discover during this process. Thus, we can maintain the expected size of the component containing x in the following way:

$$\begin{aligned}
\mathrm{E}(|S(x)|) &\leq \frac{n}{m} + \Big(\frac{n}{m}\Big)^2 + \Big(\frac{n}{m}\Big)^3 + \dots \\
&= \sum_{i=1}^{n} \Big(\frac{n}{m}\Big)^i \\
&= \sum_{i=1}^{n} \Big(\frac{1}{1+\varepsilon}\Big)^i \\
&= \mathcal{O}(1).
\end{aligned}$$

□

A.2. Core Structures of Graphs with a Given Excess

In Section 4.2 we examined excess core structures of a cuckoo graph. An excess-s core structure C had the following properties:

1. C has excess exactly s.
2. C has no leaf edges.
3. C contains only components with at least two cycles.

In the experiments conducted for the experimental evaluation in Chapter 7, some of the graphs with an excess of 5 were visualized and processed using GraphViz[18]. An example of such a realistic cuckoo graph with excess 5 and an excess-3 core structure is given in Fig. 26.

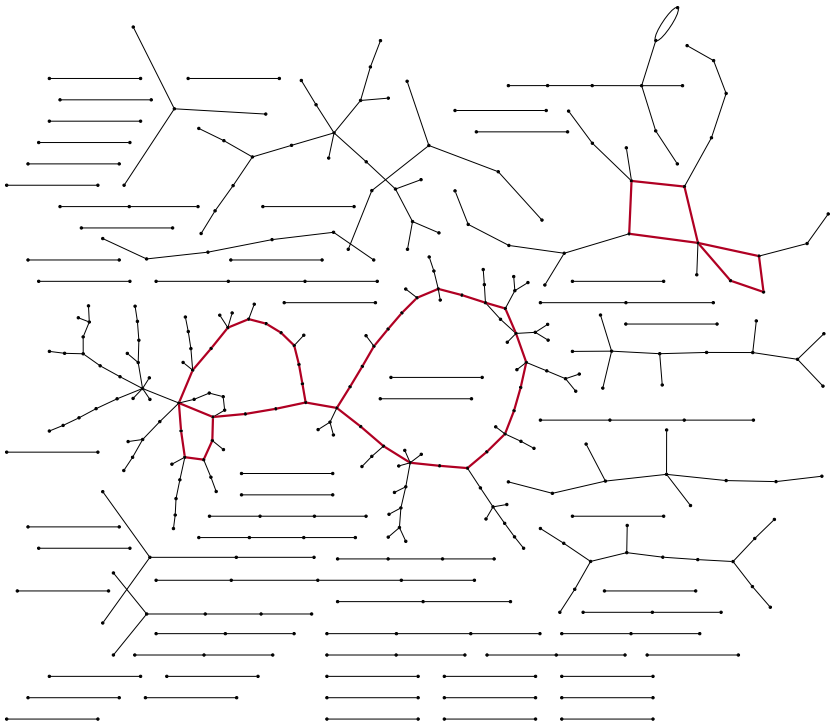


Figure 26: A cuckoo graph with excess 5. Red lines mark a possible excess-3 core structure.

[18] `http://www.graphviz.org`

A.3. Achieving Higher Load: Blocked and d-ary Cuckoo Hashing

Although an additional stash can decrease the failure probability of standard cuckoo hashing, it cannot increase the maximal load of the hash table. For most applications, a load factor of less than 50% is not desirable. After understanding how cuckoo hashing works, one could ask two questions.

1. What happens if we use more than 2 hash functions?
2. What happens if we store more than one element in a table cell?

The answers to these questions yield two different popular variants of standard cuckoo hashing that both achieve a higher load of the hash table – d-ary cuckoo hashing and blocked cuckoo hashing. The former was introduced by Fotakis et al. in [FPSS03] and the latter is due to Dietzfelbinger and Weidling [DW07].

In d-ary cuckoo hashing, we use d hash functions and only one table of size $m = (1 + \varepsilon)n$. Each hash function maps keys from a universe to cells in this table. A popular variant of this approach uses d tables of size $(1+\varepsilon)n/d$ and each hash function is responsible for its own table.

In blocked cuckoo hashing, we use two hash functions and each cell can store d elements. We use a table of size $(1 + \varepsilon)n/d$ and each hash function maps keys from a universe to a bucket. As long as the bucket does not contain d elements, a key can be inserted into this bucket.

Both of these variants use space $(1 + \varepsilon)n$ only, which is an advantage compared to the $2(1 + \varepsilon)n$ of standard cuckoo hashing. As we will neither analyze d-ary nor blocked cuckoo hashing theoretically in this thesis, we are satisfied with describing the important operations (lookup, delete, insert) and state the main known results without proving them.

The method of d-ary cuckoo hashing is a direct generalization of standard cuckoo hashing. Instead of using two hash functions, we use d hash functions. Thus 2-ary cuckoo hashing is equivalent to standard cuckoo hashing. The lookup, delete and insert operations are natural extensions of their equivalents in standard cuckoo hashing. A lookup returns true, if the element is stored in one of its d possible locations. To delete an element, one has to check at most d table cells and remove the element if it resides in one of these cells. The insertion procedure becomes more complicated.

In contrast to standard cuckoo hashing, where a key could only reside in two different positions, a key in d-ary cuckoo hashing for $d \geq 3$ has more than two possible locations. When we evict a key, its next position is not determined anymore, for we can choose from $d - 1$ possible locations.

One could ask if d-ary cuckoo hashing has a graph theoretical model like standard cuckoo hashing. We can think of a bipartite multigraph with n vertices on the left side and $(1+\varepsilon)n$ vertices on the right. The left side represents the items, the right side the different table cells. An item can be stored in d different cells and hence there are d edges from the node associated with the item on the left side to its possible locations on the right side. We can think of d-ary cuckoo hashing as a left-perfect matching. If

there exists such a matching, we can directly obtain a possible assignment of items to memory cells.

Fotakis et al. give the following failure probability of d-ary cuckoo hashing.

Theorem A.2
Given a constant $\varepsilon \in (0, 1)$, for any integer $d \geq 2(1+\varepsilon)\ln(\frac{e}{\varepsilon})$, the bipartite graph $G(L, R, E)$ contains a left-perfect matching with probability at least $1 - \mathcal{O}(n^{4-2d})$.

In 2008, Dietzfelbinger and Pagh improved this result on the connection between d and ε in [DP08]. An interesting observation is that the possible minimal values for ε and a given d, such that the data structure can be constructed with high probability, were already known in another community that considered the k-XORSAT problem. In 2009, Dietzfelbinger et al. proved this connection in [DGM+09]. In the same year, this problem was independently solved by Frieze and Melsted in [FM09] and Fountoulakis and Panagiotou in [FP09], too. We provide the threshold table loads for given values of d in Figure 27.

d	3	4	5	6	7
load	0.9179	0.9768	0.9924	0.9974	0.9991

Figure 27: Thresholds of the hash table load for given values of d in d-ary cuckoo hashing.

Hence, if we use 3-ary cuckoo hashing with $(1-\delta)m$ keys, δ should be larger than 0.0821. As we can see experimentally, $\delta = 0.09$ already works well.

We will introduce two different algorithmic approaches for insertions. We denote the key we want to insert with x. The first approach is a breadth-first-search starting from the possible locations of x.

If one of the possible locations of x is free, we can immediately store x there. Otherwise, we consider the other choices of the keys in these d cells. If we notice that there exists a key y that can be placed into an empty cell, we can insert x to the cell of y and move y to this empty table cell. Using a breadth-first-search, we obtain an empty table cell (if such a cell exists) and the path of evictions to this empty table cell. With some bookkeeping during this process, we can move the keys according to the path from x to the empty cell in the BFS tree. We call such a path an *augmenting path*.

The downside of this method is that we have to maintain a lot of information during the breadth-first-search. The analysis of this insertion algorithm is due to Fotakis et al. and states the following results.

Theorem A.3
For any positive $\varepsilon > 1/5$ and integer $d \geq 5+3\ln(1/\varepsilon)$, the incremental algorithm that augments along a shortest augmenting path takes $(1/\varepsilon)^{\mathcal{O}(\ln d)}$ expected time per left vertex/element to maintain a left-perfect matching in the cuckoo graph. Moreover, the algorithm visits at most $o(n)$ right vertices before it finds an augmenting path with high probability.

Is there a way to decrease the amount of data we have to maintain during the process of obtaining an empty table cell? The solution to this is using a random walk to find an empty cell. The approach is much simpler than before.

We are again in the situation that all possible locations of x are already occupied by other keys. We choose one of these d locations uniformly and insert x there, evicting a key y. For this key, we choose one of its $d-1$ other locations randomly and insert the key in this cell. If this cell was empty, we are done. Otherwise, we proceed to throw out random elements until an empty cell has been found.

It is clearly questionable if a random walk performs well. For a long time, this approach was not analyzed and it was just observed that it performs well in practical implementations, e.g., the experiments reported on in the paper of Fotakis et al. [FPSS03] used a random walk approach, although they only analyzed the case of using a BFS. In [FMM09], Frieze et al. provided a first step towards a well understood random walk insertion procedure. They proved the following result:

Theorem A.4
Conditioned on an event of probability $1 - \mathcal{O}(n^{4-2d})$ regarding the structure of the cuckoo graph G, the expected time for insertion into a cuckoo hash-table using Algorithm 1 and 2 is $\mathcal{O}(\log^{1+\gamma_0+2\gamma_1} n)$ where $\gamma_0 = \frac{d+\log d}{(d-1)\log(d/3)}$ and $\gamma_1 = \frac{d+\log d}{(d-1)\log(d-1)}$, assuming $d \geq 8$ and if $\varepsilon \leq \frac{1}{6}$, $d \geq 4 + 2\varepsilon - 2(1+\varepsilon)\log\left(\frac{\varepsilon}{1+\varepsilon}\right)$. Furthermore, the insertion time is $\mathcal{O}(\log^{2+\gamma_0+2\gamma_1} n)$ with probability $1 - \mathcal{O}(n^{-2d})$.

The condition belongs to the result of Fotakis et al. that the insertion of a key will fail because the graph does not have a left-perfect matching, which happens with probability $\mathcal{O}(n^{4-2d})$. Note that the values for d are too high for most practical implementations, where even $d \in \{3, 4, 5, 6\}$ performs well. Reducing the minimal value for d was left as an open problem.

On the downside of d-ary cuckoo hashing, d memory cells must be visited and it is likely that they are far away from each other memory-wise. Thus, cache architectures do not yield performance increases using this approach. Another variant of cuckoo hashing, namely blocked cuckoo hashing, stores more than one item per table cell, making it possible to probe sequential memory cells during the operation of the data structure.

Blocked cuckoo hashing was introduced by Dietzfelbinger and Weidling in [DW07]. Sanders et al. [SEK00, San01] already studied the static case with fixed bucket sizes in a different context. In contrast to d-ary cuckoo hashing, we stick to the approach of using only two hash functions to map keys into the table. Each table cell can hold up to $d \geq 1$ keys. We call a table cell that can hold d keys a block.

We can reuse the graph theoretical setting of d-ary cuckoo hashing. The left side of the bipartite multigraph represents the keys. The right side represents the possible locations of an item. Each vertex on the left side is connected to its $2d$ possible memory cells on the right side. Again, we are interested in a left-perfect matching to obtain a possible assignment of keys to table cells.

To look up or delete a key, one has to evaluate the two hash function values and look into the two possible blocks of a key. The insert operation is closely connected to the insert operation described for d-ary cuckoo hashing. Let x denote the key we are going to insert in an already pre-filled table.

If at least one of the possible blocks of x is not full, we can store x directly. If this is not the case, we have to find a sequence of key evictions such that the last key in this sequence is moved to a free block and all other keys of this sequence are moved subsequently until x can be placed into one of its two possible positions.

Dietzfelbinger and Weidling gave the following result for the failure probability of blocked cuckoo hashing.

Theorem A.5
Let $\varepsilon > 0$ be arbitrary. Assume that $d \geq 1 + \frac{\ln(1/\varepsilon)}{1-\ln 2}$. Let n be sufficiently large, let $S \subseteq U$ be an arbitrary set of n keys, and let T be a table with m blocks of size d each, where $dm \geq (1+\varepsilon)n$. Further assume that $h_1, h_2 : U \to [m]$ are fully random hash functions.

Then with probability $1 - \mathcal{O}(n^{1-d})$ the functions h_1, h_2 are suitable for S and d.

Similar to the case of d-ary cuckoo hashing there exist refinements of this observation. In 2007, Fernholz and Ramachandran [FR07] and Caine, Sanders and Wormald [CSW07] provided an exact analysis of blocked cuckoo hashing. We supply the threshold loads for blocked cuckoo hashing with block size d in Figure 28.

d	2	3	4	5	6	7
load	0.8970	0.9590	0.9803	0.9894	0.9940	0.9964

Figure 28: Thresholds of the hash table load for given values of d for blocked cuckoo hashing with block size d.

Again, there are two popular choices for an insertion algorithm. On the one hand, we can use a breadth-first-search to find an empty table cell. This approach seems unsuitable for practical implementations, because of the bookkeeping, but is well-understood analytically. [DW07] stated the following result on insertions if we use the BFS approach.

Theorem A.6
Let $\varepsilon > 0$ be arbitrary. Assume that $d \geq 90.1 \cdot \ln(1/\varepsilon)$. Let n be sufficiently large, let S denote an arbitrary set of n keys, let $x \in U - S$, and let T be a table with m blocks of size d each, where $dm \geq (1+\varepsilon)n$. Assume that $h_1, h_2 : U \to [m]$ are fully random hash functions, and that the keys from S have been stored in T by an algorithm that is ignorant of $h_1(x), h_2(x)$.

Then the expected time to insert x by the BFS procedure is $(1/\varepsilon)^{\mathcal{O}(\log d)}$.

In practice, we will rather use the simple random walk approach than the memory-hog BFS. Unfortunately, no one analyzed the performance of the random walk ap-

proach in a similar manner to [FMM09] for d-ary cuckoo hashing, although d-ary and blocked cuckoo hashing are very much alike in their graph theoretical background.

A.4. An Example for the Peeling Process

In Section 5.3, we discussed $2l$-reduced subgraphs of l-bad graphs regarding excess core structures of the cuckoo graph $G = (S, h_1, h_2)$. If G was l-bad, we claimed that there exists a subgraph $K(T)$ of G with $|g(T)| = |T| - l$ and the following properties:

1. There is one connected component in $K(T)$ that has at most $2l$ leaf and cycle edges.
2. All other connected components contain only inner edges.
3. There are at most $2l$ connected components.

This section gives an example for the described peeling process in Lemma 5.7. If G is l-bad, we start with some $T \subseteq S$ with $|g(T)| < |T| - l$ and $K(T)$ forms an excess core structure. For our example we assume $|g(T)| = |T| - l - 2$. We mark all those keys colliding under the g-function of $(h_1, h_2) \in \hat{\mathcal{R}}^{2l}_{r,m}$. If we only visualize the connected components of $K(T)$ and the number of marked edges in each component, we get the following picture.

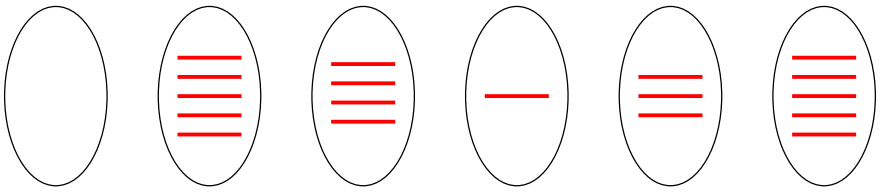


Figure 29: $K(T)$ consisting of 6 connected component and the visualized number of marked edges in each component.

The first step of our peeling process is throwing away unmarked components. Furthermore, we remove marked components under the condition that $|g(T)| \leq |T| - l$ holds. We remove the component containing one marked edge in this way (result: $|g(T)| = |T| - l - 1$) and cannot remove any further components, as $|g(T)| \leq |T| - l$ would not hold afterwards.

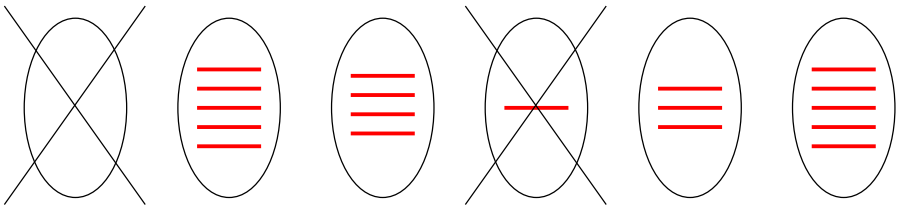


Figure 30: Removed and remaining components of $K(T)$ after the first step of the peeling process.

As described in the peeling process, we will now focus on an arbitrary connected component. For this example, we choose the component with four marked edges. Note that all these components cannot contain leaf edges, because they are part of an excess core structure. This component looks as follows.

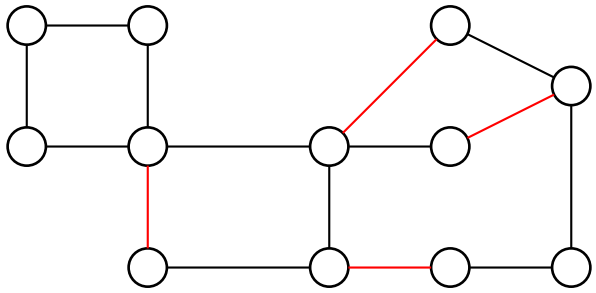


Figure 31: Component of an excess core structure with four keys colliding under g.

It is sufficient to remove one marked edge to achieve $|g(T)| = |T| - l$. We choose the rightmost marked edge to achieve this. Note that removing a colliding key can automatically unmark another edge, if both of them were the only keys colliding at this g-value. This happens in our case and we obtain the following graph.

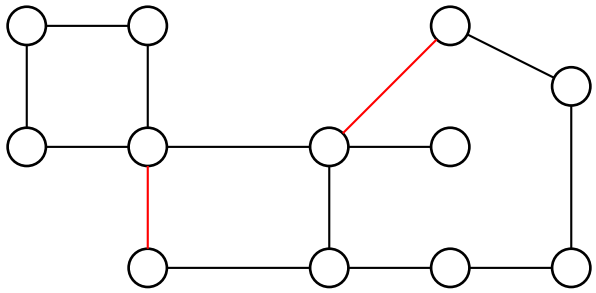


Figure 32: Component of an excess core structure after removing a marked edge.

We achieved our first goal: $|g(T)| = |T| - l$ holds. Unfortunately, the processed component is far away from how it should look in the end. Our next goal is to transform this component without destroying connectivity into a component where all leaf and cycle edges are marked without deleting a marked edge. We start by removing arbitrary unmarked cycle edges and get the following graph.

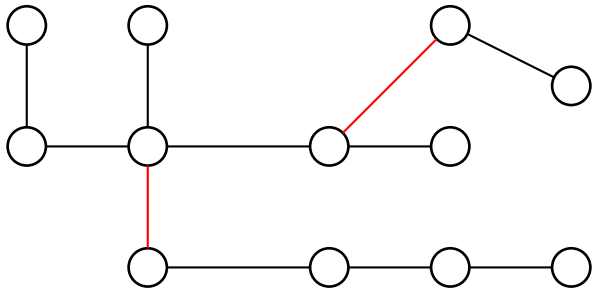


Figure 33: Component of an excess core structure after removing unmarked cycle edges.

The graph does not contain any unmarked cycle edges anymore, and we will start to repeatedly remove leaf edges until all leaf edges are marked. After a while the graph looks as follows.

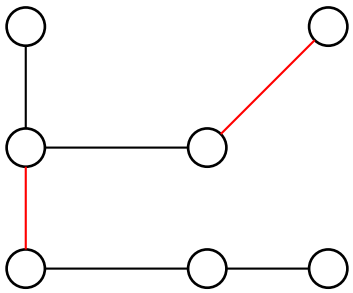


Figure 34: Component of an excess core structure during the process of removing unmarked leaf edges.

Eventually, the graph has the following structure.

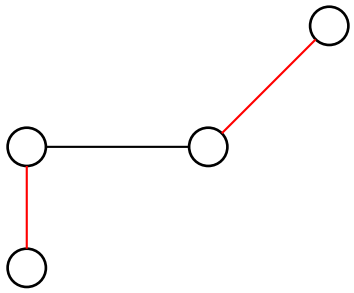


Figure 35: Component of an excess core structure after removing all non-marked leaf edges.

Together with the remaining untouched components, this is the $2l$-reduced subgraph of the l-bad graph.

B. References

[ANG10] Yuriy Arbitman, Moni Naor, and Gil Gegev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. Manuscript, 2010.

[ANS09] Yuriy Arbitman, Moni Naor, and Gil Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming*, pages 107–118, 2009.

[BM08] John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph theory.* Graduate Texts in Mathematics 244. Berlin: Springer. xii, 651 p., 2008.

[BPS07] Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors. *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007.* SIAM, 2007.

[BPZ07] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Norbert Zeh, editors, *WADS*, volume 4619 of *Lecture Notes in Computer Science*, pages 139–150. Springer, 2007.

[CKRT04] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In J. Ian Munro, editor, *SODA*, pages 30–39. SIAM, 2004.

[CSW07] Julie Anne Cain, Peter Sanders, and Nicholas C. Wormald. The random graph threshold for k-orientiability and a fast algorithm for optimal multiple-choice allocation. In Bansal et al. [BPS07], pages 469–476.

[CW77] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions (extended abstract). In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 106–112, New York, NY, USA, 1977. ACM.

[DadH90] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In Mike Paterson, editor, *ICALP*, volume 443 of *Lecture Notes in Computer Science*, pages 6–19. Springer, 1990.

[DGM+09] Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink. Tight thresholds for cuckoo hashing via xorsat. *CoRR*, abs/0912.0287, 2009.

[DGMP92] Martin Dietzfelbinger, Joseph Gil, Yossi Matias, and Nicholas Pippenger. Polynomial hash functions are reliable (extended abstract). In Werner Kuich, editor, *ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer, 1992.

[Die05] Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, August 2005.

[DM03] Luc Devroye and Pat Morin. Cuckoo hashing: Further analysis. *Inf. Process. Lett.*, 86(4):215–219, 2003.

[DP08] Martin Dietzfelbinger and Rasmus Pagh. Succinct data structures for retrieval and approximate membership (extended abstract). In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP*, volume 5125 of *Lecture Notes in Computer Science*, pages 385–396. Springer, 2008.

[DS09a] Martin Dietzfelbinger and Ulf Schellbach. On risks of using cuckoo hashing with simple universal hash classes. In Claire Mathieu, editor, *SODA*, pages 795–804. SIAM, 2009.

[DS09b] Martin Dietzfelbinger and Ulf Schellbach. Weaknesses of cuckoo hashing with a simple universal hash class: The case of large universes. In Mogens Nielsen, Antonín Kucera, Peter Bro Miltersen, Catuscia Palamidessi, Petr Tuma, and Frank D. Valencia, editors, *SOFSEM*, volume 5404 of *Lecture Notes in Computer Science*, pages 217–228. Springer, 2009.

[DW03] Martin Dietzfelbinger and Philipp Woelfel. Almost random graphs with simple hash functions. In *STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 629–638, New York, NY, USA, 2003. ACM.

[DW07] Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theor. Comput. Sci.*, 380(1-2):47–68, 2007.

[FM09] Alan M. Frieze and Páll Melsted. Maximum matchings in random bipartite graphs and the space utilization of cuckoo hashtables. *CoRR*, abs/0910.5535, 2009.

[FMM09] Alan M. Frieze, Páll Melsted, and Michael Mitzenmacher. An analysis of random-walk cuckoo hashing. In Irit Dinur, Klaus Jansen, Joseph Naor, and José D. P. Rolim, editors, *APPROX-RANDOM*, volume 5687 of *Lecture Notes in Computer Science*, pages 490–503. Springer, 2009.

[FP09] Nikolaos Fountoulakis and Konstantinos Panagiotou. Sharp load thresholds for cuckoo hashing. *CoRR*, abs/0910.5147, 2009.

[FPSS03] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. In Helmut Alt and Michel Habib, editors, *STACS*, volume 2607 of *Lecture Notes in Computer Science*, pages 271–282. Springer, 2003.

[FR07] Daniel Fernholz and Vijaya Ramachandran. The k-orientability thresholds for $G(n, p)$. In Bansal et al. [BPS07], pages 459–468.

[KMW08] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. In *ESA '08: Proceedings of the 16th annual European symposium on Algorithms*, pages 611–622, Berlin, Heidelberg, 2008. Springer-Verlag.

[Kut09] Reinhard Kutzelnigg. A further analysis of cuckoo hashing with a stash and random graphs of excess r, 2009.

[MU05] Michael Mitzenmacher and Eli Upfal. *Probability and Computing : Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[PR01] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In Friedhelm Meyer auf der Heide, editor, *ESA*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2001.

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[San01] Peter Sanders. Reconciling simplicity and realism in parallel disk models. In *SODA*, pages 67–76, 2001.

[Sch09] Ulf Schellbach. *On Risks of Using a High Performing Hashing Scheme with Common Universal Classes*. Dissertation, Technische Universität Ilmenau, Ilmenau, Germany, 2009.

[SEK00] Peter Sanders, Sebastian Egner, and Jan H. M. Korst. Fast concurrent access to parallel disks. In *SODA*, pages 849–858, 2000.

[Woe02] Philipp Woelfel. Ideen zu "almost random graphs with simple hash functions" (in german). Manuscript, 2002.

C. List of Figures

D. Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Ilmenau, den 22.03.2010

Martin Aumüller

E. Thesen

1. Die Größe des Stashs unter Nutzung einer idealisierten Einfügeoperation wird durch den sogenannten Excess des Cuckoo-Graphen beschrieben.

2. Um zu erkennen, dass ein Stash der Größe $\hat{s}$ nicht ausreicht um eine Schlüsselmenge einzufügen, genügt es im Cuckoo-Graphen eine excess-$(\hat{s}+1)$ Kernstruktur zu finden.

3. Der Stash ermöglicht es die Fehlerwahrscheinlichkeit von Cuckoo-Hashing drastisch zu senken.

4. Die Löschoperation kann so beschrieben werden, dass sie auch im schlechtesten Fall eine konstante Laufzeit hat und der Stash nach einer fehlgeschlagenen Einfügung minimale Größe besitzt.

5. Der Stash hat keinen Einfluss auf die erwartete Laufzeit erfolgreicher Einfügeoperationen.

6. Hashfunktionen aus der Klasse $\mathcal{R}^{2l}_{r,m}$ arbeiten mit hoher Wahrscheinlichkeit wie vollkommen zufällige Hashfunktionen auf den von uns betrachteten Excess-Kernstrukturen.

7. Wir können unsere Resultate bzgl. der Zufälligkeit von Hashfunktionen aus der Klasse $\mathcal{R}^{2l}_{r,m}$ auf Excess-Kernstrukturen nutzen, um eine allgemeine Sichtweise auf die Zufälligkeit dieser Funktionen auf anderen Graphklassen zu abstrahieren.

8. Die Nutzung eines kleinen Stashs in Cuckoo-Hashing ermöglicht es uns die Datenstruktur auch bei hoher Tabellenauslastung ohne Gefahr eines Rehashs zu nutzen.

9. Bei d-ärem Cuckoo-Hashing ist der Einsatz eines Stash bei ausreichend großen Tabellen ohne Nutzen.

Ilmenau, den 22.03.2010

Martin Aumüller