Die approbierte Originalversion dieser Dissertation ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

DISSERTATION

Random Bipartite Graphs and their Application to Cuckoo Hashing

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften unter der Leitung von

Univ-Prof. Dipl.-Ing. Dr. techn. Michael Drmota
Institut für Diskrete Mathematik und Geometrie
E104

eingereicht an der Technischen Universität Wien
Fakultät für Mathematik und Geoinformation

von

Dipl.-Ing. Reinhard Kutzelnigg
0025840
Badsiedlung, 385
8250 Vorau

Datum

Unterschrift

Vorwort

Die Entwicklung moderner Computer führt dazu, dass wir in unserem täglichem Leben einer immer größer werdenden Flut von Daten ausgesetzt werden. Dies betrifft sowohl die eigene Festplatte, als auch die über das Internet zugängliche Information oder das immense Wissen, das in immer mehr Datenbanken gesammelt wird. Um in dieser gewaltigen Menge an Information die Orientierung zu behalten, bedarf es ausgefeilter Verfahren, die bei der Suche und der Organisation helfen. Hashalgorithmen werden seit mehr als 50 Jahren zu diesem Zweck eingesetzt. Noch immer werden laufend neue Datenstrukturen vorgeschlagen, um möglichst optimale Leistung zu erzielen. Eines dieser Verfahren ist Cuckoo Hashing, das in dieser Arbeit ausführlich untersucht wird.

Das erste Kapitel erklärt das Prinzip von Hashalgorithmen und gibt eine kurze Übersicht der bekanntesten und gebräuchlichsten Verfahren. Im Detail werden diverse Varianten von Cuckoo Hashing erläutert, auch auf in der Praxis einsetzbare Hashfunktionen wird dabei eingegangen. Nach einer knappen Wiederholung einiger mathematischer Grundlagen erfolgt die Herleitung der Sattelpunksverfahren, die in den weiteren Kapiteln verwendet werden. Die folgenden Kapiteln befassen sich schließlich jeweils mit der Analyse eines Aspekts des Verfahrens. Insbesondere wird hier die Erfolgswahrscheinlichkeit im subkritischen und kritischen Fall, der mittlere Aufwand für den Aufbau und die Effizienz von Suchoperationen untersucht. Am Ende eines jeden dieser Kapitel befindet sich eine Interpretation der Ergebnisse, zusammen mit experimentellen Daten.

Ich möchte mich auf diesem Weg bei Herrn Prof. Drmota für seine Betreuung bedanken, die es mir ermöglicht hat, dieses interessante Thema auszuarbeiten. Vielen Dank auch an Herrn Prof. Heuberger, der sich kurzfristig bereit erklärt hat, die Rolle eines Gutachters zu übernehmen und so die rasche Fertigstellung ermöglicht hat. Weiters gilt mein Dank auch Matthias Dehmer und meinem Vater, die sich die Zeit genommen haben, diese Arbeit Korrektur zu lesen und mir viele hilfreiche Hinweise gegeben haben.

Diese Arbeit wurde soweit möglich unter Verwendung freier Software auf Linux Systemen erstellt, wovon insbesondere LaTeXund der gcc Compiler von besonderer Wichtigkeit waren. Alle im Zusammenhang mit dieser Dissertation erstellten Programme sind ebenfalls unter einer freien Lizenz erhältlich.

Kurzfassung

Diese Doktorarbeit beschäftigt sich mit der Ermittlung der Eigenschaften spezieller zufälliger Graphen, die in enger Verbindung mit dem Algorithmus Cuckoo Hashing stehen. Dieser wurde in Pagh and Rodler [2004] eingeführt und weist als besondere Eigenschaft eine konstante Zugriffszeit auf, die im Gegensatz zu herkömmlichen Hashalgorithmen nicht nur im Durchschnitt gilt. Weitere Resultate betreffend Cuckoo Hashing sind unter anderem in den Artikeln Devroye and Morin [2003], Dietzfelbinger and Weidling [2007] und Fotakis et al. [2003] angeführt, eine detaillierte Analyse aller Aspekte fehlt jedoch.

Diese Dissertation kann folgendermaßen in zwei Aspekte unterteilt werden. Der erste Schwerpunkt der Arbeit ist eine genaue Analyse des mittleren Verhaltens von Cuckoo Hashing. Dazu zählt insbesondere die Wahrscheinlichkeit, dass die Konstruktion der Datenstruktur erfolgreich ist. Es gilt, dass der Aufbau der Hashtabelle asymptotisch fast immer gelingt, so ferne die Auslastung einen festen Wert kleiner als 0.5 nicht überschreitet. Im so genannten „kritischen Fall“, der einer Auslastung von 0.5 entspricht, sinkt die Erfolgswahrscheinlichkeit asymptotisch jedoch auf einen Wert von ca. 0.861. Weiters wird eine Schranke für den mittleren Aufwand, der zum Aufbau der Datenstruktur notwendig ist hergeleitet, die linear mit der Größe der Datenstruktur wächst. All diese Untersuchungen basieren auf einer Analyse des so genannten Cuckoo Graphen, welcher ein zufälliger bipartiter Graph ist, dessen Eigenschaften in engem Zusammenhang zur Datenstruktur stehen. Dieser Graph wird mit Hilfe von erzeugenden Funktionen modelliert und anschließend wird durch Verwendung einer (doppelten) Sattelpunktsmethode eine asymptotische Approximation der Koeffizienten ermittelt. Insbesondere lassen sich dadurch Eigenschaften wie die Komponentenstruktur des Graphen, die Größe der Baumkomponenten, die Anzahl der im Graphen enthaltenen Kreise oder die Wahrscheinlichkeit, dass keine Komponente auftritt, die mehr als einen Kreis beinhaltet, ermitteln. Diese Resultate sind natürlich auch in andern Zusammenhängen von Interesse, z.B. in der Genetik, siehe Blasiak and Durrett [2005].

Ein weiterer Schwerpunkt liegt in der Untersuchung des Einflusses von Modifikationen der dem Verfahren zu Grunde liegenden Datenstruktur. Insbesondere werden zwei neue Verfahren, „asymmetrisches Cuckoo Hashing“ und „vereinfachtes Cuckoo Hashing“ genannt, eingeführt. Deren Analyse beruht wiederum auf dem Studium von zufälligen Graphen, die an das jeweilige Verfahren angepasst werden. Weiters wird eine asymptotische Analyse des Verhaltens von Suchoperationen all dieser Algorithmen durchgeführt und diese Ergebnisse werden mit den bekannten Resultaten der Standardalgorithmen Lineares Sondieren und Double Hashing verglichen. Diese Untersuchung zeigt, dass der mittlere Aufwand von Such- und Einfügeoperationen durch Verwendung von vereinfachtem Cuckoo Hashing im Vergleich zu allen anderen Varianten sinkt. Verglichen mit herkömmlichen Hashalgorithmen die auf Offener Adressierung basieren, ergibt sich eine

Beschleunigung von Suchvorgängen, jedoch steigen die Kosten von Einfügeoperationen. Eine Übersicht der in dieser Arbeit hergeleiteten Resultate bezüglich Cuckoo Hashing ist in der an die englischsprachigen Kurzfassung angeschlossenen Tabelle enthalten.

Schlussendlich wird ein in C++ implementiertes Softwarepaket erstellt, das die Simulation von allen oben erwähnten Hashalgorithmen beherrscht. Mit Hilfe eines weiteren Programmes ist es auch möglich den Wachstumsprozess von gewöhnlichen und bipartiten Graphen zu simulieren. Diese Software wird dazu verwendet um die theoretische Analyse mit numerischen Daten zu unterlegen.

Diese Arbeit steht im Zusammenhang mit den Artikeln Drmota and Kutzelnigg [2008], Kutzelnigg [2006] und Kutzelnigg [2008] und wurde vom FWF-Projekt S9604 „Analytic and Probabilistic Methods in Combinatorics“ und dem EU FP6-NEST-Adventure Programm, Projekt Nummer 028875 (NEMO) unterstützt.

Abstract

This thesis deals with the analysis of a special kind of random graphs and their application to the analysis of a relatively new hash table data structure called cuckoo hashing, that was introduced in Pagh and Rodler [2004]. Its main notable feature is, that it provides constant worst case search time. Further results concerning cuckoo hashing can be found in several other papers, in particular Devroye and Morin [2003], Dietzfelbinger and Weidling [2007], and Fotakis et al. [2003]), however no detailed analysis has been carried out so far.

The contribution of this thesis is twofold. First, we present a precise average case analysis of cuckoo hashing. In particular, we determine the probability that the construction of a cuckoo hash table produces no conflicts. We conclude that the construction is asymptotically almost always successful, if the load factor does not exceed a fixed limit less than 0.5. Moreover, we consider the "critical case", that corresponds to the load factor 0.5, and obtain, that the asymptotic success rate is reduced to approximately 0.861. Furthermore, we give an upper bound for the construction time that is linear in the size of the table. The analysis of the algorithm is based on a generating function approach to the so called Cuckoo Graph, a random bipartite graph that is closely related to the data structure. We apply a double saddle point method to obtain further asymptotic results concerning the structure of the graph, such as tree sizes, the number of cycles and the probability that no component containing more than one cycle occurs. Of course, there exist other applications of this results *e.g.* in genetics, see Blasiak and Durrett [2005].

Second, we analyse the influence on the performance caused by modifications of the underlying structure of the cuckoo hash table. The obtained data structures are named "asymmetric cuckoo hashing" and "simplified cuckoo hashing". Again, we provide an average case analysis, that is now based on different random graph models. Further, we perform an asymptotic analysis of the search costs of all this versions of cuckoo hashing and compare this results with the well known properties of double hashing and linear probing. In particular, our analysis shows, that the expected number of steps of search and insertion operations can be reduced by using the simplified version of cuckoo hashing instead of any other cuckoo hash algorithm. Compared to standard algorithms based on open addressing, we notice that the simplified data structure offers increased performance for search operations, but the expected construction time of the hash table increases. Table 0.1 presents an overview of the properties of cuckoo hashing algorithms.

Additionally, a C++ software that implements cuckoo hashing (including several modifications) as well as standard hash algorithms (*e.g.* double hashing or linear probing) is developed. Moreover, an additional program is provided to simulate the growth of bipartite or usual random graphs. All this software is used to support the analysis by numerical results.

This thesis is based on the research papers Drmota and Kutzelnigg [2008], Kutzelnigg [2006], and Kutzelnigg [2008]. Major part of the work was done within the FWF-project S9604 “Analytic and Probabilistic Methods in Combinatorics” and the EU FP6-NEST-Adventure Programme, Contract number 028875 (NEMO).

	simplified cuckoo hashing	standard cuckoo hashing	asymmetric cuckoo hashing
probability of success (subcritical)	$1-\frac{(5-2\varepsilon)(1-\varepsilon)^2}{48\varepsilon^3}\frac{1}{m}+\mathcal{O}\left(m^{-2}\right)$ Theorem 4.1	$1-\frac{(2\varepsilon^2-5\varepsilon+5)(1-\varepsilon)^3}{12(2-\varepsilon)^2\varepsilon^3}\frac{1}{m}+\mathcal{O}\left(m^{-2}\right)$ Theorem 4.2	$1-\frac{(1-\varepsilon)^3(10-2\varepsilon^3+9\varepsilon^2-15\varepsilon)}{12(2\varepsilon-\varepsilon^2-c^2)^3(c^2-1)}\frac{1}{m} + \frac{(1-\varepsilon)^3c^2(-3\varepsilon^2+9\varepsilon+2c^2-10)}{12(2\varepsilon-\varepsilon^2-c^2)^3(c^2-1)}\frac{1}{m}+\mathcal{O}\left(m^{-2}\right)$ Theorem 4.3
probability of success (critical)	$\sqrt{\frac{2}{3}}+o(1)$ Theorem 5.1	$\sqrt{\frac{2}{3}}+o(1)$ Theorem 5.2	
construction cost	$\leq \min\left(4, \frac{-\log\varepsilon}{1-\varepsilon}\right)n+\mathcal{O}(1)$ Theorem 7.1	$\leq \min\left(4, \frac{-\log\varepsilon}{1-\varepsilon}\right)n+\mathcal{O}(1)$ Theorem 7.2	$\leq \frac{1-c}{2(1-\varepsilon)}\left(\log\frac{1-c^2}{2\varepsilon(1-\varepsilon)-c^2} - \frac{2(1+c)}{\sqrt{1-c^2}}\operatorname{artanh}\frac{\varepsilon-1}{\sqrt{1-c^2}}\right)n+\mathcal{O}(1)$ Theorem 7.3
successful search	$\geq 2-\frac{1-e^{-\alpha}}{\alpha}+\mathcal{O}\left(m^{-1}\right)$ $\leq 2-\frac{1-e^{-2\alpha}}{2\alpha}+\mathcal{O}\left(m^{-1}\right)$ Theorem 8.1	$2-\frac{1}{2\alpha}\left(1-e^{-2\alpha}\right)+\mathcal{O}\left(m^{-1}\right)$ Theorem 8.2	$2-\frac{1+c}{2\alpha}\left(1-e^{-2\alpha/(1+c)}\right)+\mathcal{O}\left(m^{-1}\right)$ Theorem 8.3
unsuccessful search	$1+\alpha$ Theorem 8.1	$2-e^{-2\alpha}+\mathcal{O}\left(m^{-1}\right)$ Theorem 8.2	$2-e^{-2\alpha/(1+c)}+\mathcal{O}\left(m^{-1}\right)$ Theorem 8.3

Table 0.1: Overview of results concerning cuckoo hashing. We consider data structures possessing $2m$ memory cells and holding $\lfloor(1-\varepsilon)m\rfloor$ data points. For simplified and standard cuckoo hashing, we assume that ε is either fixed in $(0,1)$, or equals zero in the critical case. For the asymmetric algorithm, we use tables of size $m_1=\lfloor m(1+c)\rfloor$ respectively $m_2=2m-m_1$, for fixed $c\in[0,1)$ and assume that $\varepsilon\in(1-\sqrt{1-c^2},1)$ holds. Finally, we state the results concerning search operations in terms of the load factor $\alpha=\lfloor(1-\varepsilon)m\rfloor/(2m)$. As a consequence, these results can be directly compared to the well known properties of standard algorithms.

Contents

Chapter 1

Hashing

1.1 Introduction

This chapter gives a short survey of hash table based data structures which is a frequently used tool in computer science. Their efficiency has strong influence on the performance of many programs, because various applications are based on dictionary-like data structures. For example, the symbol table of a compiler for a computer language is often based on hashing (see, *e.g.*, Cormen et al. [2001]). A further example is the operating system Linux, which relies on hash tables to manage pages, buffers, inodes, and other kernel-level data objects (see, *e.g.*, Lever [2000]).

In the following, we are interested in a data structure that supports insertion, search and potentially also deletion operations. Further, we suppose that each data record is uniquely determined by a key. For instance, the (Austrian) national insurance number might be such a key assigned to an individual person. A data base query providing this number might give information as the name of the owner, age, insurance status and so on. Generally, such data structures are often called associative arrays. We may consider them as an extension of a simple array, the latter is a group of homogeneous elements of a specific data type. A hash table is just one possible implementation of an associative array, others are for instance self-balancing binary search trees or skip lists (see, *e.g.*, Cormen et al. [2001], Knuth [1998], Kemper and Eickler [2006], Pugh [1990]).

Usually, the number of stored data records is much smaller than the number of possible keys. For example, the Austrian social insurance number consists of 10 decimal digits, but there are only about 8 Million inhabitants in Austria. Thus, it would be very inefficient to use an array with 10^{10} cells to store all this data. In contrast to this, hashing requires memory proportional to the maximal number of stored keys only. We achieve this by using the so-called hash function to transform the key to a number of limited range instead of using the key as an array index directly. Figure 1.1 illustrates this approach. The value of the hash function of a given key is called the hash value of that key. This value provides us a location of the hash table, where we might store the data.

There is of course one big problem left. *What should we do if two different keys collide, that is, there exist keys that try to occupy the same memory cell?* Such a collision can be seen in Figure 1.1, because the keys d and g try to access the same storage place. We will discuss possible solutions later on, first we consider the question how likely it is, that

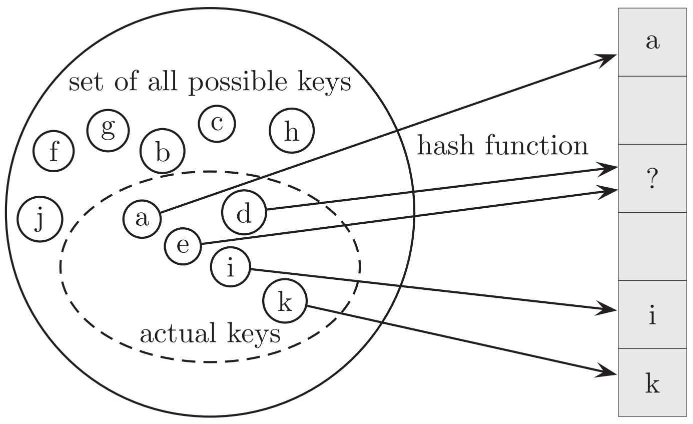


Figure 1.1: An associative array implemented by a hash table.

different keys share the same hash value under the assumption that these numbers are selected uniformly at random.

The well known birthday paradox shows us that collisions are likely to appear (see, *e.g.*, Cormen et al. [2001], Flajolet and Sedgewick [2001]). To be more precise, the probability that at least two people in a room are born on the same day of the year is greater than 50% if at least 23 people are present in the room. Transferred to hash tables, the solution of the generalised problem tells us to expect collisions, if the number of keys exceeds the square-root of the table size. However this occurs in all practical applications, otherwise the data structure would waste too much memory.

The properties of the chosen hash function do have great influence on the performance of the data structure. For instance, if all keys are mapped to the same position of the table, the performance decreases dramatically. Thus, it is important that the hash function distributes the values uniformly over the table size range m. Usually, the analysis of the performance of hash algorithms is based on the assumption that the hash values are independent uniformly drawn from $0, 1, \dots, m-1$. Hash functions, that satisfy this model under practical conditions will be discussed in Section 1.6.

Clearly, the ideal solution would be to avoid collisions completely. If we assume that all keys are known in advance, it is possible to choose the hash function in such a way, that no collisions occur. This approach is called perfect hashing and will be discussed in Section 1.4. Unfortunately it is very costly to search such a perfect hash function.

Classical techniques for dealing with collisions are briefly described in the following two sections.

1.2 Hashing with open addressing

Within the area of open addressing, we resolve collisions by successively examining the m cells of the table, until we discover an empty position. By doing so, we define a probe sequence of table positions

$$h(x,0), h(x,1), h(x,2), \dots, h(x,m-1)$$

for each key x. The first element of this sequence is of course the usual hash function $h(x)$. If this position is already occupied, we inspect $h(x,1)$ *etc.* until we either find an empty cell and store the key at this position or we perform a run through the whole sequence. In the latter case, an insertion is not possible. An example can be found in Figure 1.2. Obviously, the probe sequence should be a permutation of the set $\{0, 1, 2, \ldots, m-1\}$, which implies that insertion is only impossible if the table is fully occupied. Using this approach, the number of keys stored in the table can not exceed the number of initially allocated storage cells, except if the complete data structure is rebuilt using a table of increased size. Hence this algorithm is sometimes referred to as closed hashing, see Binstock [1996].

The most common ways to choose the probe sequence are the following (see, *e.g.*, Gonnet and Baeza-Yates [1991],Knuth [1998]):

- *Linear probing:* $h(x, i) = (h(x) + i) \mod m$. The algorithm uses the simplest possible probe sequence and is thus easy to implement. The major disadvantage of linear probing is, that it suffers from a problem known as primary clustering. Long runs of occupied cells occur with high probability, what increases the average cost of an operation. However, the practical behaviour might be better than theoretical analysis suggest, because of the memory architecture of modern computers. This is due to the fact that it might take less time to access several keys stored in adjoining memory cells if they belong to an already loaded cache line than to resolve a cache-miss (see Binstock [1996],Black et al. [1998],and Pagh et al. [2007] resp. Heileman and Luo [2005] for a different point of view).

- *Quadratic probing:* $h(x, i) = (h(x) + c_1 i + c_2 i^2) \mod m$, where c_1 and c_2 are constants. Although the expected performance of quadratic probing is decreased compared to linear probing, one of the main problems has not been resolved: Keys sharing the same hash value possess the identical probe sequence too. Thus a milder form of clustering, called secondary clustering, arises.

- *Double hashing:* $h(x, i) = (h(x) + ih_2(x)) \mod m$. The algorithm uses a second hash function h_2 that determines the increment between two successive probes. As a result, the probe sequence depends on two ways on the actual key. The value of h_2 should be relatively prime to m to ensure that all cells are covered by the probe sequence. This algorithm is still easy to implement and offers better average performance than linear and quadratic probing (see Knuth [1998]) and is hence the most common algorithm based on open addressing.

- *Exponential hashing:* $h(x, i) = (h(x) + a^i h_2(x)) \mod m$, where a is a primitive root of m. The exponential double hash family was first suggested by Smith et al. [1997] and further improved by Luo and Heileman [2004] to the form considered here. This type of hash function tends to spread the keys more randomly than standard double hashing, and it still leads to probing sequences of maximum length for all keys.

- *Uniform probing:* Uniform probing is a theoretical hashing model what assumes that the order in which the cells are probed is a random permutation of the numbers $0, 1, \ldots, m-1$. Hence it is easy to analyse but almost impossible to implement. Nonetheless, double hashing and uniform probing are indistinguishable for all practical purposes (see Gonnet and Baeza-Yates [1991]).

A major disadvantage of the just mentioned algorithms is, that it is not allowed to delete elements straight away, because otherwise keys might become unfindable. More precisely, we are not allowed to erase a key stored at position k if there exists a key x on position l such that $k = h(x,i)$ and $l = h(x,j)$ for a j greater than i hold. This problem can be overcome by “lazy deletions”. Instead of removing a key permanently, we just mark the element as deleted. Such positions are considered as empty during insertions, but on the other hand, they are treated like occupied cells during search operations. This idea is recommendable only if deletions are rare, because cells will never become empty again. Thus an unsuccessful search might take m steps, if no more empty cells exist, although the table is not full.

However, when linear probing is being used, it is possible to implement a more efficient deletion algorithm, to overcome this problem. This is due to the fact that only the run of keys starting immediately after a position k till the occurrence of the first empty cell could be influenced, if slot k is erased. Hence, we can delete a key x by modifying the table as it would have been, if x had never been inserted. See [Knuth, 1998, Algorithm R] for details. In contrast, the same approach is impracticable for other variants of open addressing, because each key could be influenced by a single deletion.

Several suggestions have been made to improve the behaviour of search operations of double hashing. This is based on the assumption that searches are much more common than insertions. Thus, it seems to be worth pursuing more work by rearranging keys during an insertion to speed up search operations. The most important techniques are (see, *e.g.*, Gonnet [1981],Knuth [1998],Munro and Celis [1986]):

- *Brent’s variation* (Brent [1973]): Whenever a collision appears, double hashing resolves this by moving the last inserted key according to its probing sequence until the first empty cell is found. Brent suggests to check if the keys occupying the locations along this probing sequence can be moved to an empty location, such that the total number of steps to find all keys decreases. More precisely, let d_i equal the number of steps that are required to move the key occupying the i-th cell of the probing sequence to an empty position. Brent’s algorithm selects the position i that minimises $i + d_i$.

- *Binary tree hashing* (Gonnet and Munro [1977]): This algorithm is a generalisation of Brent’s variant. Not only the keys occupying cells along the probing sequence of the newly inserted key might me moved, this kicked-out keys might further displace keys along their own probing sequence and so on. This modification leads to a slight improvement in the search performance compared to Brent’s algorithm. However, this variant requires additional memory, is much more complicated[1], and it takes more time to perform insertions.

- *Robin Hood hashing* (Celis et al. [1985]): In contrast to the previous rearranging schemes, this algorithm does not influence the expected number of steps to perform a successful search, however it does affect the variance. This can be easily done as follows: Instead of always moving the last inserted key, we resolve collisions moving the key that is closer to its initial position. Hence the expected length of longest probing sequence is reduced without a significantly higher insertion cost.

[1]This is due to the fact that we have to consider much more different probing sequences.

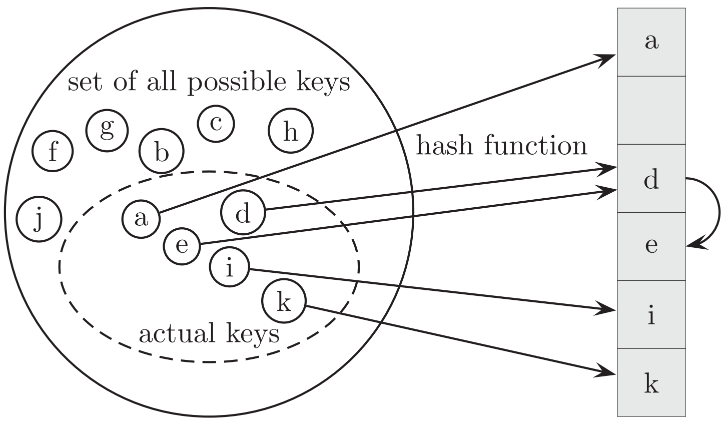


Figure 1.2: Collision resolution by open addressing.

An asymptotic approximation of the expected cost of the of search operations for non-full tables is given in Table 1.1. More details on algorithms based on open addressing can for instance be found in Cormen et al. [2001], Gonnet and Baeza-Yates [1991], or Knuth [1998].

1.3 Hashing with chaining

By using hashing with chaining, a linked list (see, *e.g.*, Cormen et al. [2001]) is used for each table entry to store all the keys, mapped to this position. The hash table itself might either consist of pointers only (this is called direct chaining) or represents an array of keys and pointers (this is called separate chaining) as depicted in Figure 1.3. To search a key, we evaluate the hash function and compare the elements of the corresponding list till the key is found or the end is reached.

We may decrease the cost of an unsuccessful search, if we keep all the lists in order by their key values. Note that this has no influence on the cost of a successful search, but we have to perform a search operation before an insertion. However, we should do the latter in any way, to assure that no duplicates are produced. Thus, there is no extra cost in keeping the lists in order.

A major advantage of hashing with chaining is, that the number of keys stored in the table might exceed the size of the table. Hence this algorithm is sometimes unfortunately called open hashing, what might easily be confused with open addressing. As a further benefit, deletions can be performed without complications, in contrast to algorithms based on open addressing (except linear probing). On the other hand, there are two drawbacks. First, hashing with chaining needs additional memory to store the pointers. However, this might be compensated by the fact, that it is sufficient to store a “abbreviated key” $a(x)$ instead of x, if x is fully determined by $a(x)$ and $h(x)$. Furthermore a memory management to allocate resp. disallocate list elements is needed. Second, it takes additional time to handle the pointers. Hence open addressing is usually preferable if we use “short” keys such as 32-bit integers and hashing with chaining is recommended for large keys like character strings.

An asymptotic approximation of the average cost of search operations of various hash

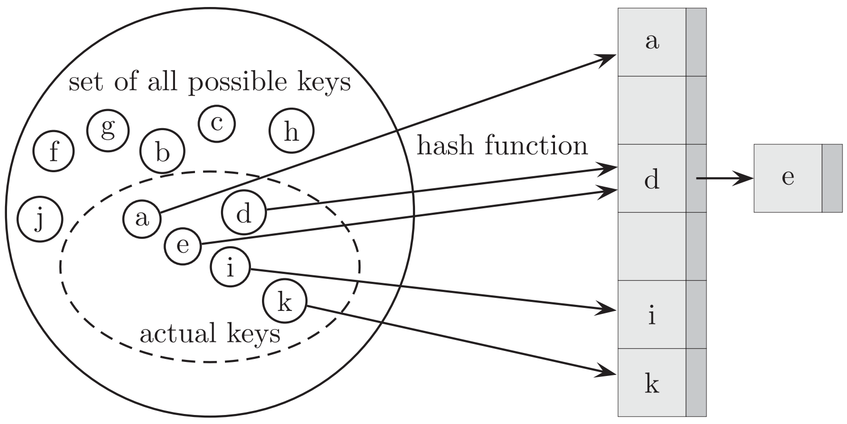


Figure 1.3: Collision resolution by chaining.

	successful search	longest expected successful search	unsuccessful search
linear probing	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$O(\log n)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
quadratic probing	$1+\log\frac{1}{1-\alpha}-\frac{\alpha}{2}$		$\frac{1}{1-\alpha}+\log\frac{1}{1-\alpha}-\alpha$
double hashing	$\frac{1}{\alpha}\log\frac{1}{1-\alpha}$	$-\log_\alpha m$	$\frac{1}{1-\alpha}$
Brent's variation	$1+\frac{\alpha}{2}+\frac{\alpha^3}{4}+\frac{\alpha^4}{15}+\dots$		
direct chaining	$1+\frac{\alpha}{2}$	$\Gamma^{-1}(m)$	α
separate chaining	$1+\frac{\alpha}{2}$	$\Gamma^{-1}(m)$	$\alpha+e^{-\alpha}$

Table 1.1: Asymptotic approximations of the cost of search operations. The results are obtained under the assumption that the hash values are independent uniformly drawn from $0, 1, \dots, m-1$. All results are presented in terms of the load factor α, that is the quotient of the number of keys and the memory cells of the data structure. Note that the given results of algorithms based on open addressing do not hold for full tables.

algorithms is given in Table 1.1. More details and proof of the claimed properties can *e.g.* be found in Cormen et al. [2001], Gonnet and Baeza-Yates [1991], or Knuth [1998].

1.4 Perfect hashing

The performance of hash table look-ups depends on the number of collisions. A perfect hash algorithm tries to completely avoid collisions using an injective hash function for a fixed set of keys. The catch is that the complexity of the construction and/or evaluation of this function increases. Another major weak point of many perfect hash functions is, that changing one key might force us to compute a completely new function. Therefore, most of this algorithms are only applicable to static tables where the data content remains unchanged after construction, but constant worst case look-up time is essential.

A survey on perfect hashing can be found in Czech et al. [1997]. For a static set of keys, an algorithm offering constant query time using $O(n)$ memory cells to store n keys was first proposed by Fredman et al. [1984]. An extension of this scheme to a dynamic dictionary was given by Dietzfelbinger et al. [1994]. However, all this solutions are more complex than the usual hash algorithms and more of theoretical interest than of practical relevance. As a consequence, new implementations based on simple modifications of standard algorithms with improved worst case behaviour have been suggested, see Azar et al. [1999], Broder and Mitzenmacher [2001], Dalal et al. [2005], Devroye and Morin [2003], Pagh and Rodler [2004], and Vöcking [2003]. One of these algorithms is cuckoo hashing, that will be discussed in detail in the next section.

1.5 Cuckoo hashing

1.5.1 Standard cuckoo hashing

Cuckoo hashing is a relatively new hash algorithm that provides constant worst case search time, contrary to the algorithms discussed in Sections 1.2 and 1.3. The algorithm was first introduced by Pagh and Rodler [2001a] (see also Pagh and Rodler [2004]) and a further analysis was done by Devroye and Morin [2003].

The algorithm is based on two tables of size m and makes use of two hash functions h_1 and h_2, both map a key to a unique position in the first resp. second table. These are the only allowed storage locations of this key and, hence search operations need at most two look-ups.

The main idea of cuckoo hashing is to resolve collisions by rearranging keys. A new key x is always inserted in the first table at position $h_1(x)$. If this cell was empty before the insertion of x, the operation is complete. Otherwise, there exists a key y such that $h_1(x) = h_1(y)$ holds. We proceed moving this key y to its alternative storage position $h_2(y)$. If this cell was preoccupied too, we proceed with this kick-out procedure until we hit an empty cell. The algorithm is named after the cuckoo, because this ejection is similar to the birds nesting habits. Of course, the insertion procedure may end up stuck in an endless loop if the same keys are kicked out again and again. In the the latter case, we perform a rehash, that is, we rebuild the whole data structure using new hash functions. As a strong point of the algorithm, this is a rare event if an ε in the interval $(0, 1)$ exists, such that the number of keys n satisfies $n = (1 - \varepsilon)m$. More details will be given in Chapter 4. Figure 1.4 depicts the evolution of a cuckoo hash table.

Similar to the model introduced in Section 1.1, the analysis of cuckoo hashing is based on the assumption, that the hash values of the keys $x_1, x_2, \dots, x_n$ form a sequence of independent uniform random integers drawn from $\{1, 2, \dots, m\}$. Further, if a rehash is necessary, we assume that the new hash values are independent from previous attempts. Hash functions suitable for the implementation of cuckoo hashing will be discussed in Section 1.6.

We model cuckoo hashing with help of a labelled bipartite graph (see, *e.g.*, Diestel [2005]), that is called the cuckoo graph (see also Devroye and Morin [2003]). The two labelled node sets T_1, T_2 correspond to the two hash tables. The insertion of a key x is encoded by an edge $(h_1(x), h_2(x)) \in T_1 \times T_2$. Further, we use edge labels to capture the evolution of the hash table. That is, the edge with label j corresponds to the j-th key that is inserted in the table.

Interestingly, the structure of this graph determines whether the hash table can be constructed successfully or not. It is is obviously necessary that every component of the cuckoo graph has less or equal edges than nodes, because it is impossible to store more than k keys in k memory cells. This means that all connected components are either trees (*i.e.* they contain no cycle) or unicyclic (*i.e.* they contain exactly one cycle). It is common to call a component of a graph complex if it is neither a tree nor unicyclic. On the other hand, it is easy to see that an endless loop in the insertion algorithms cannot occur in a tree or unicyclic component. There exist three different permitted cases that we consider separately:

- The new edge connects two different trees and we obtain a tree component of increased size. By induction, it is clear that each tree possesses exactly one node that corresponds to an empty storage cell. Once this position is fixed, there exists precisely one assignment of keys to the other memory cells. An insertion follows the unique path from the starting point to the empty cell in this tree component. Hence the number of steps is bounded by the component size of the tree and more precisely by the tree's diameter. See Figure 1.5 for an example.

- Both storage locations of the new inserted key belong to the same tree component. However there is no substantial difference compared to the previous case, instead that a unicyclic component arises.

- One vertex of the new edge belongs to a unicyclic component, while the other is contained in a tree. First, assume that the primary storage position of the new key belongs to the tree. Again there is no essential difference to the first case considered here. However, assume that the insertion procedure starts at the cyclic component that does not possess an empty cell. Note that there exist two possible assignments of the keys belonging to the edges forming the cycle: "clockwise" and "counter-clockwise". During the insertion, we follow the unique path from the starting point to the cycle, walk once around the cycle and change the orientation of the assignment, and follow the same path that brought us to the cycle back to the starting point. Further the insertion algorithm continues and places the new key in the tree component. An exemplary situation is depicted in Figure 1.6

Because of this close relation between the hash algorithm and the corresponding graph, we can analyse cuckoo hashing by considering bipartite multigraphs. For example, the probability that Cuckoo hashing works successfully with n keys and table size m equals the probability that a random bipartite multigraph with m nodes of each type and n edges has no complex component. Further, structural knowledge of the detailed structure of tree and unicyclic components provides information about the running time. For instance, the insertion cost of a key x such that the edge $(h_1(x), h_2(x))$ is contained in a tree component is bounded by the diameter of this tree. A detailed analysis of cuckoo hashing can be found in Chapters 4 and 7.

1.5.2 Asymmetric cuckoo hashing

A significant feature of the cuckoo hash algorithm described above is the unbalanced load, because the majority of keys will be usually stored in the first table. This is due to the fact, that an insertion always starts using h_1 and not a randomly selected hash function. Thus, more keys can be found requiring one step only, if a search operation always probes using h_1 first. Note that this unbalanced load does not influence the probability if the hash table is constructed successfully in any way.

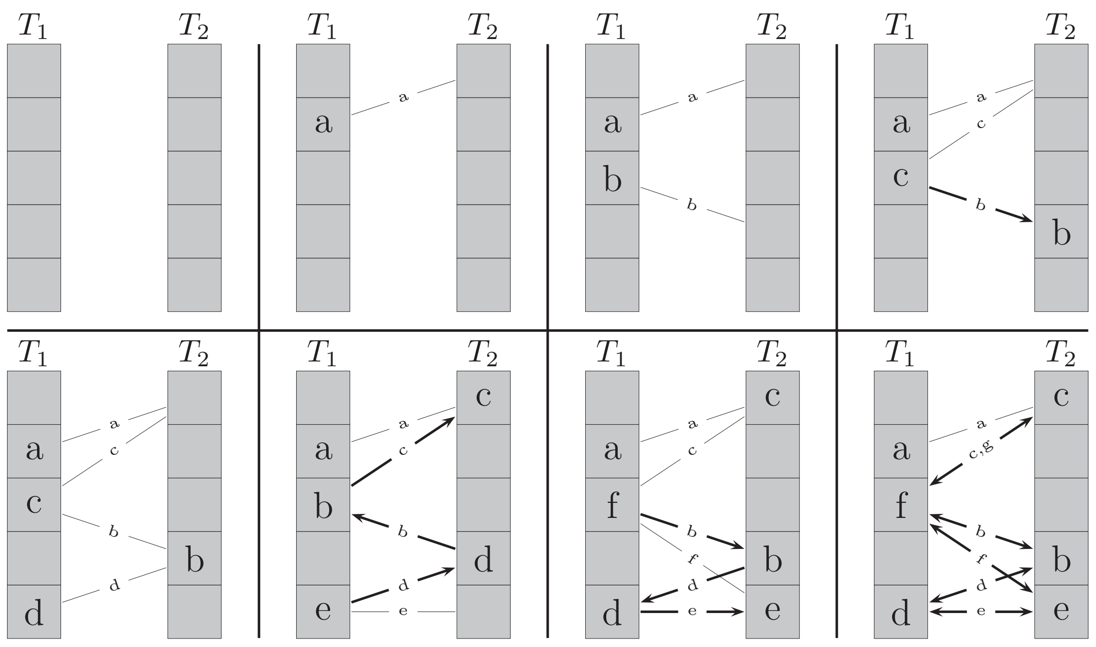


Figure 1.4: An evolving cuckoo hash table. We insert the keys a to f sequentially into the previously empty data structure. Each picture depicts the status after the insertion of a single key. The lines connect the two storage locations of a key. Thus, they indicate the values of the hash functions. Arrows symbolise the movement of a key, if it has been kicked-out during the last insertion. Finally, we try to insert the key g on the middle position of T_1, which causes and endless loop and therefore is impossible.

Due to this observation, it is a straightforward suggestion to increase the size of the first hash table compared to the second table. We expect that the number of keys stored in T_1 increases as the asymmetry increases, and hence we presume a better performance of successful search operations. On the other hand, one has to examine the influence of the asymmetry on the failure probability.

This modification was first mentioned in Pagh and Rodler [2004], but without a further analysis. A detailed study of the influence of this alteration can be found in Chapters 4, 7, and 8, see also Kutzelnigg [2008].

1.5.3 d-ary cuckoo hashing

A straightforward generalisation of cuckoo hashing was supposed by Fotakis et al. [2003] (see also Fotakis et al. [2005]). The algorithm uses d tables and d hash functions instead of only two. As a strong point, this algorithm allows a higher space utilisation than the standard data structure. For instance, the maximum load factor increases to 91% if $d = 3$ holds, to 97% for $d = 4$, and to 99% if $d = 5$ is satisfied. On the other hand, the cost of search operations increases as d increases. The original paper considers two slightly different versions of the algorithm. One might either restrict the access of a hash function to one table only, similar to the original cuckoo hash algorithm, or grant all hash function access to all memory cells.

Both variants can be analysed by considering matchings (see, *e.g.*, Diestel [2005]) in

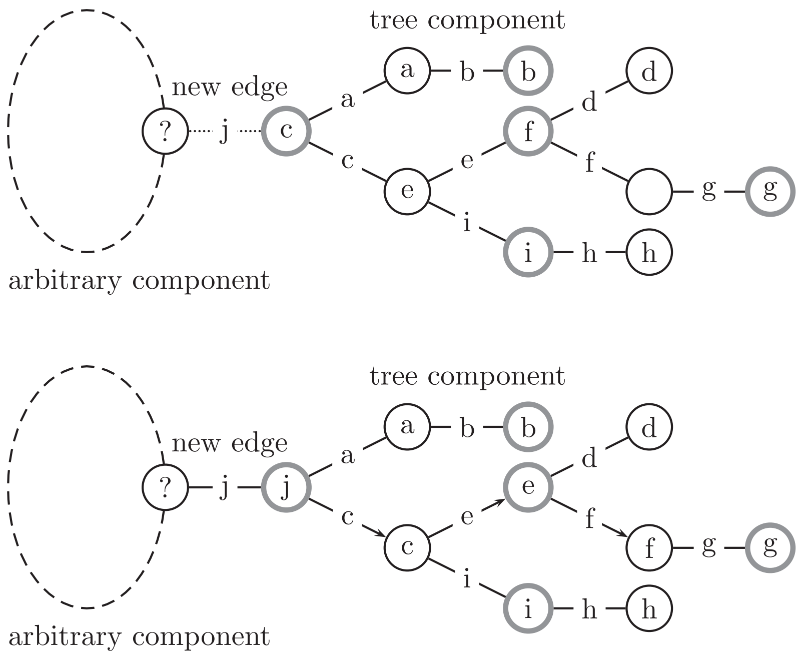


Figure 1.5: The insertion of a key possessing a primary storage location contained in a tree component. The first picture displays an exemplary situation immediately before the insertion of the key j, the second picture shows the situation after the insertion is completed.

bipartite graphs. The first type of nodes corresponds to the keys, while the second type corresponds to the memory cells. Now, for each node of first type, we select d nodes of second type uniformly at random in the allowed range and connect them to the vertex representing the key. The algorithm succeeds if and only if there exists a matching that covers all nodes corresponding to keys.

Unfortunately, there doesn't exist a direct analogon to the cuckoo graph that provides an easy criterion if the construction of the hash table is possible or not, even for the case $d = 3$. Of course, it is possible to consider hypergraphs (see, *e.g.*, Diestel [2005]) where each hyperedge corresponds to a key. However, it is unclear what components of this hypergraph are admissible for d-ary cuckoo hashing. For instance, it is straightforward to construct "bicyclic" components (*cf.* Andriamampianina and Ravelomanana [2005]) that do not produce conflicts. Some further results about d-ary cuckoo hashing are given in Czyzowicz et al. [2006], but until now, no exact analysis is known.

1.5.4 Simplified cuckoo hashing

A further simple modification of the standard algorithm is mentioned in Pagh and Rodler [2004], but again, without further analysis. Instead of using two separate tables, we "glue" them together and use one table of double size $2m$ only. Further, both hash functions address the whole table. This in some sense simplified algorithm is called simplified cuckoo hashing. As a result of this change, the probability that the first hash function hits an empty cell increases, hence we expect a better performance for search and insertion operations. Details will be discussed later, see also Drmota and Kutzelnigg [2008] and

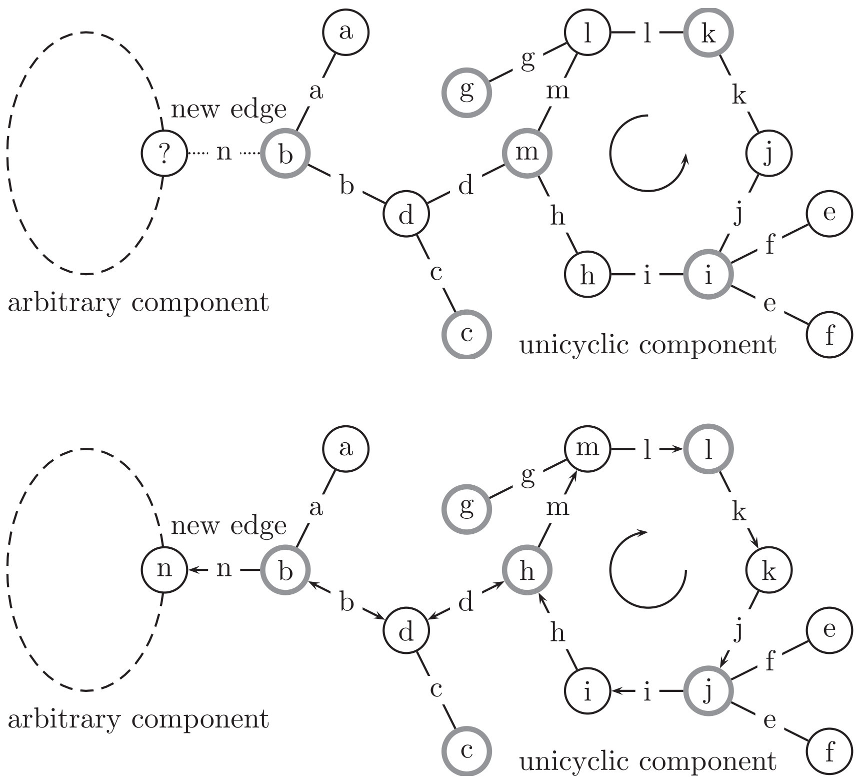


Figure 1.6: The insertion of a key possessing a primary storage position contained in an unicyclic component. The situation is displayed immediately before the insertion of key n starts and after the procedure finally places n in its secondary storage position. Note that the insertion is not yet complete if that position was previously occupied.

Kutzelnigg [2008].

As mentioned above, a similar suggestion was made in the analysis of d-ary cuckoo hashing by Fotakis et al. [2003]. However, the authors made this suggestion, because it simplifies the analysis, but they did not suggest it for practical application, due to the following problem: Given an occupied table position, we do not know any longer if this position is the primary or secondary storage position of the key. As a solution, we must either reevaluate a hash function, or preferably provide additional memory to store this information. It might be even possible to store this information in the table itself if the bit-length of a key is smaller than the length of the data type in use. We could for instance encode the number of the hash function in use in the sign of an entry.

Furthermore, a very clever variant to overcome this problem if only two hash functions are used is given in Pagh and Rodler [2004]. If we change the possible storage locations in a table of size $2m$ for a key x to be $h_1(x)$ and $(h_2(x) - h_1(x)) \mod 2m$, the alternative location of a key y stored at position i equals $(h_2(y) - i) \mod 2m$. For this reason, we assume that the second suggestion is implemented, and we do not take the cost of otherwise necessary reevaluations of hash functions into account.

Again, we model the algorithm by using a labelled multigraph, but this time we consider a non-bipartite graph possessing directed edges. Its labelled nodes represent the memory cells of the hash table, and each labelled edge represents a key x and connects $h_1(x)$

to $h_2(x)$. Further, we chose the direction of an edge such that it starts at the primary storage location $h_1(x)$. This construction is similar to the bipartite cuckoo graph of the standard algorithm described above. Once again, it is obviously necessary and sufficient for a successful construction, that every component of the cuckoo graph has less or equal edges than nodes. Thus, each connected component of the graph must either bee a tree, or unicyclic, see Chapter 4.

1.5.5 Cuckoo hashing with a stash

Another drawback of the previously mentioned algorithms based on cuckoo hashing is, that there exists a small (*cf.* Chapter 4) but practically relevant probability that the construction of the data structure fails, due to an endless loop in the insertion procedure. In this case, the standard solution is to rebuild the complete data structure, which is computationally very expensive. To overcome this weak point, Kirsch et al. [2008] suggest the usage of additional memory, the so called stash, to store keys that cannot be placed in the cuckoo hash table itself. Hereby, this stash is supposed to be a simple array, but one might also use a further hash table. Their analysis shows, that a small constant sized amount of additional memory is sufficient to reduce the failure probability dramatically.

Note that this modification has a strong influence on the number of memory cells accessed during an unsuccessful search, because all elements stored in the stash have to be inspected. Hence the performance measured in memory accesses decreases significantly (*cf.* Chapter 8), even if there is one key in the stash only. However this is an unfair comparison because of the memory system of modern computers. Since the stash is frequently accessed it will therefore be hold in the cache, in contrast to a randomly selected cell of a large table. Thus it is usually much faster to access keys contained in the stash.

1.5.6 Further variants

Naor et al. [2008] presented a modified version of the algorithm, that is history independent. In other words, the state of this data structure is not influenced by the specific order of the keys that lead to its current contents. This property is very important for applications where an unintended leak might reveal a point of attack, like cryptographic algorithms. The main idea of this modification is to give specific rules where the minimum key belongs to a tree resp. cycle is stored. Hence the position of all other keys belonging to the same component is determined.

A further modification of the standard algorithm is a load balancing data structure. Instead of always using the first hash function as starting point, one could randomly select the hash function. However, this modification results in increasing costs of successful search operations, since the number of keys stored in the first table decreases. Furthermore, an unsuccessful search takes two steps in any case, because the search can not be stopped any longer if the first probe hits an empty cell. Note that the balanced load does not influence the probability of a successful construction of the hash table, because it does not have influence on the related cuckoo graph. By mentioning all this reasons, we do not consider this variant any further.

Dietzfelbinger and Weidling [2007] suggested the usage of tables such that each table position represents a bucket of capacity d satisfying $d \geq 1$. Clearly, this modification increases the expected number of keys that can be stored in the table. But again, there is no direct analogon to the cuckoo graph known that provides an easy criterion if the construction of the hash table is possible or not, even for the case $d = 2$.

1.6 Hash functions

In principle, it is not required that keys are numeric, they may be alphanumeric or more general finite words based on an arbitrary finite alphabet. But it is straightforward (see, *e.g.*, Cormen et al. [2001]) to transform such keys into a numeric version, thus we consider numeric keys only.

1.6.1 The division method

The division method is an especially easy approach. We simply use the remainder modulo the table size,

$$h(x) = x \mod m. \tag{1.1}$$

In general, it is recommendable to choose m as prime number not to close too an exact power of two. This kind of hash functions is well suited for a software implementation of the classical hash algorithms, see Cormen et al. [2001].

1.6.2 The multiplication method

This method is equally approachable than the last one. Given a fixed real number A in the range $0 < A < 1$, we compute the fractional part of the product of A and the key x and multiply this by the table size. In short, we obtain

$$h(x) = \lfloor m\{xA\}\rfloor. \tag{1.2}$$

In contrast to the division method, the value of m is not critical. Although each real number of the interval $(0, 1)$ might be used as multiplicator, some offer better performance than other. For instance, the number $\psi^{-1} = (\sqrt{5} - 1)/2$ works very well (see Knuth [1998]).

1.6.3 Universal classes of hash functions

A weak point of standard hash algorithms is, that a any fixed hash function is inefficient for some sets of keys. In practise, the distribution according to which the keys are drawn is often unknown. Although the hash functions mentioned above behave well on uniform random input, some pattern in the input might lead to an increased number of collisions.

To overcome this bottleneck, Carter and Wegman [1979] introduced the concept of universal hashing. Instead of using a fixed hash function, we select a hash function at random (independent on the set of keys) from a accurately designed set of functions for each run of the algorithm. Due to this randomisation, the method offers good average performance on any input.

Definition 1.1. Let c be a real number greater or equal one. A set of hash functions H with domain U and range m is said to be c-universal, if for any distinct keys x and y the relation

$$|\{h \in H : h(x) = h(y)\}| \leq c\frac{|H|}{m} \tag{1.3}$$

holds. In particular, a 1-universal class is called universal.

Suppose that a hash function is chosen from a universal family uniformly at random and independent on the actual keys. Thus, we obtain roughly the same complexity for hashing with chaining as we used in Section 1.3 for random hash values (see, *e.g.*, Cormen et al. [2001]).

In particular, Carter and Wegman [1979] introduced the universal class of hash functions, whose members h are constructed as follows. Assume that a key s consists of a sequence $s_{r-1}, s_{r-2}, \ldots, s_0$ of length r of numbers satisfying $0 < s_i < b$. Further, suppose that f denotes an array of t-bit random integers $f[0], f[1], \ldots, f[b{\cdot}r]$. Then, we obtain a hash function, suitable for a table of size 2^t by the definition

$$h_f(s) = (f[s_0] \oplus f[s_0 + s_1 + 1] \oplus \cdots \oplus f[s_0 + \cdots + s_{r-1} + r - 1]), \tag{1.4}$$

where $\oplus$ is the bitwise exclusive or operator. Further, the evaluation of the members of this class is possible without multiplications. Thus, it is also suitable for hardware implementation. However, the weak point of the method is that the size of the random array might be very large especially for strings.

Another class of universal hash functions, that offers quite fast evaluation, can be found in Dietzfelbinger et al. [1997]. It consists of all the functions

$$h_a(x) = \left\lfloor (ax \mod 2^k)/2^{k-l} \right\rfloor, \tag{1.5}$$

that map $\{0, 1, \ldots, 2^k - 1\}$ to $\{0, 1, \ldots, 2^l - 1\}$, where $l \leq k$ holds and a is an odd constant satisfying $0 < a < 2^k$.

Several generalisations of the concept of universal hashing like (c, k) independent hash functions are known in the literature (see, *e.g.*, Siegel [1989]).

Definition 1.2. A family of hash functions H with domain U and range m is called (c, k)-independent, if for any distinct keys $x_1, \ldots, x_k$ and for all hash values $y_1, \ldots, y_k$ the relation

$$|\{h \in H : h(x_i) = y_i, \forall\, i = 1, \ldots, k\}| \leq c\frac{|H|}{m^k} \tag{1.6}$$

holds.

Clearly, a (c, k)-independent class is (c, l)-independent for all l smaller than k too. Further, each $(c, 2)$-independent class is c-universal.

An important example of independent hash functions are polynomial hash functions. Let U be a finite field and $a_d, a_{d-1}, \ldots, a_0$ be a sequence of elements of U. Then, we define the members of the class by

$$h(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0 \mod m, \tag{1.7}$$

where the multiplications and additions are carried out in U, see Dietzfelbinger et al. [1992] for further details.

1.6.4 Practical hash functions for cuckoo hashing

The original paper of Pagh and Rodler [2004] is based on the usage a universal class of hash functions introduced by Siegel [1989]. However, this functions exhibit a large evaluation time (but constant with respect to the table size), hence they are more of theoretical than of practical interest. Thus, the authors used different hash functions to obtain numerical data, although it is not clear whether their analysis carries through for this classes. Further, they noticed, that cuckoo hashing is rather sensitive to the choice of the hash function. They observed, that the exclusive or conjunction of three independently chosen members of the class defined by equation (1.5) works well.

Dietzfelbinger and Woelfel [2003] suggest a much more practical family of hash functions, based on polynomial hash functions, to replace Siegel's universal class. However, this new family requires still large additional storage space (memory for at least $\sqrt{m}$ numbers of the range $1, 2, \ldots, m$), and is a lot more complicate than the simple hash functions used for attaining numerical data in Pagh and Rodler [2004].

We considered several simple hash functions for the usage in all versions of cuckoo hashing. First, assume that our keys are 32-bit integer numbers. Our numerical experiments show, that functions of the form

$$ax + b \mod m, \tag{1.8}$$

are suitable for table sizes up to approximately 10^5, where a and b are random 32-bit numbers, m is a prime number, and the multiplication is performed without taking care of the overflow. Larger tables require polynomials of higher order, like the function

$$\big(ax^2 + bx + c \mod u\big) \mod m, \tag{1.9}$$

where u denotes a prime number much larger than m. Interestingly, introducing the additional calculation $\mod u$ in (1.8) did not increase the performance of the function in our experiments, however it is necessary in (1.9) for large tables. Further, the usage of 64-bit data types for the calculation of intermediate results did not have significant influence. We might also use the members of Carter and Wegman's universal class (1.4), this functions seem to work well for tables of all sizes. See also Kutzelnigg [2008] and Chapter 9.

If cuckoo hashing is used for hashing character strings, Tran and Kittitornkun [2007] suggested the usage of a class of hash functions introduced by Ramakrishna and Zobel [1997]. The functions are based on a conjunction of shift, addition, and exclusive or operations. Let S_L^a denote a bitwise left shift by a positions and S_R^b a bitwise right shift by b positions. Further let the key consist of the characters $c_1, \ldots, c_k$. Starting with a random initial block h_0, we apply the recursive definition

$$h_{i+1} = h_i \oplus (S_L^a(h_i) + S_R^b(h_i) + c_{i+1}), \tag{1.10}$$

till we obtain the hash value h_k.

Chapter 2

Mathematical Preliminaries

2.1 Generating Functions

The analysis of the cuckoo graph presented in this thesis is based on a generating function approach. Hence, we present here the most important properties for convenience of the reader. Further details, references, and omitted proofs can *e.g.* be found in Flajolet and Sedgewick [2001], Flajolet and Sedgewick [2008], Goulden and Jackson [1983], or Wilf [1999].

Definition 2.1. An unlabelled combinatorial configuration (S, w) consists of an at most countable set S and a size function w, such that the size of each element is a non-negative integer, and the number of elements of any given size is finite.
Further, a combinatorial configuration is called labelled if additionally each object of size n is linked to a permutation of the numbers $1, 2, \dots, n$. That means, each object consists in some way of n "atoms" and each of it is assigned a unique label in the range $1, 2, \dots, n$.

Definition 2.2 (Ordinary generating function)**.** The ordinary generating function of an unlabelled combinatorial configuration (S, w) is defined as the formal power series

$$S(x) = \sum_{s \in S} x^{w(s)}. \tag{2.1}$$

Further, the ordinary generating function of an infinite series $(a_n)_{n \geq 0}$ of complex numbers is given by

$$A(x) = \sum_{n \geq 0} a_n x^n. \tag{2.2}$$

The notation $[x^n]A(x)$ is henceforth used to refer to the coefficient a_n.

It is convenient to use generating functions, because basic constructions on combinatorial configurations can be translated to fundamental operations on the corresponding generating functions, see, *e.g.*, Flajolet and Sedgewick [2008]. For instance, this includes the union of disjoint combinatorial structures provided the size of the elements remains unchanged. A further example is given by the Cartesian product of configurations, if the size of a pair of objects is defined as sum of the individual sizes. Table 2.1 provides a survey over the most important constructions admissible for ordinary generating functions.

sum	$C = A + B$	$C(x) = A(x) + B(x)$
product	$C = A \times B$	$C(x) = A(x)B(x)$
sequence	$C = \mathsf{Seq}(A) = \{\varepsilon\} + A + A{\times}A + A{\times}A{\times}A + \dots$	$C(x) = \frac{1}{1-A(x)}$
compos.	$C = A(B)$	$C(x) = A(B(x)))$

Table 2.1: A basic "dictionary" of constructions useful to unlabelled combinatorial configurations, and their "translation" into ordinary generating functions.

sum	$C = A + B$	$C(x) = A(x) + B(x)$
product	$C = A * B$	$C(x) = A(x)B(x)$
sequence	$C = \mathsf{Seq}(A) = \{\varepsilon\} + A + A{*}A + A{*}A{*}A + \dots$	$C(x) = \frac{1}{1-A(x)}$
cycle	$C = \mathsf{Cyc}(A) = A + \frac{1}{2}A{*}A + \frac{1}{3}A{*}A{*}A + \dots$	$C(x) = \log \frac{1}{1-A(x)}$
set	$C = \mathsf{Set}(A) = \{\varepsilon\} + A + \frac{1}{2!}A{*}A + \frac{1}{3!}A{*}A{*}A + \dots$	$C(x) = \exp(A(x))$
compos.	$C = A(B)$	$C(x) = A(B(x))$

Table 2.2: A "dictionary" of constructions useful to labelled combinatorial configurations, and their "translation" into exponential generating functions.

Definition 2.3 (Exponential generating function)**.** The exponential generating function of a labelled combinatorial configuration (S, w) is defined as the formal power series

$$S(x) = \sum_{s \in S} \frac{x^{w(s)}}{w(s)!}. \tag{2.3}$$

Further, the exponential generating function of an infinite series $(a_n)_{n\geq 0}$ of complex numbers is given by

$$A(x) = \sum_{n \geq 0} a_n \frac{x^n}{n!}. \tag{2.4}$$

Similar to the ordinary case, there exist likewise translations of operations performed on labelled structures to the language of exponential generating functions. However, combining tagged structures might require a relabelling. The product of two configurations A and B is still built using the Cartesian product, but one has to perform all order-consistent relabellings. Thus a pair (a, b), featuring sizes $w_A(a)$ respectively $w_B(b)$, produces $\binom{w_A(a)+w_B(b)}{w_A(a)}$ different tagged elements. Table 2.2 provides an overview of combinatorial constructions applicable to labelled combinatorial configurations.

In general, we do not strictly distinguish between exponential and ordinary generating functions, because the particular type is usually unambiguous in the current context. The following theorem provides a tool that is especially useful if we are faced with the task to extract coefficients from generating functions that are implicitly obtained trough functional equations.

Theorem 2.1 (Lagrange Inversion Theorem)**.** *Let $A(x) = \sum_{n\geq 0} a_n x^n$ be a generating function that satisfies the functional equation $A(x) = x\phi(A(x))$ such that $\phi(0) \neq 0$ holds. Then the equation*

$$[x^n]g(A(x)) = \frac{1}{n}[u^{n-1}]g'(u)\phi(u)^n \tag{2.5}$$

holds for all $n \geq 1$.

Within the theory of analysis of algorithms, it is usually of interest to calculate values of parameters related to the considered algorithm, instead of just counting structures. For instance, knowledge concerning the size of the tree components of the cuckoo graph allows us to estimate the construction cost of the hash table (see Chapter 6 resp. 7). Such information is usually obtained introducing a further variable marking the parameter of interest, that leads to bivariate or more general multivariate generating functions. Using this functions, one can usually obtain information on the distribution of the observed parameter, such as expectation, variance, or even limit laws.

Given an either labelled or unlabelled combinatorial configuration (S, w), we define a d-dimensional parameter χ as a function mapping the elements of S on a d dimensional vector of natural numbers and consider the problem of counting all elements s contained in S that satisfy the relation

$$w(s) = n, \chi(s) = (k_1, \dots, k_n). \tag{2.6}$$

Definition 2.4. Let $(a_{n,\mathbf{k}})$ be a multi-index sequence of complex numbers, where $\mathbf{k}$ denotes a d dimensional vector of natural numbers. Then the ordinary multivariate generating function of this sequence is given by

$$A(x) = \sum_{n,\mathbf{k}} a_{n,\mathbf{k}} x^n \mathbf{z}^{\mathbf{k}}. \tag{2.7}$$

The exponential multivariate generating function is defined as

$$A(x) = \sum_{n,\mathbf{k}} a_{n,\mathbf{k}} \frac{x^n}{n!} \mathbf{z}^{\mathbf{k}}. \tag{2.8}$$

Note that it is possible to adopt the constructions given in Table 2.1 respectively 2.2 to this general situation under certain circumstances, depending on the interpretation of the parameter. More precisely, this is possible if the parameter is "inherited", *cf.* Flajolet and Sedgewick [2008]. That means, the value of the parameter is carried forward unchanged in case of union operations, and it is attained additively as the sum of the value of the parameters of all involved objects in case of a Cartesian product.

Moreover, modelling the bipartite cuckoo graph requires at least a "double exponential" generating function, because the nodes of both types are tagged independently. Nonetheless, the constructions given in Table 2.2 are still applicable. Consider for instance the product of the functions $f(x, y)$ and $g(x, y)$,

$$\left(\sum_{n,m\ge 0} f_{n,m} \frac{x^n y^m}{n!m!}\right)\left(\sum_{n,m\ge 0} g_{n,m} \frac{x^n y^m}{n!m!}\right) = \sum_{n,m\ge 0} \frac{x^n y^m}{n!m!} \sum_{k=0}^{n}\sum_{l=0}^{m} \binom{n}{k}\binom{m}{l} f_{k,l}\, g_{n-k,m-l}. \tag{2.9}$$

Note that the inner double sum of the right hand side of this equation takes all possible independent order preserving relabellings into account.

2.2 Convergence in Distribution

In this thesis, we assume that the reader is familiar with the theory of probability. Nevertheless, we briefly introduce the concept of weak convergence, that is used in Chapter 6.

Further details, references, and omitted proofs can for instance be found in Drmota and Soria [1995], Flajolet and Sedgewick [2008], Hwang [1996], Loéve [1977], and Papoulis and Pillai [2002].

Definition 2.5. Let F_n be a family of distribution functions, such that

$$\lim_{n\to\infty} F_n(x) = F(x) \tag{2.10}$$

holds pointwise for all continuity points x of a distribution function F. Then, we say that F_n converges weakly to F. Further, let X_n resp. X denote random variables having distribution functions F_n resp. F. Then, we say that X_n converges in distribution to X.

Definition 2.6. The characteristic function of a random variable X, having distribution function F, is defined by

$$\phi(r) = \mathbb{E}\left(e^{irX}\right) = \int\limits_{-\infty}^{\infty} e^{irx} dF(x). \tag{2.11}$$

Further, its moment generating function (also known as Laplace transform) is given by

$$\psi(r) = \mathbb{E}\left(e^{rX}\right) = \int\limits_{-\infty}^{\infty} e^{rx} dF(x). \tag{2.12}$$

The importance of this transformations is based on the fact, that convergence of distributions is related to the convergence of transforms. In particular, the following results hold:

Theorem 2.2. *Let X_n and X be random variables with characteristic functions ϕ_n and ϕ. Then, X_n converges in distribution to X if and only if*

$$\lim_{n\to\infty} \phi_n(r) = \phi(r) \tag{2.13}$$

holds pointwise for each real number r.

Theorem 2.3. *Let X_n be a sequence of random variables, and denote the corresponding moment generating functions by ψ_n. Assume that these functions exist in an Interval $I = [-a, a]$, such that $a > 0$ holds, and that*

$$\lim_{n\to\infty} \psi_n(r) = \psi(r) \tag{2.14}$$

holds pointwise for all $r \in I$. Then, X_n converges in distribution to a random variable X with moment generating function ψ.

Chapter 3

Saddle Point Asymptotics

Like a lazy hiker, the path crosses the ridge at a low point;
but unlike the hiker, the best path takes the steepest ascent to the ridge.
[. . .] The integral will then be concentrated in a small interval.

Greene and Knuth [1982]

3.1 Introduction

A saddle point of a function represents a point that is similar to the inner part of a riding saddle or like a mountain pass. It can be interpreted as the simplest way through a range. For an arbitrary surface representing the modulus of an analytic function, the saddle points are the non vanishing zeros of the derivative of this function.

Given an analytic function with nonnegative coefficients, the saddle point method enables us to investigate the asymptotic behaviour of the coefficients. Especially, it is helpful if the function is free of singularities, such that other methods like singularity analysis of generating functions (see Flajolet and Sedgewick [2008]) can not be applied.

Now, we proceed as follows to calculate an asymptotic approximation of the coefficient of x^n of a function $f(x)$. First, we use Cauchy's Integral Formula

$$f_n = \frac{1}{2\pi i}\oint f(z)z^{-(n+1)}\,dz = \frac{1}{2\pi}\int_0^{2\pi} f(x_0e^{i\phi})x_0^{-n}e^{-in\phi}\,d\phi, \tag{3.1}$$

(see, *e.g.*, Conway [1978]) and obtain an equation involving a line integral. Next, we choose the line of integration such that it passes a saddle point, where the function falls of suddenly on leaving this point. Thus, the saddle point corresponds locally to a maximum of the function on the chosen path. Since we requested nonnegative coefficients, the relation $|f(x)| \le f(|x|)$ holds. Hence, we concentrate the search for a saddle point on the real line. It is easy to see, that the derivative of $\exp(\log f(x_0) - n\log x_0)$ equals zero if and only if the radius x_0 is chosen such that

$$x_0\frac{f'(x_0)}{f(x_0)} = n \tag{3.2}$$

holds. Once this path of integration is defined, the integral is evaluated with help of Laplace's method (*cf.* Flajolet and Sedgewick [2008], Greene and Knuth [1982]). More precisely, the integral is concentrated on a small interval around the saddle point. We perform the following steps.

1. *Neglecting the tail integrals:* Choose α depending on n such that the contribution to the integral outside this arc is negligible.

$$\int\limits_{\alpha<|\phi|<\pi} f\big(x_0e^{i\phi}\big)x_0^{-n}e^{-in\phi}\,d\phi = o\left(\int\limits_{-\alpha}^{\alpha} f\big(x_0e^{i\phi}\big)x_0^{-n}e^{-in\phi}\,d\phi\right). \tag{3.3}$$

   Clearly, it is required that the saddle point corresponds to the only argument producing the maximum modulus on the path of integration. Note that the next two steps have influence on the choice of α too.

2. *Centrally approximating the integrand:* Let $h(\phi)$ denote $\log f(x_0e^{i\phi})$. We use of the local expansion

$$f\big(x_0e^{i\phi}\big)x_0^{-n}e^{-in\phi} = f(x_0)e^{h''(\phi)\phi^2+\mathcal{O}\left(h'''(\phi)\alpha^3\right)}. \tag{3.4}$$

   The angle α is required to be sufficiently *small*, such that the relation $h'''(\phi)\alpha^3 \to 0$ holds uniformly for all ϕ satisfying $|\phi| \le \alpha$ as n tends to infinity. Hence, we obtain the relation

$$\int\limits_{-\alpha}^{\alpha} f\big(x_0e^{i\phi}\big)x_0^{-n}e^{-in\phi}\,d\phi = f(x_0)\int\limits_{-\alpha}^{\alpha} e^{h''(\phi)\phi^2}d\phi + o(1). \tag{3.5}$$

3. *Completing the tails:* Finally, we require that α is *large* enough such that $-h''(\phi)\alpha^2$ tends to infinity, thus the relation

$$\int\limits_{-\alpha}^{\alpha} e^{h''(\phi)\phi^2}d\phi = \int\limits_{-\infty}^{\infty} e^{h''(\phi)\phi^2}d\phi + o(1) \tag{3.6}$$

   holds.

Further results and examples are *e.g.* given in Drmota [1994], Flajolet and Sedgewick [2008], Gardy [1995], and Greene and Knuth [1982].

3.2 Stirling's formula using saddle point method

The aim of this section is a detailed derivation of Stirling's formula by using the saddle point method (*cf.* Flajolet and Sedgewick [2008]) to provide an introductory example of this powerful technique. Further, we will show how complete asymptotic expansions can be obtained.

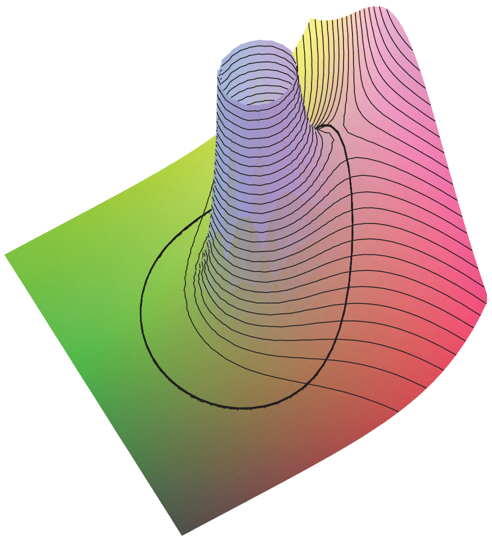


Figure 3.1: The modulus of the function e^z/z^3 close to the origin.

3.2.1 The leading term

We consider the function e^z and obtain using Cauchy's Integral Formula the equation

$$\frac{1}{n!} = \frac{1}{2\pi i}\oint e^z z^{-(n+1)}\,dz = \frac{1}{2\pi}\int_{-\pi}^{\pi} \exp\left(x_0 e^{i\phi}\right) x_0^{-n} e^{-in\phi}\,d\phi. \tag{3.7}$$

The saddle point is determined solving (3.2), and we obtain $x_0 = n$. Hence, we get the equation

$$\frac{1}{n!} = \frac{1}{2\pi}\frac{e^n}{n^n}\int_{-\pi}^{\pi} e^{nh(\phi)}d\phi, \tag{3.8}$$

where $h(\phi)$ denotes the function $e^{i\phi} - 1 - i\phi$. Figure 3.1 shows the surface and the path passing the saddle point according to $n = 3$. Note that the rate of ascent increases if n is increasing.

We set $\alpha = n^{-1/2+\varepsilon}$, where ε denotes a real number satisfying $0 < \varepsilon < 1/6$. Therefore $n\alpha^2 \to \infty$ and $n\alpha^3 \to 0$ hold, according to the heuristic mentioned in the last section. Furthermore, the modulus of $e^{h(\phi)}$ is given by $e^{\cos\phi - 1}$. We conclude that the contribution outside the arc defined by α is exponentially small,

$$\left|\int_{\alpha<|\phi|\le\pi} e^{nh(\phi)}d\phi\right| \le 2\pi e^{-n(1-cos\alpha)} = \mathcal{O}\left(e^{-Cn\alpha^2}\right) = \mathcal{O}\left(e^{-Cn^{\varepsilon}}\right). \tag{3.9}$$

Next, we obtain a Taylor expansion of $h(\phi)$ near the saddle point. More precisely, uniformly for all ϕ in $[-\alpha, \alpha]$, the relation

$$h(\phi) = -\frac{\phi^2}{2} - i\int_0^{\phi}\frac{h'''(t)}{2}(\phi - t)^2\,dt = -\frac{\phi^2}{2} + \mathcal{O}(\alpha^3) \tag{3.10}$$

holds. With help of this expansion, we can rewrite the integral that provides the main contribution as follows,

$$\int_{-\alpha}^{\alpha} e^{nh(\phi)} d\phi = \int_{-\alpha}^{\alpha} e^{-\frac{n\phi^2}{2}+\mathcal{O}(n\alpha^3)} d\phi = \int_{-\alpha}^{\alpha} e^{-\frac{n\phi^2}{2}} \left(1+\mathcal{O}(n\alpha^3)\right) d\phi. \tag{3.11}$$

We split up this integral and proceed with an estimate of the summand containing the big-$\mathcal{O}$ term,

$$\left|\int_{-\alpha}^{\alpha} e^{-\frac{n\phi^2}{2}} \mathcal{O}(n\alpha^3)\, d\phi\right| = \mathcal{O}(n\alpha^3) \int_{-\alpha}^{\alpha} e^{-\frac{n\phi^2}{2}} d\phi \leq \frac{\mathcal{O}(n\alpha^3)}{\sqrt{n}} \int_{-\infty}^{\infty} e^{-\frac{\psi^2}{2}} d\psi = \mathcal{O}\left(n^{-1+3\varepsilon}\right). \tag{3.12}$$

Afterwards, we calculate the remaining integral by extending the range of the integral to infinity. This step is justified by the fact that the tails of the integral are exponentially small (*cf.* Lemma 3.2),

$$\frac{1}{\sqrt{n}} \int_{\alpha\sqrt{n}}^{\infty} e^{-\frac{\psi^2}{2}} d\psi = \frac{1}{\sqrt{n}} \int_{0}^{\infty} e^{-\frac{1}{2}\left(\alpha\sqrt{n}+h\right)^2} dh \leq \frac{e^{-\frac{\alpha^2 n}{2}}}{\sqrt{n}} \int_{0}^{\infty} e^{-\frac{h^2}{2}} dh = \mathcal{O}\left(e^{-Cn^{\varepsilon}}\right). \tag{3.13}$$

Thus we obtain the main contribution

$$\int_{-\alpha}^{\alpha} e^{-\frac{n\phi^2}{2}} d\phi = \frac{1}{\sqrt{n}} \int_{-\infty}^{\infty} e^{-\frac{\psi^2}{2}} d\psi + \mathcal{O}\left(e^{-Cn^{\varepsilon}}\right) = \frac{\sqrt{2\pi}}{\sqrt{n}} + \mathcal{O}\left(e^{-Cn^{\varepsilon}}\right). \tag{3.14}$$

Finally, we collect all the results and obtain

$$\frac{1}{n!} = \frac{1}{2\pi} \frac{e^n}{n^n} \left(\frac{\sqrt{2\pi}}{\sqrt{n}} + \mathcal{O}\left(n^{-1+3\varepsilon}\right) + \mathcal{O}\left(e^{-Cn^{\varepsilon}}\right)\right) = \frac{1}{\sqrt{2\pi n}} \frac{e^n}{n^n} \left(1 + \mathcal{O}\left(n^{-\frac{1}{2}+3\varepsilon}\right)\right). \tag{3.15}$$

3.2.2 Further coefficients

A refinement of this method allows us to calculate complete asymptotic expansions. Since the integral taken outside the arc defined by α is exponentially small, we just have to revisit (3.11). We proceed with the refined Taylor expansion

$$nh(\phi) = nh\left(\frac{\psi}{\sqrt{n}}\right) = -\frac{\psi^2}{2} - i\frac{\psi^3}{6\sqrt{n}} + \frac{\psi^4}{24n} + R_4, \tag{3.16}$$

where the remainder R_4 is bounded uniformly for all ψ in $\left[-\alpha\sqrt{n}, \alpha\sqrt{n}\right]$ by

$$|R_4| = \left|\int_{0}^{\psi} \frac{e^{\frac{it}{\sqrt{n}}}}{n^{\frac{3}{2}} 4!} (\psi - t)^4\right| \leq \frac{|\psi|^5}{n^{\frac{3}{2}} 5!} = \mathcal{O}(n\alpha^5). \tag{3.17}$$

Further, we make use of the expansions

$$e^{-i\frac{\psi^3}{6\sqrt{n}}} = 1 - i\frac{\psi^3}{6\sqrt{n}} - \frac{\psi^6}{72n} + \int\limits_0^{\psi^3} \frac{e^{-\frac{it}{\sqrt{n}}}}{n^{\frac{3}{2}}2}(\psi^3 - t)^2 = 1 - i\frac{\psi^3}{6\sqrt{n}} - \frac{\psi^6}{72n} + \mathcal{O}\big(n^3\alpha^9\big) \quad (3.18)$$

and

$$e^{\frac{\psi^4}{24n}} = 1 + \frac{\psi^4}{24n} + \mathcal{O}\big(n^2\alpha^8\big). \quad (3.19)$$

Hence we yield

$$\int\limits_{-\alpha}^{\alpha} e^{nh(\phi)} d\phi = \frac{1}{\sqrt{n}} \int\limits_{-\alpha\sqrt{n}}^{\alpha\sqrt{n}} e^{-\frac{\psi^2}{2}} \left(1 - \frac{1}{\sqrt{n}}\frac{i\psi^3}{6} + \frac{1}{n}\left(\frac{\psi^4}{24} - \frac{\psi^6}{72}\right) + \mathcal{O}\big(n^3\alpha^9\big)\right) d\psi. \quad (3.20)$$

Again, we split up the integral and treat the summands separately. Similar to (3.12) the bound

$$\left|\int\limits_{-\alpha}^{\alpha} e^{-\frac{n\phi^2}{2}} \mathcal{O}(n^3\alpha^9)\, d\phi\right| = \mathcal{O}\left(n^{-2+9\varepsilon}\right) \quad (3.21)$$

holds. Finally, we complete the exponentially small tails and calculate the integral

$$\frac{1}{\sqrt{n}} \int\limits_{-\infty}^{\infty} e^{-\frac{\psi^2}{2}} \left(1 - \frac{1}{\sqrt{n}}\frac{i\psi^3}{6} + \frac{1}{n}\left(\frac{\psi^4}{24} - \frac{\psi^6}{72}\right)\right) d\psi = \frac{\sqrt{2\pi}}{\sqrt{n}}\left(1 + \left(\frac{1}{8} - \frac{5}{24}\right)\frac{1}{n}\right). \quad (3.22)$$

The evaluation of integrals of the form $\int_{-\infty}^{\infty} e^{-z^2/2} z^k\, dz$ is explained in Lemma 3.1. Additionally, the error terms caused by the completed tails are estimated in Lemma 3.2. Further terms of the expansion can be calculated in the same way, and the expansion

$$\frac{1}{n!} = \frac{1}{\sqrt{2\pi n}}\frac{e^n}{n^n}\left(1 - \frac{1}{12n} + \frac{1}{288n^2} + \frac{139}{51840n^3} - \frac{571}{2488320n^4} + \mathcal{O}\left(\frac{1}{n^5}\right)\right), \quad (3.23)$$

can be obtained.

Lemma 3.1. *Let l be a natural number.*

$$\int\limits_{-\infty}^{\infty} e^{-\frac{z^2}{2}} z^l\, dz = \begin{cases} 1\cdot 3\cdot 5\ldots(l-1)\sqrt{2\pi} & \ldots l \text{ even} \\ 0 & \ldots l \text{ odd} \end{cases} \quad (3.24)$$

Proof. Integration by parts shows us that the equation

$$\int\limits_{-\infty}^{\infty} e^{-\frac{z^2}{2}} z^l\, dz = \left.\frac{e^{-\frac{z^2}{2}} z^{l+1}}{l+1}\right|_{-\infty}^{\infty} + \frac{1}{l+1}\int\limits_{-\infty}^{\infty} e^{-\frac{z^2}{2}} z^{l+2}\, dz$$

holds. We already know from Gauß that the integral $\int_{-\infty}^{\infty} e^{-z^2/2}\, dz$ equals $\sqrt{2\pi}$ and obtain the result for the even case by induction. Now, assume l is odd. This implies that $e^{-z^2/2}z^l$ is an odd function. Further, the absolute value of the integral is bounded and therefore zero.

A different proof can be obtained by using the equation

$$\sum_{l\geq 0}\frac{x^l}{l!}\int_{-\infty}^{\infty} e^{-\frac{z^2}{2}} z^l\,dz = \int_{-\infty}^{\infty} e^{-\frac{z^2}{2}+zx-\frac{x^2}{2}+\frac{x^2}{2}}\,dz = e^{\frac{x^2}{2}}\sqrt{2\pi} = \sqrt{2\pi}\sum_{l=0}^{\infty}\frac{x^{2l}}{l!2^l}. \tag{3.25}$$

□

Lemma 3.2. *Let l be a natural number, w a real number greater than zero, and c be a constant in $(0, 1/2)$. Then, the bound*

$$\int_{w}^{\infty} e^{-\frac{z^2}{2}} z^l\,dz = \mathcal{O}\left(e^{-cw^2}\right) \tag{3.26}$$

holds.

Proof. We perform the substitution $z = h + w$. Thus, we obtain the equation

$$\int_{w}^{\infty} e^{-\frac{z^2}{2}} z^l\,dz = \int_{0}^{\infty}(h+w)^l e^{-\frac{(h+w)^2}{2}}\,dh \leq e^{-\frac{w^2}{2}}\sum_{i=0}^{l}\binom{l}{i}w^{l-i}\int_{0}^{\infty}h^i e^{-\frac{h^2}{2}}\,dh. \tag{3.27}$$

Since all the integrals are constant, we obtain the bound $p_l(w)e^{-w^2/2}$, where $p_l(w)$ denotes a polynomial of degree l with real coefficients. Clearly, $p_l(w)$ is in $\mathcal{O}(e^{\varepsilon w^2})$ for each arbitrarily small but fixed and positive ε. Hence, the claimed bound holds. □

3.3 Asymptotic expansions

In Chapter 4, we derive the generating function of the cuckoo graph and subsequently apply a saddle point method. This section provides the required method to extract an asymptotic expansion of the k-th coefficient $f_k = [x^k]f(x)$ of a suitable power series $f(x) = \sum_{k=0}^{\infty} f_k x^k$.

Theorem 3.1. *Let $f(x)$ and $g(x)$ be analytic functions with a positive radius of convergence R larger than x_0 as defined below, and let $m = ak$ hold for a positive constant a. Furthermore, let all coefficients of $f(x)$ and $g(x)$ be non negative and assume that the relations $f(0) \neq 0$ and $\gcd\{m|[x^m]f(x) > 0\} = 1$ hold. Then, the asymptotic series expansion*

$$[x^m]g(x)f(x)^k = \frac{g(x_0)f(x_0)^k}{x_0^m\sqrt{2\pi k\kappa_2}}\left(1 + \frac{H}{24\kappa_2^3}\frac{1}{k} + \mathcal{O}\left(\frac{1}{k^2}\right)\right), \tag{3.28}$$

is valid, where x_0 is uniquely defined by

$$\frac{m}{k} = \frac{x_0 f'(x_0)}{f(x_0)}. \tag{3.29}$$

Generally, let the cummulants κ_i and $\overline{\kappa}_i$ be

$$\kappa_i = \left[\frac{\partial^i}{\partial u^i}\log f(x_0e^u)\right]_{u=0}, \qquad \overline{\kappa}_i = \left[\frac{\partial^i}{\partial u^i}\log g(x_0e^u)\right]_{u=0}. \tag{3.30}$$

H can be expressed by

$$12\kappa_2\kappa_3\overline{\kappa}_1 + 3\kappa_2\kappa_4 - 12\kappa_2^2\overline{\kappa}_1^2 - 12\kappa_2^2\overline{\kappa}_2 - 5\kappa_3^2. \tag{3.31}$$

The leading term of this asymptotic expansion is already given in Gardy [1995], and the calculation of further terms is suggested as possible extension by the author. A detailed asymptotic expansion for $g = 1$ can be found in Drmota [1994].

Further coefficients can be calculated analogously, but the complexity of the explicit calculation increases for each further term. Here, we content ourselves with the result stated above, since it is adequate for the analysis of simplified cuckoo hashing.

In general, it is not necessary that the functions f and g fulfil all the strong technical conditions of the theorem, as long as the unique existence of the saddle point is granted. However this assumptions are highly natural if gf^k is a generating function counting some combinatorial objects. Further, one has to adjust the calculations, if the maximal value is attained more than once on the line of integration.

Proof. We start showing that the saddle point x_0 on the real line is unique if it exists. It suffices to check that $[xf'(x)/f(x)]_{x=0} = 0$ and that the function $xf'(x)/f(x)$ is continuous and increasing for real valued x. Since we required nonnegative coefficients, the relation $|f(x)| \leq f(|x|)$ holds. Thus it can be shown that f is logarithmically convex (see Conway [1978]). This implies that $(log f(x))' = f'(x)/f(x)$ is increasing.

Further, x_0 is the unique maximum of f on the line of integration. This is due to the fact that the maximum is attained only if all coefficients of f are aligned in the same phase. Hence it occurs as a consequence of the aperiodicity $(\gcd\{m|[x^m]f(x) > 0\} = 1)$ on the positive real line only.

Again, we use Cauchy's Formula as starting point:

$$[x^m]g(x)f(x)^k = \frac{1}{2\pi i} \int\limits_{|x|=x_0} \frac{g(x)f(x)^k}{x^{m+1}}\, dx = \frac{1}{2\pi i} \int\limits_{|x|=x_0} e^{k\log f(x)-(m+1)\log x+\log g(x)}\, dx. \tag{3.32}$$

We perform the substitution $x = x_0 e^{is}$, and obtain

$$[x^m]g(x)f(x)^k = \frac{1}{2\pi x_0^m} \int\limits_{-\pi}^{\pi} e^{k\log f\left(x_0e^{is}\right)-mis+\log g\left(x_0e^{is}\right)}\, ds. \tag{3.33}$$

The contribution outside a proper chosen interval $[-\alpha, \alpha]$ is exponentially small compared to the integral taken along this range. Similar to previous calculations, we choose α such that $k\alpha^2 \to \infty$ and $k\alpha^3 \to 0$ hold as k turns to infinity. To achieve this, we set $\alpha = k^{-1/2+\varepsilon}$, where ε denotes a positive number satisfying $\varepsilon < 1/6$. Hence we obtain the equation

$$e^{k\log f\left(x_0e^{is}\right)-mis} = f(x_0)^k e^{-k\kappa_2\frac{s^2}{2}+\mathcal{O}(k\alpha^3)}, \tag{3.34}$$

uniformly for all s in $[-\alpha, \alpha]$. Thereby, the bound implied by the big-$\mathcal{O}$ term depends on the maximum of the third derivative of $\log f(x_0e^{is})$ with respect to s taken over the interval $[-\alpha, \alpha]$. Since x_0 is the unique maximum of the modulus of f along the path of integration, it is clear that the modulus of the function outside the range $[-\alpha, \alpha]$ is bounded by the modulus taken on the boundary points of this interval, at least for sufficiently large k. Using the Taylor expansion derived above, we thus obtain the estimation

$$\left|\frac{1}{2\pi x_0^m} \int\limits_{\alpha<|s|\leq\pi} g(x_0e^{is}) e^{k\log f\left(x_0e^{is}\right)-mis}\, ds\right| \leq \frac{g(x_0)f(x_0)^k}{x_0^m}\mathcal{O}\left(e^{-ck\alpha^2}\right), \tag{3.35}$$

where c denotes a positive constant.

Next, we consider the remaining integral and substitute $u = \sqrt{k}s$. In what follows, we make use of the Taylor expansions listed below:

$$k \log f\left(x_0 e^{i\frac{u}{\sqrt{k}}}\right) - mi\frac{u}{\sqrt{k}} = k\log f(x_0) - \frac{\kappa_2 u^2}{2} + \frac{i\kappa_3 u^3}{6\sqrt{k}} + \frac{\kappa_4 u^4}{24k} + \mathcal{O}\left(k\alpha^5\right), \tag{3.36}$$

$$\log g\left(x_0 e^{i\frac{u}{\sqrt{k}}}\right) = \log g(x_0) + \frac{i\overline{\kappa}_1 u}{\sqrt{k}} - \frac{\overline{\kappa}_2 u^2}{2k} + \mathcal{O}\left(\alpha^3\right), \tag{3.37}$$

$$e^{-\frac{i\kappa_3 u^3}{6\sqrt{k}}} = 1 - \frac{i\kappa_3 u^3}{6\sqrt{k}} - \frac{\kappa_3^2 u^6}{72k} + \mathcal{O}\left(k^3\alpha^9\right) \tag{3.38}$$

$$e^{\frac{\kappa_4 u^4}{24k}} = 1 + \frac{\kappa_4 u^4}{24k} + \mathcal{O}\left(k^2\alpha^8\right), \tag{3.39}$$

$$e^{\frac{i\overline{\kappa}_1 u}{\sqrt{k}}} = 1 + \frac{i\overline{\kappa}_1 u}{\sqrt{k}} - \frac{\overline{\kappa}_1^2 u^2}{2k} + \mathcal{O}\left(\alpha^3\right), \tag{3.40}$$

$$e^{-\frac{\overline{\kappa}_2 u^2}{2k}} = 1 - \frac{\overline{\kappa}_2 u^2}{2k} + \mathcal{O}\left(\alpha^4\right). \tag{3.41}$$

Thus, the integral can be rewritten as

$$\begin{aligned}
&\frac{1}{2\pi x_0^m}\int_{-\alpha}^{\alpha} e^{k\log f\left(x_0e^{is}\right) - mis - \log g\left(x_0 e^{is}\right)}\,ds \\
&= \frac{g(x_0)f(x_0)^k}{2\pi x_0^m\sqrt{k}}\int_{-\alpha\sqrt{k}}^{\alpha\sqrt{k}} e^{-\frac{\kappa_2 u^2}{2}}\Bigg(1 - \frac{i\kappa_3 u^3}{6\sqrt{k}} + \frac{i\overline{\kappa}_1 u}{\sqrt{k}} + \frac{\kappa_3\overline{\kappa}_1 u^4}{6k} - \frac{\kappa_3^2 u^6}{72k} \\
&\qquad + \frac{\kappa_4 u^4}{24k} - \frac{\overline{\kappa}_1^2 u^2}{2k} - \frac{\overline{\kappa}_2 u^2}{2k} + \mathcal{O}\left(k^3\alpha^9\right)\Bigg)\,du.
\end{aligned} \tag{3.42}$$

First, we focus the analysis on the part of the integral that contains the error term. It is easy to see, that the modulus of this integral is bounded by

$$\left|\int_{-\alpha\sqrt{k}}^{\alpha\sqrt{k}} e^{-\frac{\kappa_2 u^2}{2}}\mathcal{O}\left(k^3\alpha^9\right)\,du\right| \le \mathcal{O}\left(k^3\alpha^9\right)\int_{-\infty}^{\infty} e^{-\frac{\kappa_2 u^2}{2}}\,du = \mathcal{O}\left(k^3\alpha^9\right) = \mathcal{O}\left(k^{-\frac{3}{2}+9\varepsilon}\right). \tag{3.43}$$

Second, we complete the tails, what causes exponentially small error terms (see also Lemma 3.2). This allows us to compute the integrals by using Lemma 3.1,

$$\begin{aligned}
&\int_{-\infty}^{\infty} e^{-\frac{\kappa_2 u^2}{2}}\left(1 - \frac{i\kappa_3 u^3}{6\sqrt{k}} + \frac{i\overline{\kappa}_1 u}{\sqrt{k}} + \frac{\kappa_3\overline{\kappa}_1 u^4}{6k} - \frac{\kappa_3^2 u^6}{72k} + \frac{\kappa_4 u^4}{24k} - \frac{\overline{\kappa}_1^2 u^2}{2k} - \frac{\overline{\kappa}_2 u^2}{2k}\right)du \\
&= \frac{\sqrt{2\pi}}{\sqrt{\kappa_2}}\left(1 + \frac{\kappa_3\overline{\kappa}_1}{2k\kappa_2^2} - \frac{5\kappa_3^2}{24k\kappa_2^3} + \frac{3\kappa_4}{24k\kappa_2^2} - \frac{\overline{\kappa}_1^2}{2k\kappa_2} - \frac{\overline{\kappa}_2}{2k\kappa_2}\right).
\end{aligned} \tag{3.44}$$

Finally, we collect all the results and obtain the claimed expansion. Obviously, we can easily apply the known proof technique to calculate further terms. Thus, it should be clear, that the error term in (3.28) is of order $1/k^2$. □

3.4 Asymptotic expansions of bivariate functions

The aim of this section is the development of a bivariate version of the theorem established in the last section. This result will be applied analysing the bipartite cuckoo graph in Chapter 4.

Theorem 3.2. *Let $f(x,y)$ and $g(x,y)$ be analytic functions in a ball around $(x,y) = (0,0)$, that contains (x_0, y_0) as defined below and let m_1/k and m_2/k be constants greater than zero. Further, let all coefficients $[x^{m_1}y^{m_2}]f(x,y)$ and $[x^{m_1}y^{m_2}]g(x,y)$ be non negative, and such that almost all indices (m_1, m_2) can be represented as a finite linear combination of the set $\{(m_1,m_2)|[x^{m_1}y^{m_2}]f(x,y) > 0\}$ with positive integers as coefficients. Then the asymptotic series expansion*

$$[x^{m_1}y^{m_2}]g(x,y)f(x,y)^k = \frac{g(x_0,y_0)f(x_0,y_0)^k}{2\pi x_0^{m_1}y_0^{m_2}k\sqrt{\Delta}}\left(1 + \frac{H}{24\Delta^3}\frac{1}{k} + \mathcal{O}\left(\frac{1}{k^2}\right)\right), \tag{3.45}$$

is valid, where x_0 and y_0 are uniquely defined by

$$\frac{m_1}{k} = \frac{x_0}{f(x_0,y_0)}\left[\frac{\partial}{\partial x}f(x,y)\right]_{(x_0,y_0)}, \qquad \frac{m_2}{k} = \frac{y_0}{f(x_0,y_0)}\left[\frac{\partial}{\partial y}f(x,y)\right]_{(x_0,y_0)}. \tag{3.46}$$

Generally, let the cummulants κ_{ij} and $\overline{\kappa}_{ij}$ be

$$\kappa_{ij} = \left[\frac{\partial^i}{\partial u^i}\frac{\partial^j}{\partial v^j}\log f(x_0e^u, y_0e^v)\right]_{(0,0)}, \qquad \overline{\kappa}_{ij} = \left[\frac{\partial^i}{\partial u^i}\frac{\partial^j}{\partial v^j}\log g(x_0e^u, y_0e^v)\right]_{(0,0)}. \tag{3.47}$$

Further let $\Delta = \kappa_{20}\kappa_{02} - \kappa_{11}^2$, then H can be expressed by

$$H = \alpha + \beta + \hat{\beta} + \gamma\overline{\kappa}_{10} + \hat{\gamma}\overline{\kappa}_{01} + \delta\overline{\kappa}_{10}\overline{\kappa}_{01} + \eta\overline{\kappa}_{10}^2 + \hat{\eta}\overline{\kappa}_{01}^2 + 4\eta\overline{\kappa}_{20} + 4\hat{\eta}\overline{\kappa}_{02} + 4\delta\overline{\kappa}_{11}, \tag{3.48}$$

where

$$\begin{aligned}
\alpha &= 54\kappa_{21}\kappa_{11}\kappa_{12}\kappa_{20}\kappa_{02} + 6\kappa_{22}\kappa_{20}\kappa_{02}\kappa_{11}^2 - 12\kappa_{22}\kappa_{11}^4 + 4\kappa_{03}\kappa_{11}^3\kappa_{30} \\
&\quad + 36\kappa_{21}\kappa_{11}^3\kappa_{12} + 6\kappa_{22}\kappa_{20}^2\kappa_{02}^2 + 6\kappa_{03}\kappa_{11}\kappa_{30}\kappa_{20}\kappa_{02}, & (3.49)\\
\beta &= -5\kappa_{02}^3\kappa_{30}^2 + 30\kappa_{02}^2\kappa_{30}\kappa_{11}\kappa_{21} - 24\kappa_{02}\kappa_{30}\kappa_{12}\kappa_{11}^2 - 6\kappa_{02}^2\kappa_{30}\kappa_{12}\kappa_{20} \\
&\quad - 12\kappa_{11}\kappa_{02}^2\kappa_{31}\kappa_{20} - 36\kappa_{02}\kappa_{21}^2\kappa_{11}^2 - 9\kappa_{02}^2\kappa_{21}^2\kappa_{20} + 3\kappa_{02}^3\kappa_{40}\kappa_{20} \\
&\quad - 3\kappa_{02}^2\kappa_{40}\kappa_{11}^2 + 12\kappa_{11}^3\kappa_{02}\kappa_{31}, & (3.50)\\
\gamma &= 12\Delta\left(\kappa_{02}^2\kappa_{30} - \kappa_{11}\kappa_{20}\kappa_{03} - 3\kappa_{21}\kappa_{11}\kappa_{02} + \kappa_{12}\kappa_{11}^2 + \kappa_{12}(\kappa_{02}\kappa_{20} + \kappa_{11}^2)\right), & (3.51)\\
\delta &= 24\Delta(\kappa_{11}\kappa_{20}\kappa_{02} - \kappa_{11}^3), & (3.52)\\
\eta &= 12\Delta(\kappa_{02}\kappa_{11}^2 - \kappa_{02}^2\kappa_{20}), & (3.53)
\end{aligned}$$

and $\hat{}$ indicates to replace all functions of type κ_{ij} by κ_{ji}.

This theorem generalises a result of Good [1957], where $g(x,y)$ equals 1.

Proof. The technical conditions on $f(x_0e^{is}, y_0e^{it})$ ensure that the function's maximal modulus is uniquely attained if $s = t = 0$ modulo 2π holds. Further, it can be seen that the saddle point (x_0, y_0) is unique, because the cummulants of second order are strictly positive. See Good [1957] for details.

As previously shown, we start by applying Cauchy's Formula:

$$[x^{m_1}y^{m_2}]g(x,y)f(x,y)^k = -\frac{1}{4\pi^2} \int\limits_{|x|=x_0} \int\limits_{|y|=y_0} \frac{g(x,y)f(x,y)^k}{x^{m_1+1}y^{m_2+1}}\,dy\,dx$$

$$= -\frac{1}{4\pi^2} \int\limits_{|x|=x_0} \int\limits_{|y|=y_0} e^{k\log f(x,y)-(m_1+1)\log x-(m_2+1)\log y-\log g(x,y)}\,dx\,dy. \quad (3.54)$$

Further, we substitute $x = x_0e^{is}$ and $y = y_0e^{it}$, and get

$$[x^{m_1}y^{m_2}]g(x,y)f(x,y)^k$$

$$= \frac{1}{4\pi^2x_0^{m_1}y_0^{m_2}} \int\limits_{-\pi}^{\pi}\int\limits_{-\pi}^{\pi} e^{k\log f\left(x_0e^{is},y_0e^{it}\right)-m_1is-m_2it+\log g\left(x_0e^{is},y_0e^{it}\right)}\,dt\,ds. \quad (3.55)$$

The contribution of the integral taken over the range $I = \big([-\pi,\pi]\times[-\pi,\pi]\big)\setminus\big([-\alpha,\alpha]\times[-\alpha,\alpha]\big)$ is exponentially small compared to the remaining integral. To prove this, we set $\alpha = k^{-1/2+\varepsilon}$, where ε denotes a real number satisfying $0<\varepsilon<1/6$. Furthermore, the Taylor expansion

$$e^{k\log f\left(x_0e^{is},y_0e^{it}\right)-m_1is-m_2it} = f\left(x_0e^{is},y_0e^{it}\right)^k e^{-\frac{k}{2}\left(\kappa_{20}s^2+2\kappa_{11}st+\kappa_{02}t^2\right)+\mathcal{O}\left(k^{-\frac{1}{2}+3\varepsilon}\right)} \quad (3.56)$$

can be obtained. Similar to the univariate case, the modulus of $f\left(x_0e^{is},y_0e^{it}\right)$ outside the range $[-\alpha,\alpha]\times[-\alpha,\alpha]$ is bounded by the maximum modulus attained on the boundary of this square, since x_0, y_0 is the unique maximum of f. It is easy to see, that the function $\exp\left(-\frac{\kappa_{20}}{2}s^2-\kappa_{11}st-\frac{\kappa_{02}}{2}t^2\right)$ attains its maximum along the boundary if $(s,t) = \left(-\frac{\kappa_{11}}{\kappa_{20}}\alpha,\alpha\right)$ resp. $(s,t) = \left(\alpha,-\frac{\kappa_{11}}{\kappa_{02}}\alpha\right)$ holds. Hence, there exists a positive constant c such that the bound $\exp(-ck\alpha^2)$ holds. Consequently, we derive the inequality

$$\left|\iint\limits_I g\left(x_0e^{is},y_0e^{it}\right)e^{k\log f\left(x_0e^{is},y_0e^{it}\right)-m_1is-m_2it}\,dt\,ds\right| \le 4\pi^2g(x_0,y_0)f(x_0,y_0)^ke^{-ck^{2\varepsilon}}, \quad (3.57)$$

which proves the claimed property.

Next, we substitute $u=\sqrt{k}s$ and $v=\sqrt{k}t$ and calculate Taylor expansions of the functions $\log f$ and $\log g$. More precisely, we obtain the expansions

$$k\log f\left(x_0e^{i\frac{u}{\sqrt{k}}},y_0e^{i\frac{v}{\sqrt{k}}}\right) - m_1i\frac{u}{\sqrt{k}} - m_2i\frac{v}{\sqrt{k}} = k\log f(x_0,y_0)$$

$$-\frac{1}{2}\left(\kappa_{20}u^2+2\kappa_{11}uv+\kappa_{02}v^2\right) - \frac{i}{6\sqrt{k}}\left(\kappa_{30}u^3+3\kappa_{21}u^2v+3\kappa_{12}uv^2+\kappa_{03}v^3\right)$$

$$+\frac{1}{24k}\left(\kappa_{40}u^4+4\kappa_{31}u^3v+6\kappa_{22}u^2v^2+4\kappa_{13}uv^3+\kappa_{04}v^4\right)+\mathcal{O}\left(k\alpha^5\right), \quad (3.58)$$

and

$$\log g\big(x_0e^{i\frac{u}{\sqrt{k}}},y_0e^{i\frac{v}{\sqrt{k}}}\big)$$

$$= \log g(x_0, y_0) + \frac{i}{\sqrt{k}}\left(\overline{\kappa}_{10}u + \overline{\kappa}_{01}v\right) - \frac{1}{2k}\left(\overline{\kappa}_{20}u^2 + 2\overline{\kappa}_{11}uv + \overline{\kappa}_{02}v^2\right) + \mathcal{O}\left(\alpha^3\right) \quad (3.59)$$

in the neighbourhood of (x_0, y_0). The linear terms vanish due to the choice of the saddle point. Further, we require Taylor expansions of $\exp(S(u,v))$ for all summands $S(u,v)$ of (3.58) and (3.59) depending on k:

$$\begin{aligned}\exp\left(-i\frac{1}{6\sqrt{k}}\left(\kappa_{30}u^3 + 3\kappa_{21}u^2v + 3\kappa_{12}uv^2 + \kappa_{03}v^3\right)\right)& \\ = 1 - \frac{i}{6\sqrt{k}}&\left(\kappa_{30}u^3 + 3\kappa_{21}u^2v + 3\kappa_{12}uv^2 + \kappa_{03}v^3\right) \\ - \frac{1}{72k}&\left(\kappa_{30}u^3 + 3\kappa_{21}u^2v + 3\kappa_{12}uv^2 + \kappa_{03}v^3\right)^2 + \mathcal{O}\left(k^3\alpha^9\right), \quad (3.60)\end{aligned}$$

$$\begin{aligned}\exp&\left(\frac{1}{24k}\left(\kappa_{40}u^4 + 4\kappa_{31}u^3v + 6\kappa_{22}u^2v^2 + 4\kappa_{13}uv^3 + \kappa_{04}v^4\right)\right) \\ &= 1 + \frac{1}{24k}\left(\kappa_{40}u^4 + 4\kappa_{31}u^3v + 6\kappa_{22}u^2v^2 + 4\kappa_{13}uv^3 + \kappa_{04}v^4\right) + \mathcal{O}\left(k^2\alpha^8\right), \quad (3.61)\end{aligned}$$

$$\exp\left(\frac{i}{\sqrt{k}}\left(\overline{\kappa}_{10}u + \overline{\kappa}_{01}v\right)\right) = 1 + \frac{i}{\sqrt{k}}\left(\overline{\kappa}_{10}u + \overline{\kappa}_{01}v\right) - \frac{1}{2k}\left(\overline{\kappa}_{10}u + \overline{\kappa}_{01}v\right)^2 + \mathcal{O}(\alpha^3), \quad (3.62)$$

$$\exp\left(-\frac{1}{2k}\left(\overline{\kappa}_{20}u^2 + 2\overline{\kappa}_{11}uv + \overline{\kappa}_{02}v^2\right)\right) = 1 - \frac{1}{2k}\left(\overline{\kappa}_{20}u^2 + 2\overline{\kappa}_{11}uv + \overline{\kappa}_{02}v^2\right) + \mathcal{O}(\alpha^4). \quad (3.63)$$

By using these expansions, we can rewrite the remaining integral in the following way:

$$\begin{aligned}&\frac{1}{4\pi^2 x_0^{m_1} y_0^{m_2}} \int_{-\alpha}^{\alpha}\int_{-\alpha}^{\alpha} g\left(x_0e^{is}, y_0e^{it}\right) e^{k\log f\left(x_0e^{is}, y_0e^{it}\right) - m_1is - m_2it} dt\, ds \\ &= \frac{g(x_0,y_0)f(x_0,y_0)^k}{4k\pi^2 x_0^{m_1} y_0^{m_2}} \int_{-\alpha\sqrt{k}}^{\alpha\sqrt{k}}\int_{-\alpha\sqrt{k}}^{\alpha\sqrt{k}} e^{-\frac{1}{2}\left(\kappa_{20}u^2 + 2\kappa_{11}uv + \kappa_{02}v^2\right)} \\ &\quad\times\left(1 - \frac{i}{6\sqrt{k}}\left(\kappa_{30}u^3 + 3\kappa_{21}u^2v + 3\kappa_{12}uv^2 + \kappa_{03}v^3\right)\right. \\ &\qquad + \frac{i}{\sqrt{k}}\left(\overline{\kappa}_{10}u + \overline{\kappa}_{01}v\right) - \frac{1}{72k}\left(\kappa_{30}u^3 + 3\kappa_{21}u^2v + 3\kappa_{12}uv^2 + \kappa_{03}v^3\right)^2 \\ &\qquad + \frac{1}{24k}\left(\kappa_{40}u^4 + 4\kappa_{31}u^3v + 6\kappa_{22}u^2v^2 + 4\kappa_{13}uv^3 + \kappa_{04}v^4\right) \\ &\qquad - \frac{1}{2k}\left(\overline{\kappa}_{10}u + \overline{\kappa}_{01}v\right)^2 - \frac{1}{2k}\left(\overline{\kappa}_{20}u^2 + 2\overline{\kappa}_{11}uv + \overline{\kappa}_{02}v^2\right) \\ &\qquad \left. + \frac{1}{6k}\left(\kappa_{30}u^3 + 3\kappa_{21}u^2v + 3\kappa_{12}uv^2 + \kappa_{03}v^3\right)\left(\overline{\kappa}_{10}u + \overline{\kappa}_{01}v\right) + \mathcal{O}\left(\alpha^9k^3\right)\right) dv\, du. \quad (3.64)\end{aligned}$$

Assume without loss of generality that κ_{20} is greater than or equal κ_{02}. Now, we substitute $u = \sqrt{\kappa_{02}/\Delta}\, a$ and $v = -\kappa_{11}/\sqrt{\kappa_{02}\Delta}\, a + b/\sqrt{\kappa_{02}}$, where Δ denotes the term

$\kappa_{20}\kappa_{02} - \kappa_{11}^2$. Starting from this definition, the equation

$$\int_{-\alpha\sqrt{k}}^{\alpha\sqrt{k}} \int_{-\alpha\sqrt{k}}^{\alpha\sqrt{k}} e^{-\frac{1}{2}\left(\kappa_{20}u^2+2\kappa_{11}uv+\kappa_{02}v^2\right)} dv\, du = \frac{1}{\sqrt{\Delta}} \int_{-\mu}^{\mu} \int_{-\nu(a)}^{\nu(a)} e^{-\frac{a^2+b^2}{2}} db\, da \tag{3.65}$$

holds, where the bounds μ and $\nu(a)$ are given by $\alpha\sqrt{k\Delta/\kappa_{02}}$ and $\alpha\sqrt{k\kappa_{02}} + a\kappa_{11}/\sqrt{\Delta}$. Note that for all a satisfying $-\mu \leq a \leq \mu$, the inequality $\nu(a) \geq \nu_{\min} = \alpha\sqrt{k}(\sqrt{\kappa_{02}} - \kappa_{11}/\sqrt{\kappa_{02}})$ is valid. Further, the relation $\kappa_{20} \geq \kappa_{02}$ implies that $\sqrt{\kappa_{02}} - \kappa_{11}/\sqrt{\kappa_{02}}$ is greater than zero. Thus we can complete the tails of integrals depending on b by using Lemma 3.2 as follows:

$$\int_{\nu(a)}^{\infty} e^{-\frac{b^2}{2}} b^l\, db \leq \int_{\nu_{\min}}^{\infty} e^{-\frac{b^2}{2}} b^l\, db = \mathcal{O}\left(e^{-ck^{2\varepsilon}}\right). \tag{3.66}$$

We continue with an estimation of the part of (3.64) containing the error term.

$$\left| \int_{-\alpha\sqrt{k}}^{\alpha\sqrt{k}} \int_{-\alpha\sqrt{k}}^{\alpha\sqrt{k}} e^{-\frac{1}{2}\left(\kappa_{20}u^2+2\kappa_{11}uv-\kappa_{02}v^2\right)} \mathcal{O}\left(k^3\alpha^9\right) dv\, du \right|$$
$$\leq \frac{\mathcal{O}\left(k^3\alpha^9\right)}{\sqrt{\Delta}} \int_{-\mu}^{\mu} \int_{-\nu(a)}^{\nu(a)} e^{-\frac{a^2+b^2}{2}} db\, da \leq \mathcal{O}\left(k^3\alpha^9\right) \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-\frac{a^2+b^2}{2}} db\, da = \mathcal{O}\left(k^{-\frac{3}{2}+9\varepsilon}\right). \tag{3.67}$$

Furthermore, we introduce the simplified notation

$$I(p,q) = \int_{-\mu}^{\mu} \int_{-\nu(a)}^{\nu(a)} e^{-\frac{a^2+b^2}{2}} a^p b^q\, db\, da. \tag{3.68}$$

Obviously, all terms of the form $a^p b^q$ where at least one exponent is odd lead to an integral of size zero, once the tails are completed. Thus, we omit them in the following calculation. The substitution transforms the integral of (3.64) to

$$\begin{aligned}
I(0,0) &- \kappa_{03}^2 \frac{I(6,0)}{72\kappa_{02}^3} + \left(\kappa_{04} + 4\kappa_{03}\overline{\kappa}_{01}\right)\frac{I(4,0)}{24\kappa_{02}} - \left(\overline{\kappa}_{02} + \overline{\kappa}_{01}^2\right)\frac{I(2,0)}{2\kappa_{02}} \\
+ &\left(\kappa_{11}^6\kappa_{03}^2 + 6\kappa_{11}^4\kappa_{03}\kappa_{02}^2\kappa_{21} - 2\kappa_{11}^3\kappa_{03}\kappa_{02}^3\kappa_{30} - 6\kappa_{11}^5\kappa_{03}\kappa_{02}\kappa_{12} - 18\kappa_{11}^3\kappa_{12}\kappa_{02}^3\kappa_{21}\right. \\
&\left. + 9\kappa_{12}^2\kappa_{11}^4\kappa_{02}^2 + 6\kappa_{11}^2\kappa_{12}\kappa_{02}^4\kappa_{30} + 9\kappa_{21}^2\kappa_{11}^2\kappa_{02}^4 - 6\kappa_{02}^5\kappa_{11}\kappa_{21}\kappa_{30} + \kappa_{02}^6\kappa_{30}^2\right)\frac{I(0,6)}{72\kappa_{02}^3\Delta^3} \\
- &\left(5\kappa_{11}^4\kappa_{03}^2 - 20\kappa_{11}^3\kappa_{03}\kappa_{02}\kappa_{12} - 2\kappa_{11}\kappa_{03}\kappa_{02}^3\kappa_{30} + 12\kappa_{11}^2\kappa_{03}\kappa_{02}^2\kappa_{21}\right. \\
&\left. + 18\kappa_{12}^2\kappa_{11}^2\kappa_{02}^2 + 2\kappa_{12}\kappa_{02}^4\kappa_{30} - 18\kappa_{12}\kappa_{02}^3\kappa_{21}\kappa_{11} + 3\kappa_{21}^2\kappa_{02}^4\right)\frac{I(2,4)}{24\kappa_{02}^3\Delta^2} \\
- &\left(4\kappa_{11}^3\kappa_{02}\kappa_{03}\kappa_{\mathit{10}}g + 12\kappa_{11}^3\kappa_{02}\kappa_{12}\overline{\kappa}_{01} + 4\kappa_{11}^3\kappa_{02}\kappa_{13} - 12\kappa_{11}^2\kappa_{02}^2\kappa_{21}\overline{\kappa}_{01}\right. \\
&- 6\kappa_{11}^2\kappa_{02}^2\kappa_{22} - 12\kappa_{11}^2\kappa_{02}^2\kappa_{12}\overline{\kappa}_{10} - \kappa_{02}^4\kappa_{10} - 4\kappa_{02}^4\kappa_{30}\overline{\kappa}_{10} \\
&\left. + 12\kappa_{11}\kappa_{02}^3\kappa_{21}\overline{\kappa}_{10} + 4\kappa_{11}\kappa_{02}^3\kappa_{30}\overline{\kappa}_{01} + 4\kappa_{11}\kappa_{02}^3\kappa_{31} - \kappa_{11}^4\kappa_{04} - 4\kappa_{11}^4\kappa_{03}\overline{\kappa}_{01}\right)\frac{I(0,4)}{24\kappa_{02}^3\Delta^2}
\end{aligned}$$

$$
\begin{aligned}
&+ \left(3\kappa_{02}^2\kappa_{12}^2 + 2\kappa_{02}^2\kappa_{03}\kappa_{21} - 10\kappa_{11}\kappa_{03}\kappa_{12}\kappa_{02} + 5\kappa_{11}^2\kappa_{03}^2\right)\frac{I(4,2)}{24\Delta}\\
&+ \left(2\kappa_{02}^2\kappa_{21}\overline{\kappa}_{01} + \kappa_{02}^2\kappa_{22} + 2\kappa_{02}^2\kappa_{12}\overline{\kappa}_{10} - 2\kappa_{11}\kappa_{02}\kappa_{03}\overline{\kappa}_{10} - 6\kappa_{11}\kappa_{02}\kappa_{12}\overline{\kappa}_{01}\right.\\
&\qquad \left. - 2\kappa_{11}\kappa_{02}\kappa_{13} + \kappa_{11}^2\kappa_{04} + 4\kappa_{11}^2\kappa_{03}\overline{\kappa}_{01}\right)\frac{I(2,2)}{4\kappa_{02}^2\Delta}\\
&- \left(\kappa_{02}^2\overline{\kappa}_{20} + \kappa_{02}^2\overline{\kappa}_{10}^2 + \kappa_{11}^2\overline{\kappa}_{02} + \kappa_{11}^2\overline{\kappa}_{01}^2 - 2\kappa_{11}\kappa_{02}\overline{\kappa}_{01}\overline{\kappa}_{10} - 2\kappa_{11}\kappa_{02}\overline{\kappa}_{11}\right)\frac{I(0,2)}{2\kappa_{02}\Delta}. \qquad (3.69)
\end{aligned}
$$

Finally, the completion of the tails according to (3.66) and the calculation of the integrals with help of Lemma 3.1 completes the proof. □

Chapter 4

Sparse random graphs

4.1 Introduction

in the late sixties of the last century, the theory of random graphs was developed by Erdős and Rényi, see Erdős and Rényi [1959], Erdős and Rényi [1960], and Erdős and Rényi [1961], Most commonly studied in the literature are the $\mathcal{G}(n,p)$ and the $\mathcal{G}(n,M)$ model. The first consists of all simple graphs[1] possessing n vertices in which each of the $\binom{n}{2}$ edges is chosen independently with probability p. In contrast, the $\mathcal{G}(n,M)$ model selects a member of the set of all simple graphs consisting of n nodes and M edges, such that each member is chosen with the same probability. Despite this different definition, this models are closely related. A lot of analysis has been done on this topic, see, *e.g.*, Bollobás [2001] or Janson et al. [2000] for further information.

In this thesis, we consider generalisations of the $\mathcal{G}(n,M)$ model. More precisely, we admit the occurrence of multiple edges and loops. Furthermore, we consider a similar model of bipartite random graphs. Using this models, we obtain results to analyse the success rate of cuckoo hashing.

4.2 Random graphs and simplified cuckoo hashing

In Chapter 1, we introduced the concept of the cuckoo graph and observed that simplified cuckoo hashing is successful if and only if the corresponding multigraph does not possess a complex component. Thus we are interested in the probability, that a graph consisting of $2m$ labelled nodes and n labelled edges consists of tree and unicyclic components only. To insert an edge, we generate an ordered pair of independent uniformly at random selected nodes $\langle x,y\rangle$ and add the edge directed from x to y. This model is similar to the multigraph process in Janson et al. [1993], except that we introduced edge labels and directions instead of compensation factors. With help of a generating function approach, we obtain the following result.

Theorem 4.1. *Suppose that $\varepsilon \in (0,1)$ is fixed. Then the probability that a simplified cuckoo hash of $n = \lfloor (1-\varepsilon)m \rfloor$ data points into a table of size $2m$ succeeds, (that is, the*

[1]A graph is called simple, if it does not possess multiple edges nor self-loops. Non-simple graphs are also called multigraphs, *cf.* Diestel [2005]

corresponding cuckoo graph contains no complex component,) is equal to

$$1 - \frac{(5-2\varepsilon)(1-\varepsilon)^2}{48\varepsilon^3}\frac{1}{m} + \mathcal{O}\left(\frac{1}{m^2}\right). \tag{4.1}$$

This result is given in Drmota and Kutzelnigg [2008] and Kutzelnigg [2008], together with a sketch of the proof.

Proof of Theorem 4.1

First, we count the number of all multigraphs, regardless if they contain complex components or not. Let $G_{2m,n}$ denote the set of all node and edge labelled directed multigraphs (V, E) with $|V| = 2m$ and $|E| = n$. Obviously, there are $\binom{2m}{2}$ ways to chose an undirected edge connecting two different nodes and two possible directions for each of this edges. Additionally there exist $2m$ different possibilities to add slings. Each of the edges might be chosen multiple times. Thus, the equation

$$\#G_{2m,n} = \left[\frac{v^n}{n!}\right] (e^v)^{2m} \left(e^{2v}\right)^{\binom{2m}{2}} = (4m^2)^n \tag{4.2}$$

holds.

Further, let $G^\circ_{2m,n}$ be the graph class without complex components and let

$$g^\circ(x, v) = \sum_{m,n} \#G^\circ_{2m,n} \frac{x^{2m}}{(2m)!}\frac{v^n}{n!}. \tag{4.3}$$

Equation (4.3) denotes the corresponding double exponential bivariate generating function. Our next goal is to describe this function. For this purpose, we start considering tree components. Since the number of edges of a tree equals the number of nodes minus one, we can concentrate on counting nodes. Furthermore, we don't have to worry about edge orientations because there exist exactly 2^{k-1} possible ways to introduce directions in an undirected tree possessing k nodes. Thus, we are interested in the generating function $\tilde{t}(x)$ of unrooted (vertex) labelled trees, because they model a tree component of the cuckoo graph. This generating function is strongly related to the generating function of rooted trees $t(x)$ that satisfies the functional equation (*cf.* Flajolet and Sedgewick [2008])

$$t(x) = xe^{t(x)}. \tag{4.4}$$

We obtain each rooted tree whose root is labelled by one if we take all unrooted trees and declare the node with label one to be root. Further, all other rooted trees can be constructed from a pair T_1, T_2 of rooted trees as follows. One of this trees, say T_2, contains the node with label one and we simply add an edge connecting the roots of these trees. Now we declare the root of T_1 to be the root of the whole tree. Since the last step produces each tree with a root not labelled by one exactly twice, we obtain the equation (*cf.* Flajolet et al. [1989])

$$\tilde{t}(x) = t(x) - \frac{1}{2}t(x)^2, \tag{4.5}$$

counting all (vertex) labelled unrooted trees.

Next, we consider components containing exactly one cycle. Hereby, we do not care for the orientation of edges, that means we do not require a directed cycle. Such a component consists of cyclic connected nodes, that can be considered to generate roots of rooted trees. Thus we obtain the generating function

$$\frac{t(x)^k}{2k}, \tag{4.6}$$

for a component possessing a cycle of length k. If k equals one, the cycle is just a sling and hence there exists one edge whose orientation is unchangeable which is compensated by a factor 1/2. Further, the case k equals two corresponds to a double edge. In the latter case, the interchange of the labels assigned to the repeated edge does not change the graph, what is compensated with a factor 1/2. An additional factor 1/2 arises, since every graph is produced twice. Similarly, the product is divided by $2k$ to account for cyclic order and change of orientation if k is greater than two. Consequently the generating function of an unicyclic component is given by

$$\sum_{k\geq 1} \frac{t(x)^k}{2k} = \frac{1}{2}\log\frac{1}{1-t(x)}, \tag{4.7}$$

see also Janson et al. [1993].

Using this intermediate results, we obtain the following result.

Lemma 4.1. *The generating function $g^\circ(x,v)$ is given by*

$$g^\circ(x,v) = \frac{e^{\frac{1}{2v}\tilde{t}(2xv)}}{\sqrt{1-t(2xv)}}. \tag{4.8}$$

Proof. We introduce the variable v that counts the number of edges and introduce edge orientations. Starting from (4.5) resp. (4.7), we obtain the bivariate generating functions $\frac{1}{2v}\tilde{t}(2xv)$ and $\frac{1}{2}\log\frac{1}{1-t(2xv)}$. Finally the function $g^\circ(x,v)$ is obtained by a set composition. □

It is easy to see, that the insertion of a new edge reduces the number of tree components by one, conditioned that no complex component exists after the insertion. Thus, a graph without complex component that possesses $2m$ nodes and n edges, contains exactly $2m-n$ tree components. Using this fact and Cauchy's Formula, we obtain the integral representation

$$\begin{aligned}\#G^\circ_{2m,n} &= \left[\frac{x^{2m}v^n}{(2m)!n!}\right] g^\circ(x,v) = \left[\frac{x^{2m}}{(2m)!}\right] \frac{n!\,\tilde{t}(2x)^{2m-n}}{(2m-n)!\,2^{2m-n}\sqrt{1-t(2x)}} \\ &= [x^{2m}]\frac{2^n n!(2m)!\,\tilde{t}(x)^{2m-n}}{(2m-n)!\sqrt{1-t(x)}} = \frac{2^n n!(2m)!}{(2m-n)!}\oint \frac{\tilde{t}(x)^{2m-n}}{\sqrt{1-t(x)}}\frac{dx}{x^{m+1}}. \end{aligned} \tag{4.9}$$

This integral can be asymptotically evaluated with help of the saddle point method. To do so, we consider an infinite sequence of hash tables. We define the ratio

$$\varepsilon' = 1 - \frac{n}{m} = 1 - \frac{\lfloor(1-\varepsilon)m\rfloor}{m}. \tag{4.10}$$

Starting with a table size $2m$ and $n = (1 - \varepsilon')m$ keys, we increase both parameters such that the quotient of table size and number of keys remains constant. For instance, we may consider the sequence $(m, (1-\varepsilon')m), (2m, 2(1-\varepsilon')m), (3m, 3(1-\varepsilon')m), \ldots$ of pairs of parameters and the corresponding hash tables.

Obviously, all technical conditions of Theorem 3.1 are fulfilled. Thus we can apply it by setting

$$f(x) \to \tilde{t}(x), \quad g(x) \to \frac{1}{\sqrt{1-t(x)}}, \quad k \to 2m - n = m(1+\varepsilon'), \quad \text{and} \quad m \to 2m.$$

The calculation of the saddle point requires knowledge of the first derivative of $\tilde{t}(x)$, that itself involves $t'(x)$. This derivatives can be obtained by implicit differentiation. Hence,

$$t'(x) = e^{t(x)} + xe^{t(x)}t'(x) = \frac{e^{t(x)}}{1-t(x)}, \tag{4.11}$$

and

$$\tilde{t}'(x) = t'(x) - t(x)t'(x) = e^{t(x)} = \frac{t(x)}{x}. \tag{4.12}$$

Further, the saddle point x_0 is obtained using

$$\frac{2m}{2m-n} = x_0 \frac{\tilde{t}'(x_0)}{\tilde{t}(x_0)} = \frac{t(x_0)}{t(x_0) - \frac{1}{2}t(x_0)^2}. \tag{4.13}$$

Thus we get the solution

$$t(x_0) = 1 - \varepsilon' \quad \text{resp.} \quad x_0 = (1-\varepsilon')e^{-(1-\varepsilon')}. \tag{4.14}$$

Additionally, we conclude that

$$\tilde{t}(x_0) = t(x_0) - \frac{1}{2}t(x_0)^2 = \frac{1}{2}(1 - \varepsilon'^2) \tag{4.15}$$

holds. Similarly, we calculate the cummulants κ_2, κ_3, $\overline{\kappa}_1$, and $\overline{\kappa}_2$. Consequently, we obtain the equation

$$\begin{aligned}\kappa_2 &= \left[\frac{d^2}{d\phi^2}\log\tilde{t}\left(x_0e^{\phi}\right)\right]_{\phi=0} = \left[\frac{d}{d\phi}\frac{t\left(x_0e^{\phi}\right)}{\tilde{t}\left(x_0e^{\phi}\right)}\right]_{\phi=0} = \frac{t'(x_0)\tilde{t}(x_0) - t(x_0)\tilde{t}'(x_0)}{\tilde{t}(x_0)^2}x_0 \\ &= \frac{t(x_0)\tilde{t}(x_0) - t(x_0)^2(1-t(x_0))}{\tilde{t}(x_0)^2(1-t(x_0))} = \frac{t(x_0)}{2(1-t(x_0))\left(1-\frac{1}{2}t(x_0)\right)^2} = \frac{2(1-\varepsilon')}{\varepsilon'(1+\varepsilon')^2},\end{aligned} \tag{4.16}$$

and likewise the results

$$\kappa_3 = 2\frac{2\varepsilon'^3 - 5\varepsilon'^2 + 2\varepsilon' + 1}{(\varepsilon'+1)^3\varepsilon'^3}, \qquad \overline{\kappa}_1 = \frac{1-\varepsilon'}{2\varepsilon'^2}, \qquad \text{and} \qquad \overline{\kappa}_2 = \frac{(1-\varepsilon')(2-\varepsilon')}{2\varepsilon'^4}. \tag{4.17}$$

Using these intermediate results, we obtain that the leading term of the asymptotic expansion of the integral in (4.9) equals

$$\frac{\tilde{t}(x_0)^{m(1+\varepsilon')}}{\sqrt{1-t(x_0)}x_0^{2m}\sqrt{2\pi m(1+\varepsilon')\kappa_2}} = \frac{(1+\varepsilon')^{m(1+\varepsilon')}e^{2m(1-\varepsilon')}}{(1-\varepsilon')^{m(1-\varepsilon')}2^{m(1+\varepsilon')}}\frac{\sqrt{1+\varepsilon'}}{\sqrt{4\pi m(1-\varepsilon')}}, \tag{4.18}$$

and the next term of the expansion is given by

$$\frac{12\kappa_2\kappa_3\overline{\kappa}_1 + 3\kappa_2\kappa_4 - 12\kappa_2^2\overline{\kappa}_1^2 - 12\kappa_2^2\overline{\kappa}_2 - 5\kappa_3^2}{24\kappa_2^3 m(1+\varepsilon')} = \frac{5 - 12\varepsilon' + 4\varepsilon'^2 + 12\varepsilon^3 - \varepsilon'^4}{48\varepsilon'^3(\varepsilon'-1)m}. \tag{4.19}$$

Further, we calculate an asymptotic expansion of the factor preceding the integral applying Stirling's formula (3.23),

$$\begin{aligned}&\frac{2^n n!(2m)!}{(2m-n)!}\\ &\quad= \frac{2^n n^n (2m)^{2m}}{(2m-n)^{2m-n}e^{2n}}\frac{\sqrt{4\pi n m}}{\sqrt{2m-n}}\left(1+\frac{1}{12n}+\frac{1}{24m}-\frac{1}{12(2m-n)}+\mathcal{O}\left(\frac{1}{m^2}\right)\right)\\ &\quad= \frac{2^{m(3-\varepsilon')}m^{2m(1-\varepsilon')}(1-\varepsilon')^{m(1-\varepsilon')}}{(1+\varepsilon')^{m(1+\varepsilon')}e^{2m(1-\varepsilon')}}\frac{\sqrt{4\pi m(1-\varepsilon')}}{\sqrt{1+\varepsilon'}}\left(1+\frac{1+4\varepsilon'-\varepsilon'^2}{24m(1-\varepsilon'^2)}+\mathcal{O}\left(\frac{1}{m^2}\right)\right).\end{aligned} \tag{4.20}$$

Combining these results, we get the desired asymptotic expansion,

$$\#G^{\circ}_{2m,m(1-\varepsilon')} = (2m)^{2m(1-\varepsilon')}\left(1-\frac{(5-2\varepsilon')(1-\varepsilon')^2}{48\varepsilon'^3}\frac{1}{m}+\mathcal{O}\left(\frac{1}{m^2}\right)\right). \tag{4.21}$$

Thus, the probability that a randomly selected graph does not contain a complex component equals

$$\frac{\#G^{\circ}_{2m,m(1-\varepsilon')}}{\#G_{2m,m(1-\varepsilon')}} = 1-\frac{(5-2\varepsilon')(1-\varepsilon')^2}{48\varepsilon'^3}\frac{1}{m}+\mathcal{O}\left(\frac{1}{m^2}\right). \tag{4.22}$$

Since

$$\varepsilon' = 1-\frac{\lfloor m(1-\varepsilon)\rfloor}{m} = 1-\frac{m(1-\varepsilon)-\{m(1-\varepsilon)\}}{m} = \varepsilon+\mathcal{O}\left(\frac{1}{m}\right) \tag{4.23}$$

holds, we can replace ε' by ε in (4.22). Consequently, the same expansion holds if we consider the series $(m, \lfloor m(1+\varepsilon)\rfloor), (m+1, \lfloor (m+1)(1+\varepsilon)\rfloor), (m+2, \lfloor (m+2)(1+\varepsilon)\rfloor), \ldots$ and the corresponding hash tables, what completes the proof of the theorem.

4.3 Bipartite random graphs and standard cuckoo hashing

Our next goal is the adoption of this ideas to bipartite cuckoo graphs, as defined in Chapter 1. Again, we are interested in the probability, that such a graph contains no complex component. More precisely, we consider bipartite multigraphs consisting of m labelled nodes of each type and n labelled edges. Each of the labelled edges represents a key and connects two independent uniform selected nodes of different type. The analysis is once again based on generating functions. However, we have to replace the univariate functions by (double exponential) bivariate generating functions to model the different types of nodes, what makes the analysis more complicated. Nonetheless, we obtain the following result.

Theorem 4.2. *Suppose that $\varepsilon \in (0,1)$ is fixed. Then the probability $p(n,m)$ that a cuckoo hash of $n = \lfloor (1-\varepsilon)m \rfloor$ data points into two tables of size m succeeds, (that is, the corresponding cuckoo graph contains no complex component,) is equal to*

$$p(n,m) = 1 - \frac{(2\varepsilon^2 - 5\varepsilon + 5)(1-\varepsilon)^3}{12(2-\varepsilon)^2\varepsilon^3}\frac{1}{m} + \mathcal{O}\left(\frac{1}{m^2}\right). \tag{4.24}$$

A sketch of the proof can be found in Kutzelnigg [2006]. A detailed version of the proof is given in Drmota and Kutzelnigg [2008].

Proof of Theorem 4.2

Once more, we start counting all bipartite graphs without considering the type of their components. Let $G_{m_1,m_2,n}$ denote the set of all node and edge labelled bipartite multigraphs (V_1, V_2, E) with $|V_1| = m_1$, $|V_2| = m_2$, and $|E| = n$. By definition, it is clear that the number of all graphs of the family $G_{m_1,m_2,n}$ equals

$$\#G_{m_1,m_2,n} = m_1^n m_2^n. \tag{4.25}$$

In particular, we are interested in the case $m_1 = m_2 = m$ and $n = \lfloor (1-\varepsilon)m \rfloor$, where $\varepsilon \in (0,1)$. This means that the graph is relatively sparse.

Next, let $G^\circ_{m_1,m_2,n}$ denote those graphs in $G_{m_1,m_2,n}$ without complex components, that is, all components are either trees or unicyclic. Further,

$$g^\circ(x,y,v) = \sum_{m_1,m_2,n} \#G^\circ_{m_1,m_2,n} \frac{x^{m_1}}{m_1!}\frac{y^{m_2}}{m_2!}, \frac{v^n}{n!} \tag{4.26}$$

denotes the corresponding generating function. First, we want to describe this generating function. For this purpose we will now consider bipartite trees.

We call a tree bipartite if the vertices are partitioned into two classes V_1 ("black" nodes) and V_2 ("white" nodes) such that no node has a neighbour of the same class. They are called labelled if the nodes of first type, that is nodes in V_1, are labelled by $1, 2, \ldots, |V_1|$ and the nodes of second type are independently labelled by $1, 2, \ldots, |V_2|$, see also Gimenez et al. [2005].

Let T_1 denote the set of bipartite rooted trees, where the root is contained in V_1, similarly T_2 the set of bipartite rooted trees, where the root is contained in V_2, and $\tilde{T}$ the class of unrooted bipartite trees. Furthermore, let t_{1,m_1,m_2} resp. t_{2,m_1,m_2} denote the number of trees in T_1 resp. T_2 with m_1 nodes of type of type 1 and m_2 of type 2. Similarly we define $\tilde{t}_{m_1,m_2}$. The corresponding generating functions are defined by

$$t_1(x,y) = \sum_{m_1,m_2 \geq 0} t_{1,m_1,m_2} \frac{x^{m_1}}{m_1!}\frac{y^{m_2}}{m_2!}, \tag{4.27}$$

$$t_2(x,y) = \sum_{m_1,m_2 \geq 0} t_{2,m_1,m_2} \frac{x^{m_1}}{m_1!}\frac{y^{m_2}}{m_2!}, \tag{4.28}$$

and by

$$\tilde{t}(x,y) = \sum_{m_1,m_2 \geq 0} \tilde{t}_{m_1,m_2} \frac{x^{m_1}}{m_1!}\frac{y^{m_2}}{m_2!}. \tag{4.29}$$

The assertion of the following lemma is a first attempt exploring these trees.

Lemma 4.2. *The generating functions $t_1(x,y)$, $t_2(x,y)$, and $\tilde{t}(x,y)$ are given by*

$$t_1(x,y) = xe^{t_2(x,y)}, \quad t_2(x,y) = ye^{t_1(x,y)}, \tag{4.30}$$

and by

$$\tilde{t}(x,y) = t_1(x,y) + t_2(x,y) - t_1(x,y)t_2(x,y). \tag{4.31}$$

Furthermore we have

$$t_{1,m_1,m_2} = m_1^{m_2} m_2^{m_1-1}, \quad t_{2,m_1,m_2} = m_1^{m_2-1} m_2^{m_1}, \tag{4.32}$$

and

$$\tilde{t}_{m_1,m_2} = m_1^{m_2-1} m_2^{m_1-1}. \tag{4.33}$$

The explicit formula for $\tilde{t}_{m_1,m_2}$ is originally due to Scoins [1962].

Proof. The functional equations (4.30) are obviously given by their recursive description. Note that $t_1(x,y) = t_2(y,x)$ holds and that $t_1(x,x)$ equals the usual tree function $t(x)$ defined in (4.4). Thus, $t_1(x,y)$ and $t_2(x,y)$ are surely analytic functions for $|x| < e^{-1}$ and $|y| < e^{-1}$. This holds due to the fact that the radius of convergence of $t(x)$ equals $1/e$.

In the last section, we mentioned that the generating function of usual unrooted labelled trees is given by $t(x) - t(x)^2/2$. Thus, (4.31) is a generalisation of this result, and can be proved in a similar way. First, consider a rooted tree, possessing a black root labelled by 1, as an unrooted tree. Next, examine an unordered pair (t_1, t_2) of trees from $T_1 \times T_2$, and join the roots by an edge. If the black node labelled by 1 is contained in t_1, consider the root of t_2 as new root, and we obtain a tree possessing a white root and at least one black node. Else, consider the root of t_1 as new root, and we obtain a tree with a black root node not labelled by 1.

Lagrange inversion applied to the equation $t_1(x,y) = x\exp\left(ye^{t_1(x,y)}\right)$ yields

$$\begin{aligned}
[x^{m_1}y^{m_2}]t_1(x,y) &= [y^{m_2}]\frac{1}{m_1}[u^{m_1-1}]\left(e^{ye^u}\right)^{m_1} \\
&= [y^{m_2}]\frac{1}{m_1}[u^{m_1-1}]\sum_{k\ge 0}\sum_{l\ge 0}\frac{m_1^k y^k}{k!}\frac{u^l k^l}{l!} \\
&= [y^{m_2}]\frac{1}{m_1}\sum_{k\ge 0}\frac{m_1^k y^k}{k!}\frac{k^{m_1-1}}{(m_1-1)!} = \frac{m_1^{m_2} m_2^{m_1-1}}{m_1! m_2!}.
\end{aligned} \tag{4.34}$$

Furthermore, $\tilde{t}_{m_1,m_2} = t_{1,m_1,m_2}/m_1 = t_{2,m_1,m_2}/m_2$ holds since there are exactly m_1 ways to choose the root of type 1 in an unrooted tree with m_1 nodes of type 1. □

Later on, we will make use of the partial derivatives of this functions.

Lemma 4.3. *The partial derivatives of the functions $\tilde{t}(x,y)$, $t_1(x,y)$ and $t_1(x,y)$ are given by*

$$\frac{\partial}{\partial x}\tilde{t}(x,y) = \frac{t_1(x,y)}{x}, \qquad \frac{\partial}{\partial y}\tilde{t}(x,y) = \frac{t_2(x,y)}{y}, \tag{4.35}$$

$$\frac{\partial}{\partial x}t_1(x,y) = \frac{t_1(x,y)}{x(1-t_1(x,y)t_2(x,y))}, \qquad \frac{\partial}{\partial y}t_1(x,y) = \frac{t_1(x,y)\,t_2(x,y)}{y(1-t_1(x,y)t_2(x,y))}, \tag{4.36}$$

and

$$\frac{\partial}{\partial x}t_2(x,y) = \frac{t_1(x,y)\,t_2(x,y)}{x(1-t_1(x,y)t_2(x,y))}, \qquad \frac{\partial}{\partial y}t_2(x,y) = \frac{t_1(x,y)}{y(1-t_1(x,y)t_2(x,y))}. \tag{4.37}$$

Proof. All this results can be easily calculated using implicit differentiation. For instance, we obtain $\frac{\partial}{\partial x}t_1(x,y)$ with the equation system

$$\frac{\partial}{\partial x}t_1(x,y) = \frac{\partial}{\partial x}\left(xe^{t_2(x,y)}\right) = e^{t_2(x,y)} + xe^{t_2(x,y)}\frac{\partial}{\partial x}t_2(x,y), \tag{4.38}$$

$$\frac{\partial}{\partial x}t_2(x,y) = \frac{\partial}{\partial x}\left(ye^{t_1(x,y)}\right) = ye^{t_1(x,y)}\frac{\partial}{\partial x}t_1(x,y). \tag{4.39}$$

□

Next, we draw our attention on unicyclic components.

Lemma 4.4. *The generating function of a connected graph with exactly one cycle is given by*

$$c(x,y) = \sum_{k\geq 1}\frac{1}{2k}t_1(x,y)^k t_2(x,y)^k = \frac{1}{2}\log\frac{1}{1-t_1(x,y)t_2(x,y)}. \tag{4.40}$$

Proof. Of course, a cycle has to have an even number of nodes, say $2k$, where k nodes are black and the other k nodes are white. A cyclic node of black colour can be considered as the root of a rooted tree of the set T_1 and similarly, a white cyclic node can be considered as the root of a rooted tree of the set T_2. Note that we have to divide the product of the generating functions $t_1(x,y)^k t_2(x,y)^k$ by $2k$ to account for cyclic order and change of orientation. Hence, the corresponding generating functions of a unicyclic graph with $2k$ cyclic points is given by

$$\frac{1}{2k}t_1(x,y)^k t_2(x,y)^k. \tag{4.41}$$

Consequently, the claimed equation holds. □

Using these functions, we can describe the generating function $g^\circ(x,y,v)$.

Lemma 4.5. *The generating function $g^\circ(x,y,v)$ is given by*

$$g^\circ(x,y,v) = \frac{e^{\frac{1}{v}\tilde{t}(xv,yv)}}{\sqrt{1-t_1(xv,yv)t_2(xv,yv)}}. \tag{4.42}$$

Proof. We have to count graphs where each component is either an unrooted tree (that is counted by $\tilde{t}(x,y)$) or a graph with exactly one cycle. Since a cyclic component of size m_1+m_2 possesses exactly the same number of edges as nodes and since there are $(m_1+m_2)!$ possible edge labels, the corresponding generating function that takes the edges into account in given by $c(xv,yv)$. Similarly, a tree of size m_1+m_2 has exactly $n = m_1+m_2-1$ edges. Consequently the generating function $\tilde{t}(xv,yv)/v$ corresponds to a bipartite unrooted tree. Hence, the generating function $g^\circ(x,y,v)$ is given by

$$g^\circ(x,y,v) = e^{\frac{1}{v}\tilde{t}(xv,yv)+c(xv,yv)} = \frac{e^{\frac{1}{v}\tilde{t}(xv,yv)}}{\sqrt{1-t_1(xv,yv)t_2(xv,yv)}}, \tag{4.43}$$

which completes the proof of the lemma. □

Corollary 4.1. *The number of graphs $\#G^\circ_{m_1,m_2,n}$ is given by*

$$\#G^\circ_{m_1,m_2,n} = \frac{m_1!\,m_2!\,n!}{(m_1+m_2-n)!}[x^{m_1}y^{m_2}]\frac{\tilde{t}(x,y)^{m_1+m_2-n}}{\sqrt{1-t_1(x,y)t_2(x,y)}}. \tag{4.44}$$

We use the corollary and Cauchy's Formula and obtain

$$\#G^\circ_{m,m,n} = \frac{-(m!)^2\, n!}{4\pi^2(2m-n)!} \oint\oint \frac{\tilde{t}(x,y)^{2m-n}}{\sqrt{1-t_1(x,y)t_2(x,y)}} \frac{dx}{x^{m+1}} \frac{dy}{y^{m+1}}. \tag{4.45}$$

This is in fact an integral that can be asymptotically evaluated using a (double) saddle point method, see Theorem 3.2. Additionally, we obtain by Stirling's formula (3.23) the asymptotic expansion

$$\begin{aligned}\frac{(m!)^2\, n!}{(2m-n)!} &= \frac{2\pi m^{2m+1} n^n}{e^{2n}(2m-n)^{2m-n}} \sqrt{\frac{n}{2m-n}} \\ &\times \left(1 + \frac{1+\varepsilon'-\varepsilon'^2}{6(1-\varepsilon')^2 m} + \frac{1+2\varepsilon'-\varepsilon'^2-2\varepsilon'^3+\varepsilon'^4}{72(1-\varepsilon'^2)^2 m^2} + \mathcal{O}\left(\frac{1}{m^3}\right)\right), \end{aligned}\tag{4.46}$$

where $\varepsilon' = 1 - n/m$.

For our problem, it turns out that if

$$\varepsilon' = 1 - \frac{n}{m} = 1 - \frac{\lfloor (1-\varepsilon)m \rfloor}{m}, \tag{4.47}$$

is fixed in $(0,1)$, the saddle point is given by

$$x_0 = y_0 = \frac{n}{m} e^{-\frac{n}{m}} = (1-\varepsilon')e^{\varepsilon'-1} < \frac{1}{e}. \tag{4.48}$$

This can be easily checked. By symmetry it is clear that $x_0 = y_0$. Further, $t_1(x,x) = t(x) = xe^{t(x)}$ equals the tree function. Hence we get

$$t_1(x_0,x_0) = 1-\varepsilon' = \frac{n}{m}, \qquad \tilde{t}(x_0,x_0) = 1-\varepsilon'^2 = \frac{n}{m}\left(2 - \frac{n}{m}\right). \tag{4.49}$$

For instance, we further obtain

$$\kappa_{20} = \frac{t_1(x_0,y_0)}{(1-t_1(x_0,y_0)t_2(x_0,y_0))\tilde{t}(x_0,y_0)} - \frac{t_1(x_0,y_0)^2}{\tilde{t}(x_0,y_0)^2} = \frac{\varepsilon'^2-\varepsilon'+1}{(\varepsilon'+1)^2\,\varepsilon'\,(2-\varepsilon')}, \tag{4.50}$$

$$\kappa_{11} = \frac{t_1(x_0,y_0)t_2(x_0,y_0)}{(1-t_1(x_0,y_0)t_2(x_0,y_0))\tilde{t}(x_0,y_0)} - \frac{t_1(x_0,y_0)t_2(x_0,y_0)}{\tilde{t}(x_0,y_0)^2} = \frac{1-2\varepsilon'}{(\varepsilon'+1)^2\,\varepsilon'\,(2-\varepsilon')}, \tag{4.51}$$

and

$$\Delta = \frac{1-\varepsilon'}{\varepsilon'(2-\varepsilon')(\varepsilon'+1)^3} = \frac{m^4 n}{(2m-n)^3(m^2-n^2)}. \tag{4.52}$$

Further cummulants can be calculated in the same way, but have been computed with help of a computer algebra system in a half-automatic way. The maple source file is included in the attached CD-Rom, see also Appendix B.

We set

$$f \to \tilde{t}, \quad g \to \frac{1}{\sqrt{1-t_1t_2}}, \quad k \to 2m-n, \quad m_1 \to m, \quad \text{and} \quad m_2 \to m,$$

in Theorem 3.2 and apply the saddle point method. Thus, we obtain an asymptotic expansion of the double integral of (4.45) possessing the leading coefficient

$$\frac{\tilde{t}(x_0,y_0)^{2m-n}}{2\pi(2m-n)x_0^m y_0^m\sqrt{1-t_1(x_0,y_0)t_2(x_0,y_0)}\sqrt{\Delta}} = \frac{e^{2n}m^{2n+1}(2m-2)^{2m-n-1}}{2\pi n^n m^{2m}\sqrt{\Delta}\sqrt{m^2-n^2}}. \tag{4.53}$$

Furthermore, the coefficient of $1/m$ of this asymptotic expansion is given by

$$C = \frac{\varepsilon'^6 - 10\varepsilon'^5 + 21\varepsilon'^4 - 2\varepsilon'^3 - 27\varepsilon'^2 + 20\varepsilon' - 5}{12\varepsilon'^3(-2+\varepsilon')^2(1-\varepsilon')}. \tag{4.54}$$

With help of this results and (4.46) we finally obtain the asymptotic expansion

$$\begin{aligned}\#G^{\circ}_{m,m,(1-\varepsilon')m} &= m^{2(1-\varepsilon')m}\left(1+\frac{C}{(1+\varepsilon')m}+\frac{1+\varepsilon'-\varepsilon'^2}{6(1-\varepsilon')^2m}+\mathcal{O}\left(\frac{1}{m^2}\right)\right),\\ &= m^{2(1-\varepsilon')m}\left(1-\frac{1}{m}\frac{(2\varepsilon'^2-5\varepsilon'+5)(1-\varepsilon')^3}{12(2-\varepsilon')^2\varepsilon'^3}+\mathcal{O}\left(\frac{1}{m^2}\right)\right),\end{aligned} \tag{4.55}$$

of the number of graphs without complex components.

We may now replace ε' by $\varepsilon = \varepsilon' + \mathcal{O}(1/m)$. Let $p(n,m)$ denote the probability, that every component of the cuckoo graph is either a tree or unicyclic, after the insertion of n edges. So, we finally obtain

$$p(n,m) = \frac{\#G^{\circ}_{m,m,\lfloor(1-\varepsilon)m\rfloor}}{\#G_{m,m,\lfloor(1-\varepsilon)m\rfloor}} = 1-\frac{1}{m}\frac{(2\varepsilon^2-5\varepsilon+5)(1-\varepsilon)^3}{12(2-\varepsilon)^2\varepsilon^3}+\mathcal{O}\left(\frac{1}{m^2}\right). \tag{4.56}$$

This step completes the proof of Theorem 4.2. Figure 4.1 depicts the graph of $h(\varepsilon) = (2\varepsilon^2-5\varepsilon+5)(1-\varepsilon)^3/(12(2-\varepsilon)^2\varepsilon^3)$. The series expansion of the function $h(\varepsilon)$ with respect to $\varepsilon \to 0$ is given by

$$h(\varepsilon) = \frac{5}{48}\varepsilon^{-3} - \frac{5}{16}\varepsilon^{-2} + \frac{21}{64}\varepsilon^{-1} - \frac{13}{96} + \frac{3}{256}\varepsilon + \frac{1}{256}\varepsilon^2 + \frac{1}{1024}\varepsilon^3 + \mathcal{O}\left(\varepsilon^4\right). \tag{4.57}$$

We want to note that it is also possible to obtain a slightly more precise asymptotic expansion for

$$p(n,m) = 1 - \frac{h(\varepsilon)}{m} - \frac{\hat{h}(\varepsilon)}{m^2} + \mathcal{O}\left(\frac{1}{m^3}\right), \tag{4.58}$$

where $\tilde{h}(\varepsilon)$ is again explicit. This can be done by refining the calculations related to Lemma 3.2.

For example, we can apply these expansions in order to obtain asymptotic representations for the probability $q(n+1,m)$ that the insertion of the $n+1$-st edge creates a bicyclic component, conditioned on the property, that the first n insertions did not create such a component.

Lemma 4.6. *The probability that the insertion of the $n+1$-st inserted key forces a rehash is given by*

$$q(n+1,m) = -\frac{h'(\varepsilon)}{m^2} + \mathcal{O}\left(\frac{1}{m^3}\right). \tag{4.59}$$

This is uniform for $n/m \le 1-\eta$, assuming $\eta > 0$.

Proof. By definition we have $p(n+1,m) = (1-q(n+1,m))p(n,m)$. Hence we get

$$\begin{aligned} q(n+1,m) &= \frac{p(n,m)-p(n+1,m)}{p(n,m)} \\ &= \left(\frac{h(\varepsilon)}{m} - \frac{h\left(\varepsilon-\frac{1}{m}\right)}{m} + \frac{\hat{h}(\varepsilon)}{m^2} - \frac{\hat{h}\left(\varepsilon-\frac{1}{m}\right)}{m^2} + \mathcal{O}\left(\frac{1}{m^3}\right)\right)\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right) \\ &= -\frac{h'(\varepsilon)}{m^2} - \frac{\hat{h}'(\varepsilon)}{m^3} + \mathcal{O}\left(\frac{1}{m^3}\right) \\ &= \frac{(1-\varepsilon)^2(-\varepsilon^3+8\varepsilon^2-15\varepsilon+10)}{4(2-\varepsilon)^3\varepsilon^4}\frac{1}{m^2} + \mathcal{O}\left(\frac{1}{m^3}\right). \end{aligned} \tag{4.60}$$

□

4.4 Asymmetric cuckoo hashing

As mentioned in Chapter 1, asymmetric cuckoo hashing uses tables of different size. To be precise, we choose the tables in such a way, that the first table holds more memory cells than the second one. Thus, we expect that the number of keys actually stored in the first table increases, what leads to improved search and insertion performance. In this section, we adopt the previous analysis such that it covers the asymmetric variant too. During the analysis, we make use of the factor of asymmetry c. This constant determines the size of both hash tables, which hold $m_1 = \lfloor m(1+c)\rfloor$ respectively $m_2 = 2m - m_1$ cells. Thus, the equation $c = 0$ corresponds to the standard and hence symmetric algorithm.

Theorem 4.3. *Suppose that $c \in [0,1)$ and $\varepsilon \in (1-\sqrt{1-c^2}, 1)$ are fixed. Then, the probability that an asymmetric cuckoo hash of $n = \lfloor (1-\varepsilon)m\rfloor$ data points into two tables of size $m_1 = \lfloor m(1+c)\rfloor$ respectively $m_2 = 2m - m_1$ succeeds, (that is, the corresponding cuckoo graph contains no complex component,) is equal to*

$$1 - \frac{(1-\varepsilon)^3(10-2\varepsilon^3+9\varepsilon^2-3c^2\varepsilon^2+9\varepsilon c^2-15\varepsilon+2c^4-10c^2)}{12(2\varepsilon-\varepsilon^2-c^2)^3(c^2-1)}\frac{1}{m} + \mathcal{O}\left(\frac{1}{m^2}\right). \tag{4.61}$$

See also Kutzelnigg [2008] for a sketch of the proof of this theorem.

Proof of Theorem 4.3

Since Corollary 4.1 already covers cuckoo graphs with a different number of nodes of each type, the only difference to the proof of Theorem 4.2 is the application of the saddle point method. Now, our starting point is the generalised equation

$$\#G^\circ_{m_1,m_2,n} = \frac{-m_1!\,m_2!\,n!}{4\pi^2(m_1+m_2-n)!}\oint\oint \frac{\tilde{t}(x,y)^{m_1+m_2-n}}{\sqrt{1-t_1(x,y)t_2(x,y)}}\frac{dx}{x^{m_1+1}}\frac{dy}{y^{m_2+1}}. \tag{4.62}$$

According to Theorem 3.2, the saddle point is determined by the system consisting of the equations

$$\frac{m_1}{m_1+m_2-n} = \frac{x_0}{\tilde{t}(x_0,y_0)}\frac{\partial}{\partial x}\tilde{t}(x_0,y_0) \quad \text{and} \quad \frac{m_2}{m_1+m_2-n} = \frac{y_0}{\tilde{t}(x_0,y_0)}\frac{\partial}{\partial y}\tilde{t}(x_0,y_0). \tag{4.63}$$

Using Lemma 4.3, the system becomes to

$$\frac{m_1}{m_1+m_2-n} = \frac{t_1(x_0,y_0)}{\tilde{t}(x_0,y_0)} \quad \text{and} \quad \frac{m_2}{m_1+m_2-n} = \frac{t_2(x_0,y_0)}{\tilde{t}(x_0,y_0)}. \tag{4.64}$$

Further, with help of Lemma 4.2, we obtain the following equations:

$$\begin{aligned} \frac{m_1}{m_1+m_2-n} &= \frac{t_1(x_0,y_0)}{t_1(x_0,y_0)+t_2(x_0,y_0)-t_1(x_0,y_0)t_2(x_0,y_0)}, \\ \frac{m_2}{m_1+m_2-n} &= \frac{t_2(x_0,y_0)}{t_1(x_0,y_0)+t_2(x_0,y_0)-t_1(x_0,y_0)t_2(x_0,y_0)}. \end{aligned} \tag{4.65}$$

Solving this system for $t_1(x_0,y_0)$ and $t_2(x_0,y_0)$ exhibits the solution

$$t_1(x_0,y_0) = \frac{n}{m_2} \quad \text{and} \quad t_2(x_0,y_0) = \frac{n}{m_1}. \tag{4.66}$$

Finally, it turns out that the saddle point is given by

$$x_0 = t_1(x_0,y_0)e^{-t_2(x_0,y_0)} = \frac{n}{m_2}e^{-\frac{n}{m_1}} \quad \text{and} \quad y_0 = t_2(x_0,y_0)e^{-t_1(x_0,y_0)} = \frac{n}{m_1}e^{-\frac{n}{m_2}}. \tag{4.67}$$

Again, we introduce the notation

$$\varepsilon' = 1 - \frac{n}{m} = 1 - \frac{\lfloor (1-\varepsilon)m \rfloor}{m}. \tag{4.68}$$

We observe that due to the singularity of the denominator, the saddle point method is only applicable if the relation $1 > t_1(x_0,y_0)t_2(x_0,y_0)$ holds. Hence we obtain the inequality

$$1 > \frac{n}{m_1}\frac{n}{m_2} = \frac{n}{(1+c)m}\frac{n}{(1-c)m} = \frac{(1-\varepsilon')^2}{1-c^2}, \tag{4.69}$$

and finally the condition

$$\varepsilon' > 1 - \sqrt{1-c^2}. \tag{4.70}$$

The cummulants can be calculated in a similar way as the saddle point, but have been computed using a computer algebra system in a half-automatic way. The maple source file is included in the attached CD-Rom, see also Appendix B.

Further, we apply Theorem 3.2 using the setting

$$f \to \tilde{t}, \quad g \to \frac{1}{\sqrt{1-t_1t_2}}, \quad k \to m(1+\varepsilon'), \quad m_1 \to m(1+c), \quad \text{and} \quad m_2 \to m(1-c),$$

to obtain an asymptotic expansion of the double integral of (4.62). In particular, we obtain that this asymptotic expansion possesses a leading coefficient equal to

$$\begin{aligned} &\frac{\tilde{t}(x_0,y_0)^{2m-n}}{2\pi(2m-n)x_0^m y_0^m \sqrt{1-t_1(x_0,y_0)t_2(x_0,y_0)}\sqrt{\kappa_{20}\kappa_{02}-\kappa_{11}^2}} \\ &\qquad = \frac{(1-\varepsilon'^2)^{m(1+\varepsilon')}e^{2m(1-\varepsilon')}(1-c)^{m(1+c)}(1+c)^{m(1-c)}\sqrt{1+\varepsilon'}}{2\pi m(1-c^2)^{m(1+\varepsilon')}(1-\varepsilon')^{2m}\sqrt{(1-\varepsilon')(1-c^2)}}, \end{aligned} \tag{4.71}$$

and the coefficient of $1/m$ is given by

$$C = \frac{-10 - 45\varepsilon' + 74\varepsilon'^2 + 10\varepsilon'^2 c^4 - 6\varepsilon'^3 c^4 + 2c^4 - 23\varepsilon'^3 - 44\varepsilon'^4 + 41\varepsilon'^5}{12(-1+\varepsilon')(c^2 - 2\varepsilon' + \varepsilon'^2)^3} + \frac{-12\varepsilon'^6 + \varepsilon'^7 - 2\varepsilon' c^6 - 6\varepsilon' c^2 - 16\varepsilon'^3 c^2 + 24\varepsilon'^4 c^2 - 8\varepsilon'^5 c^2}{12(-1+\varepsilon')(c^2 - 2\varepsilon' + \varepsilon'^2)^3}. \quad (4.72)$$

Additionally, we obtain by using Stirling's formula (3.23) the asymptotic expansion

$$\begin{aligned}\frac{m_1!\, m_2!\, n!}{(m_1+m_2-n)!} &= \frac{(m(1+c))!\,(m(1-c))!\,(m(1-\varepsilon))!}{(m(1+\varepsilon'))!} \\ &= \frac{2\pi m(1+c)^{m(\varepsilon'+c)}(1-c)^{m(\varepsilon'-c)} m^{2m\varepsilon'}(1-\varepsilon')^{m(1-\varepsilon')} m^{m(1-\varepsilon')}\sqrt{(1-\varepsilon')(1-c^2)}}{(1+\varepsilon')^{m(1+\varepsilon')} m^{m(1+\varepsilon')} e^{2m(1-\varepsilon')}\sqrt{1+\varepsilon'}} \\ &\quad \times \left(1 + \frac{1+\varepsilon'-\varepsilon' c^2 - \varepsilon'^2}{6(1-\varepsilon'^2)(1-c^2)} + \mathcal{O}\left(\frac{1}{m^2}\right)\right). \quad (4.73)\end{aligned}$$

Combining this results we finally obtain the asymptotic expansion

$$\begin{aligned}\frac{\#G^\circ_{m(1+c),m(1-c),(1-\varepsilon')m}}{(m^2(1-c^2))^{m(1-\varepsilon')}} &= 1 + \frac{C}{(1+\varepsilon')m} + \frac{1+\varepsilon'-\varepsilon' c^2-\varepsilon'^2}{6(1-\varepsilon'^2)(1-c^2)} + \mathcal{O}\left(\frac{1}{m^2}\right) \\ &= 1 - \frac{(1-\varepsilon')^3(10 - 2\varepsilon'^3 + 9\varepsilon'^2 - 3c^2\varepsilon'^2 + 9\varepsilon' c^2 - 15\varepsilon' + 2c^4 - 10c^2)}{12(2\varepsilon' - \varepsilon'^2 - c^2)^3(c^2-1)m} + \mathcal{O}\left(\frac{1}{m^2}\right), \quad (4.74)\end{aligned}$$

of the number of graphs without complex components.

We now replace ε' by $\varepsilon = \varepsilon' + \mathcal{O}(1/m)$ and obtain the claimed result.

4.5 Comparison and conclusion

In this section, we discuss the influence of this chapter's results on the practical application of cuckoo hashing. First of all, we compare the results concerning the simplified algorithm with the standard data structure. The upper diagram of Figure 4.1 displays the factor of $1/m$ of the asymptotic expansion given in Theorem 4.1 resp. 4.2, depending on ε. It can be easily seen, that the failure probability of the simplified version is slightly increased for all ε. However, both functions exhibit the same behaviour as ε tends to zero. Further, the different behaviour for small load factors (that is ε close to one), is not of strong influence, since the failure rate is still small. Thus, the practical performance of this two algorithms is almost identical.

The second diagram of Figure 4.1 depicts the influence of an increasing asymmetry factor c. The leftmost curve corresponds to an asymmetry $c = 0$ and, hence, to the standard algorithm. The further curves correspond to asymmetry factors $c = 0.2$, $c = 0.3$, and $c = 0.4$ from left to right. We observe that the failure probability increases as the asymmetry increases. Especially the reduced maximum load factor is easily notified. This is a major weak point of asymmetric cuckoo hashing and one has to investigate very carefully if other properties justify the usage of this algorithm. This will be done in Chapter 7, together with an analysis of search and insertion operations.

One might argue, that the previous conclusions are based on asymptotic expansions only, thus it is not certain if the observations hold for practical relevant settings. To overcome this weak point, we present numerical results in Table 4.1. From the results given in the table, we find that our asymptotic results can be understood as a good approximation as long as the load is not too close to the critical value. We notify that our approximation tends to overestimate the number of failures if the tables are small and possess relatively high load, and that the accuracy increases with the size of the tables of the investigated data structure. Further, we notice that cuckoo hashing is very reliable, except if the load is close to the critical value. Additionally, the reliability of the algorithm increases as the table size is raised for fixed and noncritical ε.

As for the theoretical analysis, the numerical data show that the number of unsuccessful constructions is influenced by the asymmetry of the tables in use. The higher the difference of the numbers of storage cells of the two tables is, the higher is the chance of a failure. However the influence is barley noticeable if the data structure holds few keys only. Concerning simplified cuckoo hashing, we observe the expected behaviour, but we want to emphasise once more that there is only a slight increase of the number of unsuccessful constructed table of about five percent.

The presented theorems of this chapter imply strong restrictions on the load factor of the data structure by a lower bounds on ε. What happens if this bound will we exceeded? Numerically obtained data show, that the failure rate increases very fast as the number of keys inserted in the table violates the restriction on the load factor. Table 4.1 provides such data concerning asymmetric cuckoo hashing. Note that an asymmetry factor $c = 0.3$ leads to a lower bound of $\varepsilon_{\min} \approx 0.046$. Further, we obtain a minimal value $\varepsilon_{\min} \approx 0.083$ corresponding to an asymmetry of $c = 0.4$. Thus, the settings $(c = 0.3, \varepsilon = 0.04)$, $(c = 0.4, \varepsilon = 0.06)$ and $(c = 0.4, \varepsilon = 0.04)$ are slightly beyond the critical ratio. We observe that the number of unsuccessful constructed tables is by far greater than outside this critical range. The failure rate increases as the size of the data structure increases in contrast to the behaviour of non critical inputs. It might be possible to store more than the critical number of keys in a cuckoo hash table, but one cannot expect a significant higher fill ratio. A further analysis of this "critical case" can be found in Chapter 5.

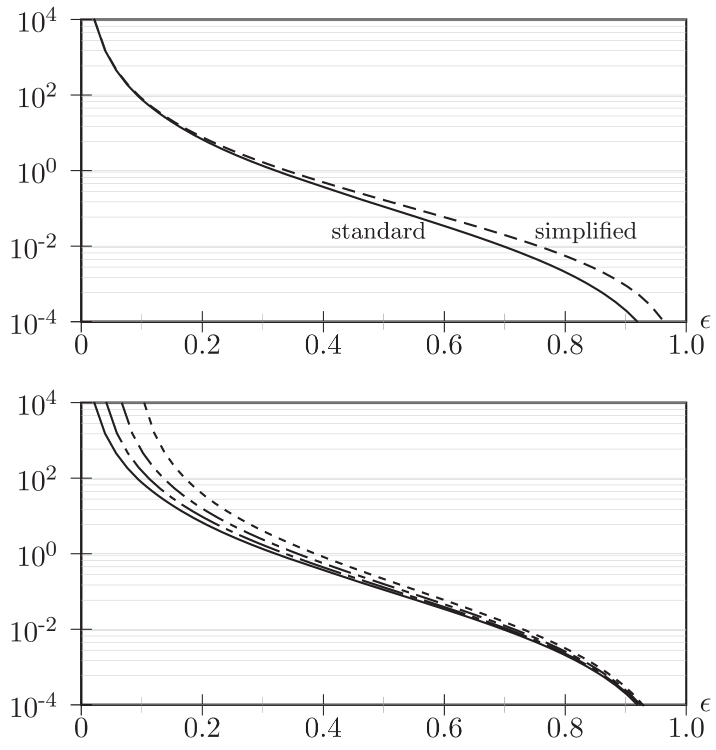


Figure 4.1: The first coefficient of the asymptotic expansion of the failure probability for various cuckoo hash algorithms. The upper diagram displays the curves corresponding to the standard algorithm and the simplified version. The second diagram shows the curves corresponding to asymmetry factors $c = 0$ (*i.e.* the standard algorithm), $c = 0.2$, $c = 0.3$, and $c = 0.4$ from left to right.

m	$\varepsilon = 0.4$		$\varepsilon = 0.2$		$\varepsilon = 0.1$		$\varepsilon = 0.06$		$\varepsilon = 0.04$	
	data	exp.	data	exp.	data	exp.	data	exp.	data	exp.
standard cuckoo hashing										
$5{\cdot}10^3$	38	36.5	649	772	5070	7606	14467	40078	25624	144036
10^4	16	18.2	288	336	3046	3803	9954	20039	19857	72018
$5{\cdot}10^4$	1	3.65	62	67.2	676	761	3127	4008	8439	14403
10^5	1	1.82	31	33.6	362	380	1753	2004	5210	7202
$5{\cdot}10^5$	0	0.36	7	6.7	76	76	398	401	1284	1440
asymmetric cuckoo hashing, $c = 0.1$										
$5{\cdot}10^3$	31	38.1	653	730	5578	8940	16589	52455	29598	216847
10^4	14	19.1	384	365	3368	4470	11853	26228	23654	108423
$5{\cdot}10^4$	2	3.81	66	73	867	894	4024	5246	11096	21685
10^5	3	1.91	43	36.5	435	447	2160	2623	6957	10842
$5{\cdot}10^5$	1	0.38	3	7.3	97	89	461	525	1926	2168
asymmetric cuckoo hashing, $c = 0.2$										
$5{\cdot}10^3$	30	43.8	858	951	8165	15423	25663	141571	45728	500000
10^4	16	21.9	456	476	5185	7712	19759	70785	40465	500000
$5{\cdot}10^4$	5	4.38	81	95.1	1388	1542	8272	14157	25929	122500
10^5	3	2.19	55	47.6	683	771	4930	7079	19836	61250
$5{\cdot}10^5$	0	0.44	10	9.5	165	154	1291	1416	7731	12250
asymmetric cuckoo hashing, $c = 0.3$										
$5{\cdot}10^3$	67	56	1336	1575	16392	51955	53808	500000	95818	-
10^4	26	28	725	787	11793	25977	49823	500000	102239	-
$5{\cdot}10^4$	3	5.6	154	157	3894	5195	36907	342931	123219	-
10^5	3	2.8	81	78.7	2187	2598	29834	171466	135348	-
$5{\cdot}10^5$	1	0.56	8	15.7	532	520	14957	34293	182111	-
asymmetric cuckoo hashing, $c = 0.4$										
$5{\cdot}10^3$	74	82.6	2958	3850	49422	500000	153957	-	243458	-
10^4	42	41.3	1673	1925	44773	500000	182475	-	302164	-
$5{\cdot}10^4$	12	8.26	373	385	30758	192279	286374	-	459241	-
10^5	6	4.13	176	193	24090	96139	351809	-	492741	-
$5{\cdot}10^5$	2	0.83	29	38.5	10627	19228	485515	-	500000	-
simplified cuckoo hashing										
$5{\cdot}10^3$	44	49.2	710	767	5272	8100	15276	41589	26802	147600
10^4	27	24.6	386	383	3122	4050	10451	20795	20536	73800
$5{\cdot}10^4$	5	4.92	87	76.7	737	810	3414	4159	8666	14760
10^5	3	2.46	32	38.3	417	405	1831	2079	5323	7380
$5{\cdot}10^5$	0	0.49	7	7.7	85	81	417	416	1358	1476

Table 4.1: Number of failures during the construction of $5{\cdot}10^5$ cuckoo hash tables. The table provides numerical results (data) as well as the expected number of failures (exp.) calculated using the corresponding asymptotic expansion. Blank entries refer to supercritical settings, where our asymptotic approximations are not applicable. We use a pseudo random generator to simulate good hash functions. Further information concerning the setup and implementation details can be found in Chapter 9 resp. Appendix A.

Chapter 5

The “Critical Case”

5.1 Introduction

The previous analysis of cuckoo hashing requested a load factor strictly below 0.5, that is the case $\varepsilon = 0$. In this chapter, we consider the asymptotic behaviour of standard and simplified cuckoo hashing using this critical load. Further, it is also interesting what happens beyond this bound and how many keys can be stored using cuckoo hashing if we proceed until an error occurs.

5.2 Simplified cuckoo hashing

Theorem 5.1. *The probability that a simplified cuckoo hash of m data points into a table of size $2m$ succeeds, (that is, the corresponding cuckoo graph contains no complex component,) is equal to*

$$\sqrt{\frac{2}{3}} + \mathcal{O}(1). \tag{5.1}$$

A proof of this theorem can be found in Flajolet et al. [1989]. See also Janson et al. [1993] for further results.

5.3 Standard cuckoo hashing

Theorem 5.2. *The probability $p(m,m)$ that a cuckoo hash of m data points into two tables of size m succeeds, (that is, the corresponding cuckoo graph contains no complex component,) is equal to*

$$p(m,m) = \sqrt{\frac{2}{3}} + \mathcal{O}(1). \tag{5.2}$$

This case is more delicate since limiting the saddle point $x_0 = 1/e$ coalesces with the singularity of the denominator in (4.44). (Note that $t_1(1/e, 1/e) = 1$.) Hence we expect a phase transition where the singularity behaviour of the denominator gets more and more important. Definitely, this is a difficult analytic problem and therefore not easy to handle. In particular, one has to handle the singularity structure of $t_1(x,y)t_2(x,y)$ around $x = 1/e$

and $y = 1/e$, which is surely feasible, but the choice of the double contour integral is not clear.

For the sake of shortness we will only work out the limiting case $\varepsilon = 0$, that is, $m = n$. Even in this case we do not work directly with the representation (4.44) but apply Lagrange's inversion formula first. In particular, we use the fact that $t_1(x, y)$ satisfies the equation $t_1 = x \exp\left(ye^{t_1}\right)$. This leads again to a saddle point integral, where the denominator is explicit and does not contain implicitly defined functions as before. The following proof is divided in two parts. We start collecting intermediate results, that will be required in the second part, that proofs the claimed property.

Intermediate Results

Lemma 5.1.

$$\#G^{\circ}_{m,m,m} = (m!)^2 \sum_{k\geq 0} \binom{2k}{k} \frac{1}{4^k} [x^m y^m] \tilde{t}(x,y)^m t_1(x,y)^k t_2(x,y)^k. \tag{5.3}$$

Proof. We use Corollary 4.1 and the series expansion

$$\frac{1}{\sqrt{1-z}} = \sum_{k\geq 0} \binom{2k}{k} \frac{1}{4^k} z^k. \tag{5.4}$$

□

Further, we define the functions

$$f(u,y) = (u + ye^u(1-u)) \exp\left(ye^u\right), \tag{5.5}$$

$$l(u,y) = u^k \left(ye^u\right)^k, \tag{5.6}$$

and

$$h(u,y) = u \frac{mu - mye^u u^2 + ku + kye^u + ku^2 - ku^2 ye^u}{u\left(u + ye^u(1-u)\right)}. \tag{5.7}$$

Lemma 5.2.

$$[x^m y^m] \tilde{t}(x,y)^m t_1(x,y)^k t_2(x,y)^k = \frac{1}{m} [u^m y^m] f(u,y)^m l(u,y) h(u,y). \tag{5.8}$$

Proof. Let the function $\phi(u,y)$ be defined as $\phi(u,y) = \exp\left(ye^u\right)$. The generating function $t_1(x,y)$ satisfies the relation $t_1(x,y) = x\phi\left(t_1(x,y), y\right)$ due to its definition. Further, if we set

$$g(u,y) = \left(u + ye^u(1-u)\right)^m u^k \left(ye^u\right)^k, \tag{5.9}$$

the relation

$$\tilde{t}(x,y)^m t_1(x,y)^k t_2(x,y)^k = g(t_1(x,y), y) \tag{5.10}$$

holds. Note that $\phi(0,y) \neq 0$, so that we can apply Lagrange's Inversion Theorem and obtain:

$$\begin{aligned} [x^m y^m] \tilde{t}(x,y)^m t_1(x,y)^k t_2(x,y)^k &= [x^m y^m] g(t_1(x,y), y) \\ &= [u^{m-1} y^m] \frac{1}{m} \phi(u,y)^m \frac{\partial}{\partial u} g(u,y) \\ &= [u^{m-1} y^m] \frac{1}{m} \left(\exp\left(ye^u\right)\right)^m \frac{\partial}{\partial u} \Big(\left(u + ye^u(1-u)\right)^m u^k \left(ye^u\right)^k \Big). \end{aligned} \tag{5.11}$$

The differentiation and some simplifications finalise the proof. □

Obviously, the coefficient $[u^m y^m]f(u,y)^m l(u,y)h(u,y)$ equals zero if k is greater than m. The saddle point method grants us access to the coefficient for "small" k. The following lemma fills the gap.

Lemma 5.3. *Assume that $k \geq m^{\frac{1}{3}+\varepsilon}$ is satisfied for a positive ε. Then, there exists a positive constant c such that*

$$[x^m y^m]\tilde{t}(x,y)^m t_1(x,y)^k t_2(x,y)^k = \mathcal{O}\left(e^{2m-cm^\varepsilon}\right) \tag{5.12}$$

holds.

Proof. Note that the bound

$$[x^m y^m]\tilde{t}(x,y)^m t_1(x,y)^k t_2(x,y)^k \leq r^{-2m}\tilde{t}(r,r)^m t_1(r,r)^k t_2(r,r)^k \tag{5.13}$$

holds for all r satisfying $0 < r < 1/e$. We set $r = (1-\eta)/e$. Recall that $t_1(x,x)$ equals the usual tree function $t(x)$ and that $\tilde{t}(x,x) = 2t(x) - t^2(x)$ holds. Further, the singular expansion of $t(x)$ around its singularity $1/e$ is well known to be (*cf.* Flajolet and Sedgewick [2008])

$$t(x) = 1 - \sqrt{2}\sqrt{1-ex} + \frac{2}{3}(1-ex) - \frac{11}{18\sqrt{2}}(1-ex)^{\frac{3}{2}} + \mathcal{O}\left((1-ex)^2\right). \tag{5.14}$$

Thus, we obtain the inequality

$$[x^m y^m]\tilde{t}(x,y)^m t_1(x,y)^k t_2(x,y)^k \leq e^{2m+\frac{4}{3}\sqrt{2}m\eta^{\frac{3}{2}}-2\sqrt{2}\eta^{\frac{1}{2}}k+\mathcal{O}(m\eta^2)+\mathcal{O}(\eta k)}. \tag{5.15}$$

Setting $\eta = m^{-2/3}$ allows us to compute the claimed bound. □

The coefficient $[u^m y^m]f(u,y)^m l(u,y)h(u,y)$ is closely connected to the Lommel functions of second kind (*cf.* Prudnikov et al. [1989]) if k is "small" (*i.e.* $k = \mathcal{O}(m^{1/3+\varepsilon})$).

Definition 5.1. Let $\mu+\nu+1 \neq 0$, $\mu-\nu+1 \neq 0$, $\frac{\mu-\nu+3}{2} \notin \mathbb{Z}\setminus\mathbb{N}$, and $\frac{\mu+\nu+3}{2} \notin \mathbb{Z}\setminus\mathbb{N}$. The Lommel function of first kind is defined via a hypergeometirc function as

$$s_{\mu,\nu}(x) = \frac{x^{\mu+1}{}_1F_2\left(1; \frac{\mu-\nu+3}{2}, \frac{\mu+\nu+3}{2}; -\frac{1}{4}x^2\right)}{(\mu+\nu+1)(\mu-\nu+1)}. \tag{5.16}$$

If further $\frac{1+\mu-\nu}{2} \notin \mathbb{Z}\setminus\mathbb{N}$ and $\frac{1+\mu+\nu}{2} \notin \mathbb{Z}\setminus\mathbb{N}$, the Lommel function of second kind is defined as

$$\begin{aligned} S_{\mu,\nu}(x) = s_{\mu,\nu}(x) &+ 2^{\mu-1}\Gamma\left(\frac{\mu+\nu+1}{2}\right)\Gamma\left(\frac{\mu+\nu+1}{2}\right) \\ &\times\left(J_\nu(x)\sin\frac{(\mu-\nu)\pi}{2} - Y_\nu(x)\cos\frac{(\mu-\nu)\pi}{2}\right), \end{aligned} \tag{5.17}$$

where $J_\nu(x)$ and $Y_\nu(x)$ denote Bessel functions.

Hereby, the Bessel functions $J_\nu(x)$ of the first kind are defined as the solutions to the differential equation

$$x^2\frac{d^2y}{dx^2}+x\frac{dy}{dx}+(x^2-\nu^2)y=0. \tag{5.18}$$

which are nonsingular at the origin. Additionally, the Bessel functions $Y_\nu(x)$ of the second kind are the solutions to the differential equation which are singular at the origin, see Abramowitz and Stegun [1970]. Further, note that the Lommel functions of first and second kind are solutions of the inhomogeneous Bessel differential equation (*cf.* Zwillinger [1992])

$$x^2\frac{d^2y}{dx^2}+x\frac{dy}{dx}+(x^2-\nu^2)y=x^{\mu+1}. \tag{5.19}$$

Lemma 5.4 (Integral representation of $S_{1,2/3}(x)$)**.** *Let x be a positive real number. Then the following relation holds,*

$$S_{1,2/3}(x)=1+\left(\frac{2}{x}\right)^{\frac{2}{3}}\int\limits_0^\infty \exp\left(-u^3-3\left(\frac{x}{2}\right)^{\frac{2}{3}}u\right)u\,du. \tag{5.20}$$

Proof. (Sketch) We denote the right hand side of (5.20) by $g(x)$. Our goal is to show that this function satisfies equation (5.18). Therefore, we start calculating its first and second derivative with respect to x:

$$\frac{dg(x)}{dx}=-\int\limits_0^\infty\left(\frac{2^{\frac{5}{3}}}{3}x^{-\frac{2}{3}}+2u\right)\exp\left(-u^3-3\left(\frac{x}{2}\right)^{\frac{2}{3}}u\right)u\,du, \tag{5.21}$$

$$\frac{d^2g(x)}{dx^2}=\int\limits_0^\infty\frac{2}{9}\left(5\cdot 2^{\frac{2}{3}}x^{-\frac{2}{3}}+15u+9\cdot 2^{\frac{1}{3}}x^{\frac{2}{3}}u^2\right)\exp\left(-u^3-3\left(\frac{x}{2}\right)^{\frac{2}{3}}u\right)u\,du. \tag{5.22}$$

After plugging in this results into equation (5.20), we conclude that this equation holds if and only if the expression

$$\int\limits_0^\infty\left(2^{\frac{2}{3}}x^{\frac{4}{3}}+\frac{4}{3}u+2^{\frac{4}{3}}x^{\frac{2}{3}}u^2\right)\exp\left(-u^3-3\left(\frac{x}{2}\right)^{\frac{2}{3}}u\right)u\,du=\frac{4}{9} \tag{5.23}$$

holds. The latter equation can be verified easily with help of its antiderivative

$$-\frac{2}{9}\left(2+3\cdot 2^{\frac{1}{3}}x^{\frac{2}{3}}u\right)\exp\left(-u^3-3\left(\frac{x}{2}\right)^{\frac{2}{3}}u\right). \tag{5.24}$$

□

Lemma 5.5. *Assume that $k = \mathcal{O}\big(m^{1/3+\varepsilon}\big)$ holds. Then, there exists a positive real constant c such that*

$$\begin{aligned}\frac{1}{m}&[u^m y^m] f(u,y)^m l(u,y) h(u,y)\\ &= \frac{\sqrt{2\pi}}{2\pi^2}\frac{e^{2m}}{m^{\frac{7}{6}}}\int_0^\infty s e^{-\frac{2}{3}s^3 - \frac{ks}{\sqrt[3]{m}}} \sin\left(\sqrt{3}ksm^{-\frac{1}{3}} + \frac{\pi}{3}\right) ds\\ &\quad + \mathcal{O}\left(e^{2m-cm^{-\frac{1}{12}}}\right) + \mathcal{O}\left(m^{-\frac{7}{6}-\frac{1}{24}} e^{2m}\int_0^\infty s e^{-\frac{2}{3}s^3-\frac{ks}{\sqrt[3]{m}}} ds\right)\end{aligned} \tag{5.25}$$

holds. We can rewrite the integral in terms of Lommel functions and obtain that the main contribution equals

$$\frac{\sqrt{2\pi}}{2\pi^2}\frac{kie^{2m}}{m^{\frac{3}{2}}}\left(S_{1,\frac{2}{3}}\left(i\frac{4k^{\frac{3}{2}}}{3\sqrt{m}}\right) - S_{1,\frac{2}{3}}\left(-i\frac{4k^{\frac{3}{2}}}{3\sqrt{m}}\right)\right). \tag{5.26}$$

Proof. In this proof, c_i denotes real positive constants. We use Lemma 5.2 and Cauchy's Formula and obtain

$$[x^m y^m]\tilde{t}(x,y)^m t_1(x,y)^k t_2(x,y)^k = -\frac{1}{m}\frac{1}{4\pi^2}\oint\oint \frac{f(u,y)^m l(u,y) h(u,y)}{u^{m+1}y^{m+1}} dy\, du. \tag{5.27}$$

The corresponding saddle point is obtained using the equation system

$$\frac{\partial}{\partial u}(\log f(u,y) - \log u - \log y) = 0, \quad \frac{\partial}{\partial y}(\log f(u,y) - \log u - \log y) = 0, \tag{5.28}$$

obtaining the solution $u_0 = 1, y_0 = 1/e$. As in Theorem 3.2, the cummulants are

$$\kappa_{ij} = \left[\frac{\partial^i}{\partial\sigma^i}\frac{\partial^j}{\partial\tau^j}\log f(u_0 e^\sigma, y_0 e^\tau)\right]_{(\sigma,\tau)=(0,0)}, \tag{5.29}$$

particularly we obtain $\kappa_{20} = 0, \kappa_{11} = 0, \kappa_{02} = 1, \kappa_{30} = -4, \kappa_{21} = -1, \kappa_{12} = 0$, and $\kappa_{03} = 1$. In fact, we cannot proceed in the same way as in Theorem 3.2, because the determinant $\Delta = \kappa_{20}\kappa_{02} - \kappa_{11}^2$ equals zero. Instead, we perform the substitution

$$y = e^{i\tau-1},\ u = \begin{cases} e^{\zeta\sigma} & \text{if } \Im(u) \geq 0 \\ e^{\bar{\zeta}\sigma} & \text{if } \Im(u) < 0 \end{cases} \text{where } \zeta = e^{i\frac{2\pi}{3}}. \tag{5.30}$$

We concentrate on the path in the upper half plane (the second part is similar) and obtain the integral

$$I = -\frac{i\zeta}{4\pi^2}\frac{1}{m}\int_0^{\frac{2\pi}{\sqrt{3}}}\int_{-\pi}^{\pi} \frac{f\big(e^{\zeta\sigma}, e^{i\tau-1}\big)^m l\big(e^{\zeta\sigma}, e^{i\tau-1}\big) h\big(e^{\zeta\sigma}, e^{i\tau-1}\big)}{e^{\zeta\sigma m} e^{(i\tau-1)m}} d\tau\, d\sigma. \tag{5.31}$$

Let $\alpha \in (0, 2\pi/\sqrt{3})$ and $\beta \in (0,\pi)$. This choice will be discussed later. We divide the integral in two parts

$$I_1 = -\frac{i\zeta}{4\pi^2}\frac{1}{m}\int_0^{\alpha}\int_{-\beta}^{\beta} \frac{f\big(e^{\zeta\sigma}, e^{i\tau-1}\big)^m l\big(e^{\zeta\sigma}, e^{i\tau-1}\big) h\big(e^{\zeta\sigma}, e^{i\tau-1}\big)}{e^{\zeta\sigma m} e^{(i\tau-1)m}} d\tau\, d\sigma, \tag{5.32}$$

and $I_2 = I - I_1$. We continue calculating an approximation of I_1 but before, we give an upper bound on I_2.

Consider the modulus of the function $f\left(e^{\zeta\sigma}, e^{i\tau-1}\right) e^{-\zeta\sigma} e^{1-i\tau}$ if (σ,τ) is an element of $\left[0, 2\pi/\sqrt{3}\right] \times [-\pi,\pi]$. A plot can be found in Figure 5.1. The function is unimodal and attains it maximum at $(\sigma,\tau) = (0,0)$. This can be verified using the mean value theorem, a bound on the derivative, and the evaluation of the function on points of a sufficient small grid. Further, a local expansion of this function around the origin is given

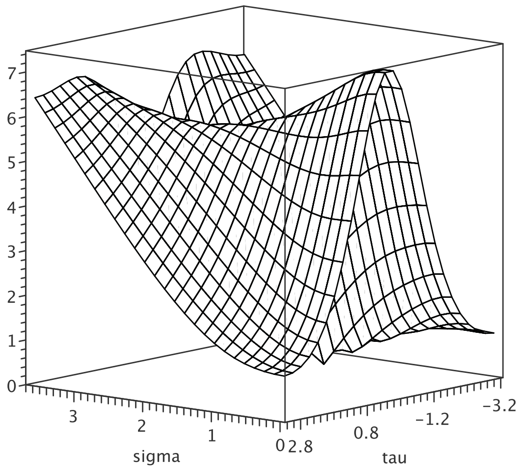


Figure 5.1: The modulus of $f\left(e^{\zeta\sigma}, e^{i\tau-1}\right) e^{-\zeta\sigma} e^{1-i\tau}$ in the relevant area.

by $\exp\left(2 - \frac{1}{2}\tau^2 - \frac{2}{3}\sigma^3 + O(\sigma^2\tau)\right)$. We choose

$$\alpha = m^{-\frac{7}{24}} \text{ and } \beta = m^{-\frac{11}{24}}, \tag{5.33}$$

such that $\alpha^3 m$ and $\beta^2 m$ tend to infinity as m goes to infinity, but on the other hand the terms of higher order like $\alpha^2\beta m$ tend to zero. Hence, we conclude that the bound

$$\left|\frac{f\left(e^{\zeta\sigma}, e^{i\tau-1}\right)}{e^{\zeta\sigma} e^{i\tau-1}}\right|^m \le e^{2m - c_0 m^{-\frac{1}{8}}} \tag{5.34}$$

holds, if $\sigma \in \left[\alpha, 2\pi/\sqrt{3}\right]$ and $|\tau| \in [\beta,\pi]$ are satisfied. Thus, we further obtain the bound

$$|I_2| = \mathcal{O}\left(e^{2m - c_0 m^{-\frac{1}{8}}}\right). \tag{5.35}$$

Next, we calculate an approximation of I_1 with help of the following local expansions: (replace ζ by $\overline{\zeta}$ for the second case)

$$f\Big(e^{\frac{\zeta s}{\sqrt[3]{m}}}, e^{\frac{it}{\sqrt{m}}-1}\Big)^m = \exp\left(m + \zeta s m^{\frac{2}{3}} + it\sqrt{m} - \frac{1}{2}t^2 - \frac{2}{3}s^3 + \mathcal{O}\left(m^{-\frac{1}{24}}\right)\right), \tag{5.36}$$

$$l\Big(e^{\frac{\zeta s}{\sqrt[3]{m}}}, e^{\frac{it}{\sqrt{m}}-1}\Big) = \exp\left(\frac{2\zeta}{\sqrt[3]{m}} ks + \mathcal{O}\left(m^{-\frac{3}{24}+\varepsilon}\right)\right), \tag{5.37}$$

$$h\Big(e^{\frac{\zeta s}{\sqrt[3]{m}}}, e^{\frac{it}{\sqrt{m}}-1}\Big) = -2\zeta s m^{\frac{2}{3}}\left(1 + \mathcal{O}\left(m^{-\frac{3}{24}}\right)\right). \tag{5.38}$$

Using this expansions, and assuming that $\varepsilon \leq \frac{1}{12}$ holds, we infer:

$$\begin{aligned}
I_1 &= \frac{-i\zeta}{4\pi^2 m^{\frac{11}{6}}} \int\limits_0^{\alpha\sqrt[3]{m}} \int\limits_{-\beta\sqrt{m}}^{\beta\sqrt{m}} \frac{f\left(e^{\frac{\zeta s}{\sqrt[3]{m}}}, e^{\frac{it}{\sqrt{m}}-1}\right)^m l\left(e^{\frac{\zeta s}{\sqrt[3]{m}}}, e^{\frac{it}{\sqrt{m}}-1}\right) h\left(e^{\frac{\zeta s}{\sqrt[3]{m}}}, e^{\frac{it}{\sqrt{m}}-1}\right)}{e^{\frac{\zeta s}{\sqrt[3]{m}}m} e^{\left(\frac{it}{\sqrt{m}}-1\right)m}} dt\, ds \\
&= \frac{i\zeta^2 e^{2m}}{2\pi^2 m^{\frac{7}{6}}} \int\limits_0^{\alpha\sqrt[3]{m}} \int\limits_{-\beta\sqrt{m}}^{\beta\sqrt{m}} s e^{-\frac{1}{2}t^2 - \frac{2}{3}s^3 + \frac{2\zeta}{\sqrt[3]{m}}ks} \left(1 + \mathcal{O}\left(m^{-\frac{1}{24}}\right)\right) dt\, ds \\
&= \frac{i\zeta^2 e^{2m}}{2\pi^2 m^{\frac{7}{6}}} \int\limits_0^{\alpha\sqrt[3]{m}} \int\limits_{-\beta\sqrt{m}}^{\beta\sqrt{m}} s e^{-\frac{1}{2}t^2 - \frac{2}{3}s^3 + \frac{2\zeta}{\sqrt[3]{m}}ks} dt\, ds + \mathcal{O}\left(m^{-\frac{29}{24}} e^{2m} \int\limits_0^{\infty} s e^{-\frac{2}{3}s^3 - \frac{ks}{\sqrt[3]{m}}} ds\right).
\end{aligned} \tag{5.39}$$

We denote the remaining integral by I_1':

$$I_1' = \int\limits_0^{\alpha\sqrt[3]{m}} \int\limits_{-\beta\sqrt{m}}^{\beta\sqrt{m}} s e^{-\frac{1}{2}t^2 - \frac{2}{3}s^3 + \frac{2\zeta}{\sqrt[3]{m}}ks} dt\, ds. \tag{5.40}$$

Our next step is to "complete the tails":

$$\int\limits_{\beta\sqrt{m}}^{\infty} e^{-\frac{1}{2}t^2} dt = \int\limits_0^{\infty} e^{-\frac{1}{2}(h+\beta\sqrt{m})^2} dh = e^{-\frac{1}{2}\beta^2 m} \int\limits_0^{\infty} e^{-\frac{1}{2}h^2 - h\beta\sqrt{m}} dh \leq \sqrt{\frac{\pi}{2}} e^{-\frac{1}{2}\beta^2 m}, \tag{5.41}$$

$$\begin{aligned}
\left| \int\limits_{\alpha\sqrt[3]{m}}^{\infty} s e^{-\frac{2}{3}s^3 + \frac{2\zeta}{\sqrt[3]{m}}ks} ds \right| &\leq \int\limits_{\alpha\sqrt[3]{m}}^{\infty} s e^{-\frac{2}{3}s^3 - \frac{ks}{\sqrt[3]{m}}} ds \\
&\leq e^{-\frac{2}{3}\alpha^3 m} \left(\int\limits_0^{\infty} h e^{-\frac{2}{3}h^3} dh + \alpha\sqrt[3]{m} \int\limits_0^{\infty} e^{-\frac{2}{3}h^3} dh \right) \leq \left(c_1 + c_2 \alpha\sqrt[3]{m}\right) e^{-\frac{2}{3}\alpha^3 m}.
\end{aligned} \tag{5.42}$$

We apply this bounds and get the representation

$$\begin{aligned}
I_1' &= \left(\int\limits_{-\infty}^{\infty} e^{-\frac{1}{2}t^2} dt + \mathcal{O}\left(e^{-c_3\beta^2 m}\right) \right) \left(\int\limits_0^{\infty} s e^{-\frac{2}{3}s^3 + \frac{2\zeta}{\sqrt[3]{m}}ks} ds + \mathcal{O}\left(e^{-c_4\alpha^3 m}\right) \right) \\
&= \sqrt{2\pi} \int\limits_0^{\infty} s e^{-\frac{2}{3}s^3 + \frac{2\zeta}{\sqrt[3]{m}}ks} ds + \mathcal{O}\left(e^{-c_4\alpha^3 m}\right) + \mathcal{O}\left(e^{-c_3\beta^2 m}\right).
\end{aligned} \tag{5.43}$$

The second part of the integral representing (5.27) has the main contribution

$$-\frac{\sqrt{2\pi} i \bar{\zeta}^2 e^{2m}}{2\pi^2 m^{\frac{7}{6}}} \int\limits_0^{\infty} s e^{-\frac{2}{3}s^3 + \frac{2\bar{\zeta}}{\sqrt[3]{m}}ks} ds \tag{5.44}$$

and the error terms are of the same order as before. □

Proof of Theorem 5.2

All the intermediate results of the previous section allow us the calculate the number of graphs without complex components. Our starting point is Lemma 5.1 and further, with help of Lemma 5.2 we obtain the equation

$$\#G^{\circ}_{m,m,m} = \frac{(m!)^2}{m}\sum_{k\geq 0}\binom{2k}{k}\frac{1}{4^k}[u^my^m]f(u,y)^m l(u,y)h(u,y) \tag{5.45}$$

$$= (m!)^2(S_1+S_2+S_3), \tag{5.46}$$

where the sums S_1, S_2, and S_3 are defined as below. The sum S_3 is easily to handle, because we have already mentioned that all coefficients are zero in the corresponding range of k:

$$S_3 = \frac{1}{m}\sum_{k=m}^{\infty}\binom{2k}{k}\frac{1}{4^k}[u^my^m]f(u,y)^m l(u,y)h(u,y) = 0. \tag{5.47}$$

We proceed with an upper bound of S_2, which is obtained with help of Lemma 5.3:

$$S_2 = \frac{1}{m}\sum_{k=\left\lceil m^{\frac{1}{3}+\varepsilon}\right\rceil}^{m-1}\binom{2k}{k}\frac{1}{4^k}[u^my^m]f(u,y)^m l(u,y)h(u,y) \leq m\,\mathcal{O}\left(e^{2m-cm^{\varepsilon}}\right). \tag{5.48}$$

The last sum is more complicated to deal with. First, we plug in the result of Lemma 5.5:

$$\begin{aligned}
S_1 &= \frac{1}{m}\sum_{k=0}^{\left\lfloor m^{\frac{1}{3}+\varepsilon}\right\rfloor}\binom{2k}{k}\frac{1}{4^k}[u^my^m]f(u,y)^m l(u,y)h(u,y)\\
&= e^{-\frac{\pi i}{6}}\frac{\sqrt{2\pi}}{2\pi^2}\frac{e^{2m}}{m^{\frac{7}{6}}}\sum_{k=0}^{\left\lfloor m^{\frac{1}{3}+\varepsilon}\right\rfloor}\binom{2k}{k}\frac{1}{4^k}\int_0^{\infty} se^{-\frac{2}{3}s^3+\frac{2\zeta}{\sqrt[3]{m}}ks}ds\\
&\quad + e^{\frac{\pi i}{6}}\frac{\sqrt{2\pi}}{2\pi^2}\frac{e^{2m}}{m^{\frac{7}{6}}}\sum_{k=0}^{\left\lfloor m^{\frac{1}{3}+\varepsilon}\right\rfloor}\binom{2k}{k}\frac{1}{4^k}\int_0^{\infty} se^{-\frac{2}{3}s^3+\frac{2\bar{\zeta}}{\sqrt[3]{m}}ks}ds\\
&\quad + \sum_{k=0}^{\left\lfloor m^{\frac{1}{3}+\varepsilon}\right\rfloor}\binom{2k}{k}\frac{1}{4^k}\mathcal{O}\left(e^{2m-cm^{-\frac{1}{12}}}\right)\\
&\quad + \sum_{k=0}^{\left\lfloor m^{\frac{1}{3}+\varepsilon}\right\rfloor}\binom{2k}{k}\frac{1}{4^k}\mathcal{O}\left(m^{-\frac{7}{6}-\frac{1}{24}}e^{2m}\int_0^{\infty} se^{-\frac{2}{3}s^3-\frac{ks}{\sqrt[3]{m}}}ds\right)\\
&= S_{11}+S_{12}+S_{13}+S_{14}.
\end{aligned} \tag{5.49}$$

At this point, it is again convenient to split up the calculation:

$$S_{11} = e^{-\frac{\pi i}{6}}\frac{\sqrt{2\pi}}{2\pi^2}\frac{e^{2m}}{m^{\frac{7}{6}}}\sum_{k=0}^{\left\lfloor m^{\frac{1}{3}+\varepsilon}\right\rfloor}\binom{2k}{k}\frac{1}{4^k}\int_0^{\infty} se^{-\frac{2}{3}s^3+\frac{2\zeta}{\sqrt[3]{m}}ks}ds$$

$$= e^{-\frac{\pi i}{6}} \frac{\sqrt{2\pi}}{2\pi^2} \frac{e^{2m}}{m^{\frac{7}{6}}} \int_0^\infty s e^{-\frac{2}{3}s^3} \sum_{k=0}^{\left\lfloor m^{\frac{1}{3}+\varepsilon} \right\rfloor} \binom{2k}{k} \frac{1}{4^k} e^{\frac{2\zeta}{\sqrt[3]{m}} ks} ds$$
$$= e^{-\frac{\pi i}{6}} \frac{\sqrt{2\pi}}{2\pi^2} \frac{e^{2m}}{m^{\frac{7}{6}}} \left(I_1 + I_2 + I_3\right). \tag{5.50}$$

Once again, we split up and get:

$$|I_1| = \left| \int_0^{m^{-\delta}} s e^{-\frac{2}{3}s^3} \sum_{k=0}^{\left\lfloor m^{\frac{1}{3}+\varepsilon} \right\rfloor} \binom{2k}{k} \frac{1}{4^k} e^{\frac{2\zeta}{\sqrt[3]{m}} ks} ds \right|$$
$$\leq \int_0^{m^{-\delta}} s e^{-\frac{2}{3}s^3} \sum_{k=0}^{\infty} \binom{2k}{k} \frac{1}{4^k} e^{-\frac{1}{\sqrt[3]{m}} ks} ds$$
$$= \int_0^{m^{-\delta}} s e^{-\frac{2}{3}s^3} \left(1 - e^{-\frac{s}{\sqrt[3]{m}}}\right)^{-\frac{1}{2}} ds = \mathcal{O}\left(m^{\frac{1}{6}-\delta}\right). \tag{5.51}$$

During this calculation, we used the bound $1/\sqrt{1-e^{-x}} \leq 1 + 1/\sqrt{x}$. This estimate is also useful for the "tail" I_3:

$$|I_3| = \left| \int_{m^{\frac{1}{3}-\gamma}}^{\infty} s e^{-\frac{2}{3}s^3} \sum_{k=0}^{\left\lfloor m^{\frac{1}{3}+\varepsilon} \right\rfloor} \binom{2k}{k} \frac{1}{4^k} e^{\frac{2\zeta}{\sqrt[3]{m}} ks} ds \right|$$
$$\leq \int_{m^{\frac{1}{3}-\gamma}}^{\infty} s e^{-\frac{2}{3}s^3} \left(1 - e^{-\frac{s}{\sqrt[3]{m}}}\right)^{-\frac{1}{2}} ds$$
$$\leq m^{\frac{1}{6}} \int_{m^{\frac{1}{3}-\gamma}}^{\infty} \sqrt{s} e^{-\frac{2}{3}s^3} ds + \int_{m^{\frac{1}{3}-\gamma}}^{\infty} s^2 e^{-\frac{2}{3}s^3} ds = m^{\frac{1}{6}} \mathcal{O}\left(e^{-cm^{1-3\gamma}}\right). \tag{5.52}$$

The integral I_2 provides the main contribution:

$$I_2 = \int_{m^{-\delta}}^{m^{\frac{1}{3}-\gamma}} s e^{-\frac{2}{3}s^3} \sum_{k=0}^{\left\lfloor m^{\frac{1}{3}+\varepsilon} \right\rfloor} \binom{2k}{k} \frac{1}{4^k} e^{\frac{2\zeta}{\sqrt[3]{m}} ks} ds$$
$$= \int_{m^{-\delta}}^{m^{\frac{1}{3}-\gamma}} s e^{-\frac{2}{3}s^3} \left(\sum_{k=0}^{\infty} \binom{2k}{k} \frac{1}{4^k} e^{\frac{2\zeta}{\sqrt[3]{m}} ks} + \mathcal{O}\left(e^{-m^{\varepsilon-\delta}}\right) \right) ds$$
$$= \int_{m^{-\delta}}^{m^{\frac{1}{3}-\gamma}} s e^{-\frac{2}{3}s^3} \left(1 - e^{\frac{2\zeta}{\sqrt[3]{m}} s}\right)^{-\frac{1}{2}} ds + \mathcal{O}\left(e^{-m^{\varepsilon-\delta}}\right)$$

$$\begin{aligned}
&= \int_{m^{-\delta}}^{m^{\frac{1}{3}-\gamma}} s e^{-\frac{2}{3}s^3} \left(\frac{m^{\frac{1}{6}} e^{\frac{\pi i}{6}}}{\sqrt{2s}} + \mathcal{O}\left(m^{-\frac{\gamma}{2}}\right) \right) ds + \mathcal{O}\left(e^{-m^{\varepsilon-\delta}}\right) \\
&= \frac{m^{\frac{1}{6}} e^{\frac{\pi i}{6}}}{\sqrt{2}} \int_0^\infty \sqrt{s} e^{-\frac{2}{3}s^3} ds + \mathcal{O}\left(m^{-\frac{\gamma}{2}}\right) + \mathcal{O}\left(e^{-m^{\varepsilon-\delta}}\right) \\
&\quad + \mathcal{O}\left(m^{\frac{1}{6}-\delta}\right) + m^{\frac{1}{6}} \mathcal{O}\left(e^{-cm^{1-3\gamma}}\right) \\
&= e^{\frac{\pi i}{6}} \frac{m^{\frac{1}{6}} \sqrt{\pi}}{2\sqrt{3}} + \mathcal{O}\left(e^{-m^{\varepsilon-\delta}}\right) + \mathcal{O}\left(m^{\frac{1}{6}-\delta}\right) + m^{\frac{1}{6}} \mathcal{O}\left(e^{-cm^{1-3\gamma}}\right). \qquad (5.53)
\end{aligned}$$

For instance, we may set $\varepsilon = \frac{1}{12}$, $\delta = \frac{1}{24}$, and $\gamma = \frac{1}{12}$. Thus, we finally obtain:

$$S_{11} = \frac{\sqrt{2} e^{2m}}{4\pi\sqrt{3}m} \left(1 + \mathcal{O}\left(m^{-\delta}\right)\right). \qquad (5.54)$$

The second sum can be handled in the same way. In particular we obtain the same result:

$$S_{12} = \frac{\sqrt{2} e^{2m}}{4\pi\sqrt{3}m} \left(1 + \mathcal{O}\left(m^{-\delta}\right)\right). \qquad (5.55)$$

What is now still missing, are bounds for the remaining sums. These can be straightforward attained for S_{13}:

$$S_{13} = \sum_{k=0}^{\left\lfloor m^{\frac{1}{3}+\varepsilon} \right\rfloor} \binom{2k}{k} \frac{1}{4^k} \mathcal{O}\left(e^{2m - cm^{-\frac{1}{12}}}\right) = \mathcal{O}\left(m^{\frac{1}{3}+\varepsilon} e^{2m - cm^{-\frac{1}{12}}}\right). \qquad (5.56)$$

The sum S_{14} is a bit more complicate to handle, but we can proceed as in the calculation of S_{11}:

$$S_{14} = \sum_{k=0}^{\left\lfloor m^{\frac{1}{3}+\varepsilon} \right\rfloor} \binom{2k}{k} \frac{1}{4^k} \mathcal{O}\left(m^{-\frac{7}{6}-\frac{1}{24}} e^{2m} \int_0^\infty s e^{-\frac{2}{3}s^3 - \frac{ks}{\sqrt[3]{m}}}\right) ds = \mathcal{O}\left(e^{2m} m^{-1-\frac{1}{24}}\right). \qquad (5.57)$$

Putting these results together, we finally obtain the equation

$$\#G^\circ_{m,m,m} = (m!)^2 \frac{\sqrt{2} e^{2m}}{2\pi\sqrt{3}m} \left(1 + \mathcal{O}\left(m^{-\frac{1}{24}}\right)\right) = m^{2m} \sqrt{\frac{2}{3}} \left(1 + \mathcal{O}\left(m^{-\frac{1}{24}}\right)\right). \qquad (5.58)$$

Together with (4.25), this step completes the proof of Theorem 5.2.

5.4 Conclusion

In this chapter we investigated the "critical" value $\varepsilon = 0$, that corresponds to the relation $m = n$. We conclude that the success rate of both simplified and standard cuckoo hashing drops from $1 + \mathcal{O}(1/m)$ to $\sqrt{2/3} + o(1)$. The numerical results given in Table 5.1 exhibit a similar behaviour, see also Table 4.1. Note that the observed number of errors is

$m(=n)$	5000	10000	50000	100000	500000
standard cuckoo hashing	81053	83138	86563	88021	89177
simplified cuckoo hashing	83100	85357	87972	88415	89751

Table 5.1: Number of failures during the construction of $5\cdot 10^5$ cuckoo hash tables possessing a load factor 0.5, that is, $\varepsilon = 0$ holds. According to Theorem 4.1 resp. 4.2, we expect 91752 errors. We use a pseudo random generator to simulate good hash functions. Further information concerning the setup and implementation details can be found in Chapter 9 resp. Appendix A.

m	5000	10000	50000	100000	500000
standard cuckoo hashing	5208	10322	50917	101440	504143
asymmetric c.h., $c = 0.1$	5181	10272	50661	100930	501618
asymmetric c.h., $c = 0.2$	5103	10118	49897	99406	493992
asymmetric c.h., $c = 0.3$	4972	9855	48593	96800	481019
asymmetric c.h., $c = 0.4$	4782	9476	46704	93030	462228
simplified cuckoo hashing	5204	10318	50907	101428	504133

Table 5.2: Average number of edges of the cuckoo graph at the moment when the first bicyclic component occurs. The table provides numerical data obtained over a sample size of $5\cdot 10^5$. We use the same setup as in Table 5.1.

slightly below the expectation calculated using the asymptotic approximation. However, the accuracy increases with increasing size of the table.

It is a well known fact that the structure of a growing[1] (usual) random graph rapidly changes if the number of edges approaches half the number of nodes. In particular, after this so called "phase transition", it is very likely that a "giant component" exists, that is by far lager than any other remaining component. Further details are for instance given in Flajolet et al. [1989], Janson et al. [1993], and Janson et al. [2000]. It seems to be very likely that bipartite random graphs exhibit a similar behaviour, however no details are known so far, *cf.* Blasiak and Durrett [2005].

Table 5.2 provides the numerically obtained average number of edges at the moment before the first bicyclic component is created. Hence, this number equals the average number of keys of a cuckoo hash table at the moment when a newly inserted key produces an endless loop and forces a rehash. From the data given in the table, we conclude that this number is only slightly greater than m for simplified and standard cuckoo hashing. Due to Flajolet et al. [1989], we know that the first bicyclic component of a usual random graph appears at "time" $m+\Theta\left(m^{-2/3}\right)$, what is in accordance with our numerical results. Moreover, we conjure that the same (or a very similar) result holds for the bipartite graph too. Concerning asymmetric cuckoo hashing, we observe once more that that asymmetry decreases the maximum amount of allocated memory, that actually holds keys.

[1]We fix the number of vertices but successively add new edges.

Chapter 6

The Structure of the Cuckoo Graph

6.1 Introduction

The performance of insertion operations using a cuckoo hash data structure is strongly influenced by the properties of the underlying cuckoo graph. For instance, consider the insertion of key into a tree component. The shape of the tree affects the number of kick-out steps necessary to place the new key. In particular, the diameter of the tree (that is the longest distance between to nodes) and the size of the tree component are upper bounds of this number. On the other hand, an insertion operation dealing with a unicyclic component takes at most twice the size of the component's steps, see Devroye and Morin [2003] or Chapter 1 for further details.

This chapter studies the properties of usual and bipartite random cuckoo graphs, such as the expected size of the tree components, the number of cycles, and the number of nodes contained in cyclic components. Based on these results, we provide an upper bound for the construction cost in Chapter 7. Some parameters might also be of interest in other applications, see, *e.g.*, Blasiak and Durrett [2005].

6.2 The non-bipartite Cuckoo Graph

We start our analysis considering usual random graphs related to simplified cuckoo hashing. In what follows, we prove the following results, see also Drmota and Kutzelnigg [2008].

Theorem 6.1. *Suppose that $\varepsilon \in (0,1)$ is fixed and that $n = \lfloor (1-\varepsilon)m \rfloor$. Then a labelled random multigraph with $2m$ vertices and n edges satisfies the following properties.*

1. *The number of unicyclic components with cycle length k has in limit a Poisson distribution $Po(\lambda_k)$ with parameter*

$$\lambda_k = \frac{1}{2k}(1-\varepsilon)^k, \tag{6.1}$$

   and the number of unicyclic components has in limit a Poisson distribution $Po(\lambda)$, too, with parameter

$$\lambda = -\frac{1}{2}\log\varepsilon. \tag{6.2}$$

2. *Denote the number of tree components with k vertices by t_k. Mean and variance of this random variable are asymptotically equal to*

$$m\mu = 2m\frac{k^{k-2}(1-\varepsilon)^{k-1}e^{k(\varepsilon-1)}}{k!}, \tag{6.3}$$

respectively

$$m\sigma^2 = m\mu - m\frac{2e^{2k(\varepsilon-1)}k^{2k-4}(1-\varepsilon)^{2k-3}(k^2\varepsilon^2+k^2\varepsilon-4k\varepsilon+2)}{(k!)^2}. \tag{6.4}$$

Furthermore t_k satisfies a central limit theorem of the form

$$\frac{t_k-\mu}{\sigma} \to N(0,1). \tag{6.5}$$

3. *The number of vertices contained in cycles has in limit the distribution with characteristic function*

$$\phi(s) = \sqrt{\frac{\varepsilon}{1-e^{is}(1-\varepsilon)}}, \tag{6.6}$$

and, hence, expectation is asymptotically given by

$$\frac{1-\varepsilon}{2\varepsilon}, \tag{6.7}$$

and variance by

$$\frac{(1-\varepsilon)}{2\varepsilon^2}. \tag{6.8}$$

4. *Furthermore, the expected value of the number of nodes in unicyclic components is asymptotically given by*

$$\frac{(1-\varepsilon)}{2\varepsilon^2}, \tag{6.9}$$

and its variance by

$$\frac{(1-\varepsilon)(2-\varepsilon)}{2\varepsilon^4}. \tag{6.10}$$

Proof of Theorem 6.1

In this proof, we only consider graphs of $G^{\circ}_{2m,n}$, *i.e.* the set of graphs without complex components, but all results hold for unrestricted random graphs of $G_{2m,n}$ too. This can be easily seen in the following way. Consider a random variable ξ defined on the set $G_{2m,n}$ and ξ', its restriction to $G^{\circ}_{2m,n}$. Further, we denote the corresponding distribution functions by $F_\xi(x)$ resp. $F_{\xi'}(x)$. Obviously, the relation

$$|F_\xi(x) - F_{\xi'}(x)| \le \mathbb{P}(G_{2m,n} \setminus G^{\circ}_{2m,n}) = \mathcal{O}(1/m) \tag{6.11}$$

holds due to Lemma 6.1 and Theorem 4.1.

Lemma 6.1. *Consider a probability space $(\Omega, \mathbb{P})$ and a subset Ω' of Ω such that $\mathbb{P}(\Omega \setminus \Omega') = \mathcal{O}(1/m)$ holds. Further, we define the probability measure $\mathbb{P}'$ using $\mathbb{P}'(A) = \mathbb{P}(A \cap \Omega)/\mathbb{P}(\Omega')$. Then, the relation*

$$|\mathbb{P}(A) - \mathbb{P}'(A)| \leq \mathbb{P}(\Omega \setminus \Omega') = \mathcal{O}\left(\frac{1}{m}\right), \tag{6.12}$$

is valid.

Proof. We start using basic transformations and obtain the equality

$$\mathbb{P}(A) = \mathbb{P}(A \cap \Omega) + \mathbb{P}(A \cap (\Omega \setminus \Omega')) = \mathbb{P}'(A)(1 - \mathbb{P}(\Omega \setminus \Omega')) + \mathbb{P}(A \cap (\Omega \setminus \Omega')). \tag{6.13}$$

Hence we further derive the relation

$$\mathbb{P}(A) - \mathbb{P}'(A) = \mathbb{P}(A \cap (\Omega \setminus \Omega')) - \mathbb{P}'(A)\mathbb{P}(\Omega \setminus \Omega'), \tag{6.14}$$

which implies the claimed inequality. □

Similar to Chapter 4, we define the ratio

$$\varepsilon' = 1 - \frac{n}{m} = 1 - \frac{\lfloor (1-\varepsilon)m \rfloor}{m}. \tag{6.15}$$

Starting with a graph possessing $2m$ nodes and $n = (1-\varepsilon')m$ edges, we increase the parameters synchronously such that the ratio remains unchanged. In other words, we consider an infinite series of graphs defined by parameters like $(2m, (1-\varepsilon')m), (4m, 2(1-\varepsilon')m), (6m, 3(1-\varepsilon')m), \ldots$ and so on.

The further proof of Theorem 6.1 is divided into several parts, each of it proves separately one of the claimed properties. Again, we use a generating function approach. Recalling the generating function

$$g^{\circ}(x, v) = \exp\left(\frac{1}{2v}\tilde{t}(2xv) + \log\frac{1}{\sqrt{1 - t(2xv)}}\right) = \frac{e^{\frac{1}{2v}\tilde{t}(2xv)}}{\sqrt{1 - t(2xv)}}. \tag{6.16}$$

established in Lemma 4.1 that counts graphs without complex components. Now, we introduce an additional variable to "mark" the parameter of interest, see for instance Flajolet and Odlyzko [1990], Flajolet and Sedgewick [2008], and Drmota and Soria [1995, 1997] for further details of this method.

Number of Cycles

Lemma 6.2. *The moment generating function of the limiting distribution of the number of cycles resp. the number of cycles of length k in a graph of $G^{\circ}_{2m,m(1+\varepsilon')}$ is given by*

$$\psi_c(s) = \exp\left(\frac{\log \varepsilon'}{2}(1 - e^s)\right)\left(1 + \mathcal{O}\left(\frac{1}{m}\right)\right), \tag{6.17}$$

resp.

$$\psi_k(s) = \exp\left(-\frac{(1-\varepsilon')^k}{2k}(1 - e^s)\right)\left(1 + \mathcal{O}\left(\frac{1}{m}\right)\right). \tag{6.18}$$

These results hold pointwise for any fixed real number s, as $m \to \infty$.

Proof. We start with the calculation of the total number of cycles. Hence, we introduce a new variable w, that marks each cyclic component. Thus, (6.16) becomes to

$$g_c^\circ(x,v,w) = \exp\left(\frac{1}{2v}\tilde{t}(2xv) + w\log\frac{1}{\sqrt{1-t(2xv)}}\right) = \frac{\exp\left(\frac{1}{2v}\tilde{t}(2xv)\right)}{(1-t(2xv))^{w/2}}. \tag{6.19}$$

Clearly, the equation $g_c^\circ(x,v,1) = g^\circ(x,v)$ is valid. Now, it's straightforward to calculate the k-th factorial moment

$$\frac{[x^m v^n]\left[\frac{\partial^k}{\partial w^k} g_c^\circ(x,v,w)\right]_{w=1}}{[x^m v^n] g_c^\circ(x,v,1)}, \tag{6.20}$$

and, hence, expectation and variance, accordingly. Moreover, we can identify the limiting distribution with help of the moment generating function

$$\psi_c(s) = \frac{[x^{2m}v^n] g_c^\circ(x,v,e^s)}{[x^{2m}v^n] g_c^\circ(x,v,1)}. \tag{6.21}$$

Since the number of tree components equals $2m-n$, it is straightforward to eliminate the variable v and we obtain the equation

$$\left[\frac{x^{2m}v^n}{(2m)!n!}\right] g_c^\circ(x,v,e^s) = \frac{2^n n!(2m)!}{(2m-n)!}[x^{2m}]\frac{\tilde{t}(x)^{2m-n}}{(1-t(x))^{e^s/2}}. \tag{6.22}$$

As in Chapter 4, we use Cauchy's formula and the saddle point method to obtain an asymptotic expansion. Note that we are able to use the same saddle point. However, the calculation is easier because it is sufficient to calculate the leading term using the saddle point method as developed in the proof of Theorem 3.1 although not all technical conditions are satisfied. Nevertheless, the inequality

$$\left|(1-t(x))^{-e^s/2}\right| \le (1-t(x_0))^{-e^s/2}, \tag{6.23}$$

that is satisfied on the line of integration, and conclude that the bound (3.35) still holds. Furthermore, since $e^s = \mathcal{O}(1)$, the Taylor expansion (3.37) is still applicable, and hence we obtain the same result. In the further calculation, we can concentrate our analysis on the modified term, that is $1/(1-t(x))^{e^s/2}$ instead of $1/\sqrt{1-t(x)}$, because of cancellation. Thus we obtain the moment generating function

$$\psi_c(s) = \frac{\sqrt{1-t(x_0)}}{(1-t(x_0))^{e^s/2}}\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right) = \varepsilon'^{(1-e^s)/2}\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right), \tag{6.24}$$

what completes the proof of the lemma's first part.

The proof of the second part can be obtained analogously. Note that the generating function of a component containing a cycle of length k is given by $t(x)^k/2k$. We just use the generating function

$$g_k^\circ(x,v,w) = \frac{\exp\left(\frac{1}{2v}\tilde{t}(2xv) + (w-1)\frac{1}{2k}t(2xv)^k\right)}{\sqrt{1-t(2xv)}}, \tag{6.25}$$

in which cycles of length k are marked by w and get the equation

$$\left[\frac{x^{2m}v^n}{(2m)!n!}\right] g_k^\circ(x,v,e^s) = \frac{2^n n!(2m)!}{(2m-n)!}[x^{2m}]\frac{\exp\left((e^s-1)\frac{1}{2k}t(x)^k\right)}{\sqrt{1-t(x)}}\tilde{t}(x)^{2m-n}. \tag{6.26}$$

Hence the moment generating function equals

$$\psi_k(s) = \frac{[x^{2m}v^n]g_k^\circ(x,v,e^s)}{[x^{2m}v^n]g_k^\circ(x,v,1)} = \exp\left((e^s-1)\frac{1}{2k}t(x_0)^k\right)\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right). \tag{6.27}$$

□

Note that the moment generating function of a Poisson distribution with parameter λ is given by $\exp(-\lambda(1-e^s))$. Further, the results hold true too if we replace ε' by ε because of the relation

$$\varepsilon' = \varepsilon + \mathcal{O}\left(\frac{1}{m}\right). \tag{6.28}$$

Hence we obtain the same limiting distributions if we consider the series of graphs possessing $2m$ nodes and $\lfloor(1-\varepsilon)m\rfloor$ edges as m tends arbitrary to infinity. Together with Lemma 6.1, this proves the first statement of Theorem 6.1.

Due to the additivity of the Poisson distribution, it is not surprising that the parameters are related. To be more precise, the equation

$$\sum_{k=1}^{\infty}\frac{(1-\varepsilon)^k}{2k} = \frac{1}{2}\log\varepsilon \tag{6.29}$$

holds.

Trees with fixed size

For simplification, we introduce the following notation:

Definition 6.1. Let k denote a natural number and suppose that $\varepsilon \in (0,1)$ is fixed. Then, we define the numbers

$$\mu = 2\frac{k^{k-2}(1-\varepsilon)^{k-1}e^{k(\varepsilon-1)}}{k!}, \tag{6.30}$$

and

$$\sigma^2 = \mu - \frac{2e^{2k(\varepsilon-1)}k^{2k-4}(1-\varepsilon)^{2k-3}(k^2\varepsilon^2+k^2\varepsilon-4k\varepsilon+2)}{(k!)^2}. \tag{6.31}$$

Lemma 6.3. *The number of tree components with k vertices of a randomly chosen member of $G^\circ_{2m,(1-\varepsilon')m}$ possesses mean*

$$m\mu + \mathcal{O}(1) \tag{6.32}$$

and variance

$$m\sigma^2 + \mathcal{O}(1). \tag{6.33}$$

Proof. The proof of this result is more difficult to obtain than the proof of the previous lemma, although we apply the same principle. We need the number of unrooted trees of size k, denoted by $\tilde{t}_k$. Note that the number of rooted trees possessing k node equals k^{k-1}. Since there exist exactly k possibilities to select the root, we conclude that the equation $\tilde{t}_k = k^{k-2}$ holds if k is greater than one. Further, it is clear that there exists exactly one unrooted tree possessing a single node and hence the previous formula holds for this special case too. Now we introduce a variable w to mark tree components of size k and obtain the generating function

$$g_t^\circ(x,v,w) = \frac{\exp\left(\frac{1}{v}\tilde{t}(2xv) + (w-1)\frac{1}{k!}\tilde{t}_k x^k (2v)^{k-1}\right)}{\sqrt{1-t(2xv)}}, \tag{6.34}$$

that allows us to calculate the $l-$th factorial moment as follows

$$\mathcal{M}_l = \frac{[x^{2m}v^n]\left[\frac{\partial^l}{\partial w^l} g_t^\circ(x,v,w)\right]_{w=1}}{[x^{2m}v^n] g_t^\circ(x,v,1)}. \tag{6.35}$$

The term included in the numerator simplifies to

$$\begin{aligned}
[x^{2m}v^n]\left[\frac{\partial^l}{\partial w^l} g_t^\circ(x,v,w)\right]_{w=1} &= [x^{2m}]2^{2m}\left[\frac{\partial^l}{\partial w^l}\frac{\left(\tilde{t}(x) + (w-1)\frac{1}{k!}\tilde{t}_k x^k\right)^{2m-n}}{(2m-n)!\sqrt{1-t(x)}}\right]_{w=1} \\
&= [x^{2m}]2^{2m}\frac{\tilde{t}(x)^{2m-n-l}}{(2m-n)!\sqrt{1-t(x)}}(2m-n)^{\underline{l}}\left(\frac{\tilde{t}_k}{k!}x^k\right)^l.
\end{aligned} \tag{6.36}$$

As usual we use Theorem 3.1 to calculate an asymptotic expansion. Hence, we obtain that the leading term of $\mathcal{M}_l$ equals

$$\frac{(2m-n)^{\underline{l}}}{\tilde{t}(x_0)^l}\left(\frac{\tilde{t}_k}{k!}x_0^k\right)^l = \frac{2^l m^l (1+\varepsilon)^l}{(1-\varepsilon^2)^l}\left(\frac{k^{k-2}}{k!}(1-\varepsilon')^k e^{(\varepsilon'-1)k}\right)^l\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right). \tag{6.37}$$

In particular, that proofs the first claim of this lemma. Further, we conclude that the variance is of order $O(m)$ too, and that its calculation requires to determine the next term of the asymptotic expansion. Now, we are doing this in a semi-automatic way using Maple and obtain

$$\begin{aligned}
\mathcal{M}_l = &\left(2m\frac{k^{k-2}(1-\varepsilon')^{k-1}}{k! e^{(1-\varepsilon')k}}\right)^l \\
&\times\left(1+\frac{(l\varepsilon'^2k^2 + \varepsilon'k - 4l\varepsilon'k + l\varepsilon'k^2 + 2l + 2 - 3k)l}{4(\varepsilon'-1)m} + \mathcal{O}\left(\frac{1}{m^2}\right)\right).
\end{aligned} \tag{6.38}$$

Finally, we conclude that the variance equals

$$m\sigma^2 = \mathcal{M}_2 - \mathcal{M}_1^2 + \mathcal{M}_1 = \left(2m\frac{k^{k-2}(1-\varepsilon')^{k-1}}{k! e^{(1-\varepsilon')k}}\right)^2 \frac{\varepsilon'^2k^2 - 4\varepsilon'k + \varepsilon'k^2 + 2}{2(\varepsilon'-1)} + m\mu, \tag{6.39}$$

that completes the proof. □

Similar to previous calculation, we may now replace ε' by ε. As a direct consequence of above's result, we obtain the following lemma.

Lemma 6.4. *The number of tree components contained in a randomly chosen member of $G^\circ_{2m,(1-\varepsilon')m}$ that possess k vertices, divided by m, converges towards μ in probability.*

Proof. Denote the number of tree components containing k nodes by t_k. Using Markov's inequality, we obtain

$$\mathbb{P}\left(\left|\frac{t_k}{m} - m\right| \geq m^{-1/3}\right) \leq \frac{\sigma^2}{m^2 m^{-2/3}}\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right) = \mathcal{O}\left(m^{-1/3}\right), \tag{6.40}$$

and, hence, the claimed result. □

However, it is possible to obtain a more precise result.

Lemma 6.5. *The number of tree components of size k of a randomly selected member of $G^\circ_{2m,(1-\varepsilon')m}$ minus $m\mu$ and divided by $\sqrt{m(1+\varepsilon')}$, has in limit the characteristic function*

$$\phi_t(r) = \exp\left(-\frac{\sigma^2}{2m(1+\varepsilon)}r^2\right)\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right). \tag{6.41}$$

This equation holds pointwise for any fixed real number r, as $m \to \infty$.

Proof. In this proof, we use of the shortened denotation $M = 2m-n = m(1+\varepsilon')$. Similar to previous calculations, we obtain the characteristic function

$$\phi_t(r) = \frac{[x^{2m}v^n]g_t^\circ\left(x, v, e^{\frac{ir}{\sqrt{M}}}\right)}{[x^{2m}v^n]g_t^\circ(x, v, 1)} e^{-i\frac{m\mu}{\sqrt{M}}r}, \tag{6.42}$$

and continue calculating

$$[x^{2m}v^n]g_t^\circ\left(x, v, e^{\frac{ir}{\sqrt{M}}}\right) = [x^{2m}]2^{2m}\frac{\tilde{t}(x)^M}{M!\sqrt{1-t(x)}}\left(1+\frac{\tilde{t}_k x^k}{k!\tilde{t}(x)}\left(e^{\frac{ir}{\sqrt{M}}}-1\right)\right). \tag{6.43}$$

In principle, this proof is based on the same ideas as the derivation of Theorem 3.1. That means, we use Cauchy's Integral Formula and apply an adopted saddle point method. The main contribution of the integral corresponds again to the arc satisfying $|s| \leq \alpha = M^{-\frac{1}{2}+\delta}$, where $0 < \delta < \frac{1}{6}$ holds. In this range, the Taylor expansion

$$\begin{aligned}
&M\log\left(1+\frac{\tilde{t}_k x_0^k e^{isk}}{k!\tilde{t}(x_0 e^{is})}\left(e^{\frac{ir}{\sqrt{M}}}-1\right)\right)\\
&\quad= M\frac{\tilde{t}_k x_0^k e^{isk}}{k!\tilde{t}(x_0 e^{is})}\left(e^{\frac{ir}{\sqrt{M}}}-1\right) - \frac{1}{2}M\left(\frac{\tilde{t}_k x_0^k e^{isk}}{k!\tilde{t}(x_0 e^{is})}\right)^2\left(e^{\frac{ir}{\sqrt{M}}}-1\right)^2 + \mathcal{O}\left(M^{-\frac{1}{2}}\right)\\
&\quad= c_0 ir\sqrt{M} - c_0\frac{r^2}{2} - c_1 rs\sqrt{M} + c_0^2\frac{r^2}{2} + \mathcal{O}\left(M^{-\frac{1}{2}+2\delta}\right)
\end{aligned} \tag{6.44}$$

holds, where we used the notation

$$c_i = \left[\frac{\partial^i}{\partial u^i}\frac{\tilde{t}_k x_0^k e^{ku}}{k!\,\tilde{t}(x_0 e^u)}\right]_{u=0}. \tag{6.45}$$

Particularly, we get

$$c_0 = \frac{\mu}{1+\varepsilon'} \quad \text{and} \quad c_1 = \frac{\mu}{1+\varepsilon'}\left(k - \frac{2}{1+\varepsilon'}\right). \tag{6.46}$$

Thus the integral can be rewritten as

$$\int_{-\alpha}^{\alpha} e^{-\kappa_2 \frac{s^2}{2} M + c_0 ir\sqrt{M} - c_0 \frac{r^2}{2} - c_1 rs\sqrt{M} + c_0^2 \frac{r^2}{2}} \left(1 + \mathcal{O}\left(M^{-\frac{1}{2}+2\delta}\right)\right) ds$$

$$= \frac{1}{\sqrt{M}} e^{c_0 ir\sqrt{M} - c_0 \frac{r^2}{2} + c_0^2 \frac{r^2}{2}} \int_{-\alpha\sqrt{M}}^{\alpha\sqrt{M}} e^{-\kappa_2 \frac{u^2}{2} - c_1 ru} \left(1 + \mathcal{O}\left(M^{-\frac{1}{2}+2\delta}\right)\right) du. \tag{6.47}$$

First, consider the part of the integral containing the big-$\mathcal{O}$ term and note that the bound

$$\left| \int_{-\alpha\sqrt{M}}^{\alpha\sqrt{M}} e^{-\kappa_2 \frac{u^2}{2} - c_1 ru} \mathcal{O}\left(M^{-\frac{1}{2}+2\delta}\right) du \right| \le \mathcal{O}\left(M^{-\frac{1}{2}+2\delta}\right) \int_{-\infty}^{\infty} e^{-\kappa_2 \frac{u^2}{2} - c_1 ru} = \mathcal{O}\left(M^{-\frac{1}{2}+2\delta}\right) du \tag{6.48}$$

is valid. Next, we introduce the abbreviatory notation $\nu = \sqrt{\kappa_2}\alpha\sqrt{M} + \frac{c_1\rho}{\sqrt{\kappa_2}}$ and conclude that

$$\int_{\alpha\sqrt{M}}^{\infty} e^{-\frac{\kappa_2}{2}u^2 - rc_1 u} du = \frac{1}{\sqrt{\kappa_2}} e^{\frac{c_1^2 r^2}{\kappa_2}} \int_{\nu}^{\infty} e^{-\frac{v^2}{2}} dv \tag{6.49}$$

holds. Due to Lemma 3.2, the tails are exponentially small. Finally, we conclude that the main contribution along the arc $(-\alpha, \alpha)$ equals

$$e^{c_0 ir\sqrt{M} - c_0 \frac{r^2}{2} + c_0^2 \frac{r^2}{2}} \int_{-\infty}^{\infty} e^{-\kappa_2 \frac{u^2}{2} - c_1 ru} du = \sqrt{2\pi} \exp\left(c_0 ir\sqrt{M} - c_0 \frac{r^2}{2} + c_0^2 \frac{r^2}{2} + c_1^2 \frac{r^2}{2\kappa_2}\right). \tag{6.50}$$

It remains to show that the contribution outside this arc is negligible. First consider the situation outside the range $(-\eta, \eta)$, where η denotes a number satisfying $0 < \eta < 1$. Since x_0 is the unique maximum of the modulus of $\tilde{t}(x)$ and $e^{ir/\sqrt{M}} - 1 = \mathcal{O}\left(1/\sqrt{M}\right)$ holds, it is clear that a $\gamma > 0$ exists such that

$$\left| \tilde{t}(x_0 e^{is}) + \frac{\tilde{t}_k x_0^k e^{isk}}{k!} \left(e^{\frac{ir}{\sqrt{M}}} - 1\right) \right| \le (1-\gamma)\tilde{t}(x_0) \tag{6.51}$$

holds. Hence we conclude that the contribution is exponentially small. For the remaining range, we consider again a Taylor expansion (*cf.* (6.44)) and obtain also an exponentially small contribution. □

Nodes in cycles

The distribution of the number of nodes in cycles is rather easy to obtain. Here we do not take the non-root nodes of trees attached to cycles into account. As usual, we use the variable w to mark the parameter of interest and conclude the generating function modifies to

$$g_n^\circ(x,v,w) = \frac{\exp\left(\frac{1}{2v}\tilde{t}(2xv)\right)}{\sqrt{1-wt(2xv)}}. \tag{6.52}$$

Similar to the calculation of the number of cycles, we apply Theorem 5.5, but again we have to take care of the slightly modified conditions. In particular, we make use of the bound

$$\left|\frac{1}{\sqrt{1-e^{is}t(x)}}\right| \le \frac{1}{\sqrt{|1|-|e^{is}t(x)|}} = \frac{1}{\sqrt{\varepsilon'}}, \tag{6.53}$$

that is satisfied on the line of integration. Hence we conclude the contribution outside the arc $(-\alpha,\alpha)$, is still exponential small, *cf.* (3.35). Finally, we obtain that the characteristic function equals

$$\begin{aligned}\phi_n(s) = \frac{[x^{2m}v^n]g_n^\circ(x,v,e^{is})}{[x^{2m}v^n]g_n^\circ(x,v,1)} &= \sqrt{\frac{1-t(x_0)}{1-e^{is}t(x_0)}}\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right),\\ &= \sqrt{\frac{\varepsilon'}{1-e^{is}(1-\varepsilon')}}\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right) = \sqrt{\frac{\varepsilon}{1-e^{is}(1-\varepsilon)}}\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right),\end{aligned} \tag{6.54}$$

and it is straightforward to determine expectation and variance.

Using the series expansion

$$\sqrt{\frac{\varepsilon}{1-e^{is}(1-\varepsilon)}} = \sqrt{\varepsilon}\sum_{k\ge0}\binom{-\frac{1}{2}}{k}(-1)^k(1-\varepsilon)^k e^{isk} = \sqrt{\varepsilon}\sum_{k\ge0}\frac{1{\cdot}3{\cdot}5\cdots(2k-1)}{2^k k!}(1-\varepsilon)^k e^{isk}, \tag{6.55}$$

we also obtain, that the probability, that exactly k nodes are contained in cycles, equals

$$\frac{1{\cdot}3{\cdot}5\cdots(2k-1)}{2^k k!}\sqrt{\varepsilon}(1-\varepsilon)^k, \tag{6.56}$$

in limit.

Nodes in cyclic components

Here, we also count nodes of trees attached to cycles, in contrast to the previous section. Thus we obtain the generating function

$$g_v^\circ(x,v,w) = \frac{\exp\left(\frac{1}{2v}\tilde{t}(2xv)\right)}{\sqrt{1-t(2xvw)}}. \tag{6.57}$$

Hence, it is straightforward to calculate mean and variance using the first and second derivation with respect to w. This completes the proof of Theorem 6.1.

6.3 The bipartite Cuckoo Graph

Next, we draw our attention to bipartite graphs that are related to standard cuckoo hashing. The following theorem shows us, that the structure of this graph is similar to the structure of its non-bipartite counterpart. A detailed discussion of the differences and similarities can be found at the end of this chapter.

Theorem 6.2. *Suppose that $\varepsilon \in (0,1)$ is fixed and that $n = \lfloor (1-\varepsilon)m \rfloor$. Then a labelled random bipartite multigraph with $2 \times m$ vertices and n edges satisfies the following properties.*

1. *The number of unicyclic components with cycle length $2k$ has in limit a Poisson distribution $Po(\lambda_k)$ with parameter*
$$\lambda_k = \frac{1}{2k} (1-\varepsilon)^{2k}, \tag{6.58}$$
and the number of unicyclic components has in limit a Poisson distribution $Po(\lambda)$, too, with parameter
$$\lambda = -\frac{1}{2} \log \left(1 - (1-\varepsilon)^2\right). \tag{6.59}$$

2. *Denote the number of tree components with k vertices by t_k. Mean and variance of this random variable are asymptotically equal to*
$$m\mu = 2m \frac{k^{k-2}(1-\varepsilon)^{k-1} e^{k(\varepsilon-1)}}{k!}, \tag{6.60}$$
respectively
$$m\sigma^2 = m\mu - \frac{2me^{2k(\varepsilon-1)} k^{2k-4} (1-\varepsilon)^{2k-3} (k^2\varepsilon^2 + k^2\varepsilon - 4k\varepsilon + 2)}{(k!)^2}. \tag{6.61}$$
Furthermore t_k satisfies a central limit theorem of the form
$$\frac{t_k - \mu}{\sigma} \to N(0,1). \tag{6.62}$$

3. *The number of vertices contained in cycles has in limit the distribution with characteristic function*
$$\phi(s) = \sqrt{\frac{1-(1-\varepsilon)^2}{1-e^{2is}(1-\varepsilon)^2}}, \tag{6.63}$$
and, hence, expectation is asymptotically given by
$$\frac{(1-\varepsilon)^2}{1-(1-\varepsilon)^2}, \tag{6.64}$$
and variance by
$$\frac{2(1-\varepsilon)^2}{\left(1-(1-\varepsilon)^2\right)^2}. \tag{6.65}$$

4. *Furthermore, the expected value of the number of nodes in unicyclic components is asymptotically given by*
$$\frac{(1-\varepsilon)^2}{\varepsilon \left(1-(1-\varepsilon)^2\right)}, \tag{6.66}$$
and its variance by
$$\frac{(1-\varepsilon)^2(\varepsilon^2 - 3\varepsilon + 4)}{\varepsilon^2 \left(1-(1-\varepsilon)^2\right)^2}. \tag{6.67}$$

Proof of Theorem 6.2

Similar to the proof of Theorem 6.1, it is sufficient to consider graphs of $G^{\circ}_{m,m,n}$, the set of bipartite graphs without complex components, only. Given a random variable ξ, defined on the set $G_{m,m,n}$, we denote its restriction to $G^{\circ}_{m,m,n}$ by ξ' and the corresponding distribution functions by F_ξ resp. $F_{\xi'}$. Due to Theorem 4.2 and Lemma 6.1, the relation

$$|F_\xi - F_{\xi'}| \le \mathbb{P}(G_{m,m,n} \setminus G^{\circ}_{m,m,n}) = \mathcal{O}(1/m) \tag{6.68}$$

holds.

As usual, we define the ratio

$$\varepsilon' = 1 - \frac{n}{m} = 1 - \frac{\lfloor (1-\varepsilon)m \rfloor}{m}, \tag{6.69}$$

and consider an infinite series of graphs possessing fixed ε' first (*cf.* Chapter 4).

The further proof is divided into four parts, each of it proves one of the claimed results using a generating function approach. Recall the function

$$g^{\circ}(x,v) = \exp\left(\frac{1}{v}\tilde{t}(xv,yv) + \frac{1}{2}\log\frac{1}{1-t_1(x,y)t_2(x,y)}\right) = \frac{e^{\frac{1}{v}\tilde{t}(xv,yv)}}{\sqrt{1-t_1(xv,yv)t_2(xv,yv)}}. \tag{6.70}$$

established in Lemma 4.5 that counts graphs without complex components. Again, we use the technique of introducing a new variable to "mark" the parameter of interest.

Number of Cycles

Lemma 6.6. *The moment generating function of the limiting distribution of the number of cycles resp. the number of cycles of length $2k$ in a graph of $G^{\circ}_{m,m,n}$ is given by*

$$\psi_c(s) = \exp\left(\frac{\log\left(1-(1-\varepsilon')^2\right)}{2}(1-e^s)\right)\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right), \tag{6.71}$$

resp.

$$\psi_{2k}(s) = \exp\left(-\frac{(1-\varepsilon')^{2k}}{2k}(1-e^s)\right)\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right). \tag{6.72}$$

These results hold pointwise for any fixed real number s, as $m \to \infty$.

Proof. We start considering the number of all cycles, hence we attach w once to each cyclic component, that leads us to the generating function

$$\begin{aligned} g^{\circ}_c(x,y,v,w) &= \exp\left(\frac{1}{v}\tilde{t}(xv,yv) + \frac{w}{2}\log\frac{1}{1-t_1(x,y)t_2(x,y)}\right) \\ &= \frac{\exp\left(\frac{1}{v}\tilde{t}(xv,yv)\right)}{\left(1-t_1(xv,yv)t_2(xv,yv)\right)^{w/2}}. \end{aligned} \tag{6.73}$$

Clearly, the equation $g^{\circ}_c(x,y,v,1) = g^{\circ}(x,y,v)$ is valid. Hence, the moment generating function is given by

$$\psi_c(s) = \frac{[x^m y^m v^n] g^{\circ}(x,y,v,e^s)}{[x^m y^m v^n] g^{\circ}(x,y,v,1)}. \tag{6.74}$$

Again, the number of tree components equals $2m-n$, thus the generating function simplifies to

$$\left[\frac{x^m y^m v^n}{(m!)^2 n!}\right] g_c^\circ(x,y,v,e^s) = \frac{n!(m!)^2}{(2m-n)!}[x^m y^m] \frac{\tilde{t}(x,y)^{2m-n}}{(1-t_1(x,y)t_2(x,y))^{e^s/2}}. \tag{6.75}$$

We continue using Cauchy's formula and the double saddle point method as described in Theorem 3.2. Similar to the univariate case, the method is applicable for fixed s. Hence we further obtain, that the equation

$$\begin{aligned}\psi_c(s) &= \frac{\sqrt{1-t_1(x_0,y_0)t_2(x_0,y_0)}}{(1-t_1(x_0,y_0)t_2(x_0,y_0))^{e^s/2}}\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right)\\ &= \left(1-(1-\varepsilon')^2\right)^{(1-e^s)/2}\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right)\end{aligned} \tag{6.76}$$

holds, what completes the proof of the first part of the lemma.

The proof of the second part is very similar, we just replace g_c° by the generating function

$$g_k^\circ(x,y,v,w) = \frac{\exp\left(\frac{1}{v}\tilde{t}(xv,yv) + (w-1)\frac{1}{2k}t_1(xv,yv)^k t_2(xv,yv)^k\right)}{\sqrt{1-t_1(xv,yv)t_2(xv,yv)}}. \tag{6.77}$$

Hereby, w is used to mark cycles of length $2k$. Recall that the generating function of a component containing a cycle of length $2k$ is given by $\frac{1}{2k}t_1(x,y)^k t_2(x,y)^k$, see (4.41). We proceed as usual and yield

$$\begin{aligned}&\left[\frac{x^m y^m v^n}{(m!)^2 n!}\right] g_k^\circ(x,y,v,e^s)\\ &\qquad = \frac{n!(m!)^2}{(2m-n)!}[x^m y^m]\frac{\exp\left((e^s-1)\frac{1}{2k}t_1(x,y)^k t_2(x,y)^k\right)}{\sqrt{1-t_1(x,y)t_2(x,y)}}\tilde{t}(x,y)^{2m-n}.\end{aligned} \tag{6.78}$$

Finally, the moment generating function equals

$$\psi_k(s) = \frac{[x^m y^m v^n] g_k^\circ(x,y,v,e^s)}{[x^m y^m v^n] g_k^\circ(x,y,v,1)} = \exp\left((e^s-1)\frac{1}{2k}t_1(x_0,y_0)^k t_2(x_0,y_0)^k\right)\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right), \tag{6.79}$$

and get the claimed results. □

Similar to the results obtained for the usual graph, these moment generating functions correspond to Poisson distributions. Note that we may again replace ε' by ε, because of the relation

$$\varepsilon' = \varepsilon + \mathcal{O}\left(\frac{1}{m}\right). \tag{6.80}$$

Together with Lemma 6.1, this proves the first statement of Theorem 6.2.

Once more, there exists an additive relation between the parameters, illustrated by the equation

$$\sum_{k=1}^{\infty}\frac{(1-\varepsilon)^{2k}}{2k} = \frac{1}{2}\log(1-(1-\varepsilon)^2). \tag{6.81}$$

Trees with fixed size

The proof of this result is more complicated, because the parameters depend on m. In what follows, we make use of the generating function of a bipartite tree component that possesses exactly k nodes of both types. Because of Lemma 4.2, we can write this function as

$$\tilde{t}_k(x,y) = \sum_{m_1+m_2=k} m_1^{m_2-1} m_2^{m_1-1} \frac{x^{m_1}}{m_1!} \frac{y^{m_2}}{m_2!}. \tag{6.82}$$

The following lemmata provide more detailed information about this function.

Lemma 6.7.

$$\tilde{t}_k(x_0,x_0) = \sum_{l=0}^{k} l^{k-l-1}(k-l)^{l-1} \frac{x_0^k}{l!\,(k-l)!} = 2k^{k-2}\frac{x_0^k}{k!}. \tag{6.83}$$

Proof. We apply Lagrange's Inversion Formula to obtain the coefficient of x^k in $\tilde{t}(x,x) = 2t(x) - t(x)^2$, where $t(x)$ denotes the usual tree function that satisfies $t(x) = x\exp(t(x))$. Because of the previous relation, it is also clear that the number of unrooted bipartite trees possessing k nodes equals twice the number of unrooted (usual) trees of size k. □

Lemma 6.8.

$$\left[\frac{\partial}{\partial u}\tilde{t}_k(x_0e^u, x_0e^v)\right]_{(0,0)} = x_0^k \sum_{l=0}^{k} l^{k-l}(k-l)^{l-1}\frac{1}{l!\,(k-l)!} = k^{k-1}\frac{x_0^k}{k!}. \tag{6.84}$$

Proof. The proof of this lemma is a simple application of Abel's generalisation of the Binomial Theorem,

$$x^{-1}(x+y+ka)^k = \sum_{l=0}^{k}\binom{k}{l}(x+la)^{l-1}(y+(k-l)a)^{k-l}, \tag{6.85}$$

see, *e.g.*, Riordan [1968]. We set $x \to k$, $y \to k$ and $a \to -1$ and obtain the claimed result. □

For simplification, we introduce as before the following notation:

Definition 6.2. Let k denote a natural number and suppose that $\varepsilon \in (0,1)$ is fixed. Then, we define the numbers

$$\mu = 2\frac{k^{k-2}(1-\varepsilon)^{k-1}e^{k(\varepsilon-1)}}{k!}, \tag{6.86}$$

and

$$\sigma^2 = \mu - \frac{2e^{2k(\varepsilon-1)}k^{2k-4}(1-\varepsilon)^{2k-3}(k^2\varepsilon^2 + k^2\varepsilon - 4k\varepsilon + 2)}{(k!)^2}. \tag{6.87}$$

With help of these preliminary results, we are able to prove the following lemma.

Lemma 6.9. *The number of tree components with k vertices of a randomly chosen member of of $G^{\circ}_{m,m,n}$ possesses mean*

$$m\mu + \mathcal{O}(1) \tag{6.88}$$

and variance

$$m\sigma^2 + \mathcal{O}(1). \tag{6.89}$$

Proof. We start introducing the variable w to mark trees possessing exactly k nodes and obtain the generating function

$$g^{\circ}_t(x,y,v,w) = \frac{\exp\left(\frac{1}{v}\tilde{t}(xv,yv) + (w-1)\tilde{t}_k(xv,yv)\right)}{\sqrt{1 - t_1(xv,yv)t_2(xv,yv)}}, \tag{6.90}$$

that allows us to calculate the $l-$th factorial moment as follows

$$\mathcal{M}_l = \frac{[x^m y^m v^n]\left[\frac{\partial^l}{\partial w^l} g^{\circ}_t(x,y,v,w)\right]_{w=1}}{[x^m y^m v^n] g^{\circ}_t(x,y,v,1)}. \tag{6.91}$$

We further simplify this expression and obtain the equation

$$\begin{aligned}
[x^m y^m v^n]\left[\frac{\partial^l}{\partial w^l} g^{\circ}_t(x,y,v,w)\right]_{w=1} &= [x^m y^m]\left[\frac{\partial^l}{\partial w^l} \frac{\left(\tilde{t}(x,y) + (w-1)\tilde{t}_k(x,y)\right)^{2m-n}}{(2m-n)!\sqrt{1-t(x,y)}}\right]_{w=1} \\
&= [x^{2m}]\frac{\tilde{t}(x,y)^{2m-n-l}}{(2m-n)!\sqrt{1-t(x,y)}}(2m-n)^{\underline{l}}\tilde{t}_k(x,y)^l. \quad (6.92)
\end{aligned}$$

Now, we use once more Theorem 3.2 to calculate an asymptotic expansion. By using Lemma 6.7, we obtain that the leading term of $\mathcal{M}_l$ equals

$$\frac{(2m-n)^{\underline{l}}}{\tilde{t}(x_0,y_0)^l}\tilde{t}_k(x_0,y_0)^l = \frac{m^l(1+\varepsilon)^l}{(1-\varepsilon^2)^l}\left(2\frac{k^{k-2}}{k!}(1-\varepsilon')^k e^{(\varepsilon'-1)k}\right)^l \left(1 + \mathcal{O}\left(\frac{1}{m}\right)\right). \tag{6.93}$$

Hence, we have completed the proof of the first statement. Moreover, we conclude that the variance is of order $O(m)$ too, thus its calculation requires to determine the next term of the asymptotic expansion. Similar to the "simplified" situation, we do this in a semi-automatic way using Maple and obtain the claimed result. See the corresponding worksheet for further details. □

As in previous calculations, we may of course replace ε' by ε. Again, it is possible to establish a central limit theorem.

Lemma 6.10. *The number of tree components of size k of a randomly selected member of $G^{\circ}_{m,m,(1-\varepsilon')m}$ minus $m\mu$ and divided by $\sqrt{m(1+\varepsilon')}$, has in limit the characteristic function*

$$\phi_t(r) = \exp\left(-\frac{\sigma^2}{2m(1+\varepsilon)}r^2\right)\left(1 + \mathcal{O}\left(\frac{1}{m}\right)\right). \tag{6.94}$$

This equation holds pointwise for any fixed real number r, as $m \to \infty$.

Proof. This result is again obtained using an adopted saddle point method, similar to the proof of Lemma 6.5. In the following, we make again use of the shortened denotation $M = 2m - n = m(1+\varepsilon')$. Similar to (6.44), we obtain that the Taylor expansion

$$\begin{aligned} M\log\left(1+\frac{\tilde t_k(x_0e^{is},y_0e^{it})}{\tilde t(x_0e^{is},y_0e^{it})}\left(e^{\frac{ir}{\sqrt M}}-1\right)\right) \\ = M\frac{\tilde t_k(x_0e^{is},y_0e^{it})}{\tilde t(x_0e^{is},y_0e^{it})}\left(e^{\frac{ir}{\sqrt M}}-1\right)-\frac12 M\left(\frac{\tilde t_k(x_0e^{is},y_0e^{it})}{\tilde t(x_0e^{is},y_0e^{it})}\right)^2\left(e^{\frac{ir}{\sqrt M}}-1\right)^2+\mathcal O\left(M^{-\frac12}\right) \\ = c_{00}ir\sqrt M - c_{00}\frac{r^2}{2}-(c_{10}s+c_{01}t)r\sqrt M + c_{00}^2\frac{r^2}{2}+\mathcal O\left(M^{-\frac12+2\delta}\right) \end{aligned} \tag{6.95}$$

holds, if s and t satisfy $|s|,|t| \le \alpha = M^{-\frac12+\delta}$, where $0<\delta<\frac16$ holds. Hereby, we used the notation

$$c_{ij} = \left[\frac{\partial^i}{\partial u^i}\frac{\partial^j}{\partial v^j}\frac{\tilde t_k(x_0e^u,y_0e^v)}{\tilde t(x_0e^u,y_0e^v)}\right]_{(0,0)}. \tag{6.96}$$

In particular, we obtain

$$c_{00} = \frac{\mu}{1+\varepsilon'} \quad\text{and}\quad c_{10}=c_{01}=\frac{\mu}{1+\varepsilon'}\left(k-\frac{2}{1+\varepsilon'}\right). \tag{6.97}$$

Using this Taylor expansion, we proceed as in the proof of Lemma 6.5 respectively Theorem 3.2. □

Nodes in cycles

In this part of the proof, we count the number of nodes contained in cycles, but we do not count the non root nodes of the trees attached to the cycles. Similar to proof of claims concerning the number of cycles, this result is rather easy to obtain. We make use of the generating function

$$g_n^\circ(x,y,v,w) = \frac{\exp\left(\frac1v \tilde t(xv,yv)\right)}{\sqrt{1-w^2t_1(xv,yv)t_2(xv,yv)}}. \tag{6.98}$$

Hence we get the characteristic function

$$\begin{aligned} \phi_n(s) = \frac{[x^my^mv^n]g_n^\circ(x,y,v,e^{is})}{[x^my^mv^n]g_n^\circ(x,y,v,1)} = \sqrt{\frac{1-t_1(x_0,y_0)t_2(x_0,y_0)}{1-e^{2is}t_1(x_0,y_0)t_2(x_0,y_0)}}\left(1+\mathcal O\left(\frac1m\right)\right) \\ = \sqrt{\frac{1-(1-\varepsilon')^2}{1-e^{2is}(1-\varepsilon')^2}}\left(1+\mathcal O\left(\frac1m\right)\right) = \sqrt{\frac{1-(1-\varepsilon)^2}{1-e^{2is}(1-\varepsilon)^2}}\left(1+\mathcal O\left(\frac1m\right)\right). \end{aligned} \tag{6.99}$$

using the double saddle point method, that is again applicable what can be seen as in the univariate case. It is further straightforward to calculate asymptotic mean and variance.

Finally we use the series expansion

$$\begin{aligned} \sqrt{\frac{1-(1-\varepsilon)^2}{1-e^{2is}(1-\varepsilon)^2}} = \sqrt{1-(1-\varepsilon)^2}\sum_{k\ge0}\binom{-\frac12}{k}(-1)^k(1-\varepsilon)^{2k}e^{is2k} \\ = \sqrt{1-(1-\varepsilon)^2}\sum_{k\ge0}\frac{1\cdot3\cdot5\cdots(2k-1)}{2^kk!}(1-\varepsilon)^{2k}e^{is2k}, \end{aligned} \tag{6.100}$$

to infer that the probability that exactly $2k$ nodes are contained in cycles equals

$$\frac{1\cdot 3\cdot 5\cdots(2k-1)}{2^k k!}\sqrt{1-(1-\varepsilon)^2}(1-\varepsilon)^{2k}, \tag{6.101}$$

in limit.

Nodes in cyclic components

If we count the number of all nodes contained in cyclic components, the generating function modifies to

$$g_v^\circ(x,y,v,w)=\frac{\exp\left(\frac{1}{v}\tilde{t}(xv,yv)\right)}{\sqrt{1-t_1(xvw,yvw)t_2(xvw,yvw)}}. \tag{6.102}$$

Thus we kook the nodes of trees attached to cycles into account, in contrast to the previous calculation. Thus, it is still straightforward to calculate asymptotic mean and variance. This completes the proof of Theorem 6.2.

6.4 The asymmetric bipartite Cuckoo Graph

It is even possible to adopt the results of the previous section to the case of an asymmetric bipartite graph. In particular, all results related to the cyclic structure are slightly modified only. However, the calculation of the distribution of tree components is much more complicated.

Theorem 6.3. *Suppose that $c\in[0,1)$ and $\varepsilon\in(1-\sqrt{1-c^2},1)$ are fixed and that $n=\lfloor(1-\varepsilon)m\rfloor$. Then a random labelled bipartite multigraph with $m_1=\lfloor m(1+c)\rfloor$ respectively $m_2=2m-m_1$ vertices and n edges satisfies the following properties.*

1. *The number of unicyclic components with cycle length $2k$ has in limit a Poisson distribution $Po(\lambda_k)$ with parameter*

$$\lambda_k=\frac{1}{2k}\left(\frac{(1-\varepsilon)^2}{1-c^2}\right)^k, \tag{6.103}$$

and the number of unicyclic components has in limit a Poisson distribution $Po(\lambda)$, too, with parameter

$$\lambda=-\frac{1}{2}\log\left(1-\frac{(1-\varepsilon)^2}{1-c^2}\right). \tag{6.104}$$

2. *Let T be defined by the equation*

$$T=\sum_{l=0}^{k} l^{k-l-1}(k-l)^{l-1}\frac{x_0^l y_0^{k-l}}{l!(k-l)!}. \tag{6.105}$$

Similar we define T_1 and T_2 using the equations

$$T_1=\sum_{l=0}^{k} l^{k-l}(k-l)^{l-1}\frac{x_0^l y_0^{k-l}}{l!(k-l)!} \quad \textit{and} \quad T_2=\sum_{l=0}^{k} l^{k-l-1}(k-l)^{l}\frac{x_0^l y_0^{k-l}}{l!(k-l)!}. \tag{6.106}$$

Hereby, (x_0, y_0) denotes the saddle point as calculated in Chapter 4 and is hence given by

$$x_0 = \frac{1-\varepsilon}{1-c} \exp\left(-\frac{1-\varepsilon}{1+c}\right) \quad \textit{and} \quad y_0 = \frac{1-\varepsilon}{1+c} \exp\left(-\frac{1-\varepsilon}{1-c}\right). \tag{6.107}$$

Then, the number of tree components with k vertices possesses asymptotic mean

$$\frac{m(1-c^2)}{1-\varepsilon} T, \tag{6.108}$$

and asymptotic variance

$$\begin{aligned} m\Bigg(& \frac{(T_2 - T + T_1)^2}{(-1+\varepsilon)^3} c^4 - \frac{(-T_1 + T_2)(T_2 - 2T + T_1)}{(-1+\varepsilon)^2} c^3 + \frac{(T_2^2 + T_1^2 + T)\varepsilon^2}{(-1+\varepsilon)^3} c^2 \\ & + \frac{(-4T_1T_2 - 2T - 3T_1^2 - 3T_2^2 + 2TT_1 + 2TT_2)\varepsilon}{(-1+\varepsilon)^3} c^2 + \frac{-2T^2 + 2TT_2 + 2TT_1 + T}{(-1+\varepsilon)^3} c^2 \\ & + \frac{(-T_1 + T_2)(-T_2 + 2\varepsilon T_2 - 2T - T_1 + 2\varepsilon T_1)}{(-1+\varepsilon)^2} c + \frac{(-T + T_1^2 + T_2^2)\varepsilon^2}{(-1+\varepsilon)^3} \\ & + \frac{(-2TT_1 + 2T - T_1^2 + 4T_1T_2 - 2TT_2 - T_2^2)\varepsilon}{(-1+\varepsilon)^3} + \frac{T^2 + T_2^2 - T + T_1^2 - 2T_1T_2}{(-1+\varepsilon)^3}\Bigg). \end{aligned} \tag{6.109}$$

3. *The number of vertices contained in cycles has in limit the distribution with characteristic function*

$$\phi(s) = \sqrt{\frac{1 - \frac{(1-\varepsilon)^2}{1-c^2}}{1 - e^{2is} \frac{(1-\varepsilon)^2}{1-c^2}}}, \tag{6.110}$$

and, hence, expectation is asymptotically given by

$$\frac{(1-\varepsilon)^2}{2\varepsilon - \varepsilon^2 - c^2}, \tag{6.111}$$

and variance by

$$\frac{2(1-\varepsilon)^2(1-c^2)}{(2\varepsilon - \varepsilon^2 - c^2)^2}. \tag{6.112}$$

4. *Furthermore, the expected value of the number of nodes in unicyclic components is asymptotically given by*

$$\frac{(1-\varepsilon)^2(2-\varepsilon-c^2)}{(2\varepsilon - \varepsilon^2 - c^2)^2}, \tag{6.113}$$

and its variance by

$$\frac{2c^6 + (14 + 3\varepsilon^2 - 11\varepsilon)c^4 + (-26 + 31\varepsilon - 11\varepsilon^2 + \varepsilon^4 - \varepsilon^3)c^2 + (\varepsilon^2 - 3\varepsilon + 4)(2-\varepsilon)^2}{(2\varepsilon - \varepsilon^2 - c^2)^4(1-\varepsilon)^{-2}}. \tag{6.114}$$

Proof. The proof of the statements number 1., 3., and 4. is almost identical to the proof of the corresponding statements of Theorem 6.2. The only difference is that the symmetric saddle point satisfying $x_0 = y_0$ is replaced by the asymmetric saddle point given in the theorem. However, this does not influence the actual calculation until the functions are evaluated. Hence it is sufficient to plug in the modified saddle point in (6.76), (6.79), and (6.99) to obtain the claimed results.

On the other hand, the calculation concerning the number of trees of given size becomes much more complicated, although it follows the same principle. This is due to the fact that there exist no simple closed formula for $\tilde{t}_k(x_0, y_0)$ any longer, *cf.* Lemma 6.7. The calculation itself is performed using Maple, see the corresponding worksheet for further information. □

6.5 Comparison and conclusion

In this section, we provide numerical results to support our analysis and compare the graph structures of the different variations of cuckoo hashing. See Chapter 9 for details on implementation and setup. We start considering tree components because of their fundamental impact on the behaviour of the algorithms. Note that the tree structure of the bipartite cuckoo graph possesses the same limiting distribution as the usual cuckoo graph, that is related to the simplified algorithm. Furthermore, we notice that the number of isolated nodes increases as the asymmetry increases. Thus, we expect a better performance of unsuccessful search operations because of the asymmetry.

Tables 6.1 to 6.5 display the average number of trees of size one (*i.e.* isolated nodes), two, and five counted during 10^5 experiments. Each of this tables provides data for one fixed load factor (respectively ε). Further, we consider several variations of cuckoo hashing, including the standard and simplified algorithm, as well as some asymmetric versions. Recall that higher asymmetry leads to a lower maximum load factor (*cf.* Chapter 4), hence some small values for ε are invalid for some asymmetric data structures. From the data given in this tables, we see that our asymptotic results are good approximations for all investigated settings.

Furthermore, these tables provide the numerically obtained average maximum tree component. This number is of interest, because the size of the largest tree component it is a natural bound for the maximum number of steps necessary to insert a key into a tree component. We notice that this number increases as ε decreases. The results of our experiments lead to the conjecture, that this parameter possesses again the same asymptotic behaviour for both standard and simplified cuckoo hashing. Finally, we observe that asymmetry leads to a lager maximum tree component.

Next, we draw our attention on the structure of cyclic components. Note that the corresponding results of Theorem 6.1 and 6.2 are in some sense related, but not identical. Table 6.6 provides numerical data for the number of nodes in cyclic components and the number of cycles. Our experiments, using settings $m = 5{\cdot}10^3$ to $m = 5{\cdot}10^5$, show that the size of the data structure does not have significant influence on this parameters. Because of this, we do not provide data for different table sizes. From the results presented in the table, we see again that the asymptotic results obtained in this chapter provide suitable estimates. We notice that asymmetry leads to an increased number of cycles and nodes in cyclic components. Furthermore, both parameters are higher if we consider the usual cuckoo graph instead of the symmetric bipartite version.

We do not provide further numerical results that would require more information such as the length of the cycle or the number of nodes contained in the cycle itself. This is caused by the fact that this parameters are not of high relevance for cuckoo hashing, and that it would require a different implementation of our software, see Chapter 9. However, we are again interested in the the average size of the maximum cyclic component, because the number of steps that are required to perform an insertion into a cyclic component is bounded by twice the size of the maximum cyclic component. It turns out, that this number increases with the load factor, but usually the size of the maximum tree component is dominant, except is we consider small tables possessing a high load.

		isolated nodes				trees with 2 nodes				trees with 5 nodes				average
$\varepsilon = 0.4$	m	sample mean	rel. error	sample var.	rel. error	sample mean	rel. error	sample var.	rel. error	sample mean	rel. error	sample var.	rel. error	max. tree
	5000	5488	0.005%	676	-1.114%	904	-0.031%	768	-0.755%	67	-0.071%	61	-0.307%	26
standard	10000	10976	0.004%	1339	-0.044%	1808	-0.025%	1528	-0.250%	134	0.019%	122	0.376%	30
cuckoo	50000	54881	0.001%	6681	0.139%	9036	-0.004%	7617	0.048%	672	0.009%	614	-0.426%	39
hashing	100000	109762	0.000%	13341	0.293%	18073	-0.006%	15240	0.008%	1344	0.005%	1233	-0.756%	43
	500000	548810	0.000%	66549	0.526%	90359	-0.001%	76154	0.068%	6721	0.001%	6100	0.305%	54
	5000	5498	0.009%	664	0.343%	893	-0.053%	748	0.487%	67	-0.078%	62	-0.453%	27
asymm.	10000	10996	0.004%	1334	-0.176%	1786	-0.016%	1514	-0.701%	135	0.003%	123	-0.528%	30
cuckoo	50000	54980	0.001%	6661	-0.043%	8927	-0.001%	7524	-0.062%	673	-0.007%	614	0.004%	40
hashing	100000	109961	0.000%	13239	0.584%	17854	-0.003%	15038	0.009%	1347	-0.001%	1220	0.587%	44
$c = 0.1$	500000	549804	0.000%	66289	0.443%	89271	-0.002%	75231	-0.048%	6734	-0.005%	6118	0.289%	54
	5000	5528	0.006%	651	0.793%	860	-0.033%	715	0.834%	68	-0.067%	61	0.574%	28
asymm.	10000	11057	0.004%	1317	-0.388%	1719	-0.016%	1443	-0.097%	135	-0.050%	124	-0.480%	32
cuckoo	50000	55286	0.001%	6588	-0.455%	8596	-0.004%	7218	-0.107%	677	-0.007%	616	0.461%	41
hashing	100000	110572	0.001%	13187	-0.538%	17191	-0.003%	14496	-0.524%	1353	-0.009%	1236	0.075%	46
$c = 0.2$	500000	552864	0.000%	65793	-0.322%	85952	-0.000%	71941	0.220%	6766	0.004%	6215	-0.487%	57
	5000	5582	0.007%	633	0.675%	803	-0.033%	662	0.785%	68	-0.032%	63	-0.493%	30
asymm.	10000	11164	0.002%	1271	0.286%	1605	-0.006%	1330	0.342%	136	-0.084%	125	0.414%	34
cuckoo	50000	55824	-0.001%	6399	-0.392%	8024	0.002%	6723	-0.726%	680	-0.008%	626	-0.049%	45
hashing	100000	111646	0.001%	12728	0.167%	16050	-0.004%	13387	-0.287%	1361	-0.003%	1248	0.223%	50
$c = 0.3$	500000	558235	-0.000%	63900	-0.244%	80245	0.001%	67180	-0.653%	6804	-0.003%	6244	0.136%	62
	5000	5663	0.004%	608	0.059%	719	-0.010%	589	-0.103%	68	-0.064%	63	0.260%	33
asymm.	10000	11327	0.000%	1216	-0.029%	1438	0.011%	1180	-0.229%	136	-0.055%	126	0.060%	38
cuckoo	50000	56637	0.001%	6063	0.268%	7190	-0.002%	5882	0.114%	682	-0.019%	632	-0.127%	51
hashing	100000	113274	0.000%	12198	-0.320%	14379	-0.000%	11769	0.071%	1363	0.007%	1265	-0.122%	57
$c = 0.4$	500000	566371	0.000%	60105	1.139%	71895	0.000%	58312	0.973%	6814	0.003%	6337	-0.331%	72
	5000	5488	0.002%	665	0.627%	904	-0.013%	759	0.451%	67	-0.008%	61	0.424%	26
simplified	10000	10976	0.002%	1334	0.296%	1807	-0.012%	1510	0.895%	134	-0.003%	123	-0.649%	30
cuckoo	50000	54881	0.000%	6655	0.529%	9036	-0.001%	7631	-0.130%	672	-0.013%	612	-0.037%	39
hashing	100000	109762	0.000%	13320	0.446%	18072	-0.002%	15120	0.799%	1344	0.008%	1229	-0.386%	43
	500000	548811	0.000%	66987	-0.128%	90360	-0.001%	76377	-0.224%	6721	-0.001%	6134	-0.248%	53

Table 6.1: Number of trees of sizes one, two, and five for $\boldsymbol{\varepsilon = 0.4}$. The table provides sample mean and sample variance of the number of trees obtained over a sample of size 10^5. Additionally we give the relative error with respect to the asymptotic approximations of Theorem 6.1. Finally, the table depicts the numerically obtained average size of the maximum tree component.

		isolated nodes				trees with 2 nodes				trees with 5 nodes				average
$\varepsilon = 0.2$	m	sample mean	rel. error	sample var.	rel. error	sample mean	rel. error	sample var.	rel. error	sample mean	rel. error	sample var.	rel. error	max. tree
	5000	4493	0.007%	856	0.351%	808	-0.023%	695	0.256%	78	-0.061%	75	-0.348%	67
standard	10000	8986	0.003%	1719	-0.028%	1615	-0.008%	1395	-0.101%	156	-0.017%	150	0.252%	81
cuckoo	50000	44933	0.001%	8557	0.398%	8076	-0.003%	6933	0.494%	781	-0.002%	755	-0.582%	118
hashing	100000	89866	-0.001%	17018	0.960%	16152	0.001%	13873	0.442%	1563	-0.008%	1493	0.568%	135
	500000	449328	0.000%	85933	-0.021%	80759	-0.000%	69648	0.033%	7815	-0.002%	7507	0.029%	178
	5000	4507	0.007%	850	0.376%	795	-0.004%	684	-0.096%	78	-0.108%	75	-0.000%	69
asymm.	10000	9015	0.006%	1710	-0.246%	1590	-0.018%	1361	0.431%	156	-0.012%	150	-0.249%	83
cuckoo	50000	45077	0.001%	8576	-0.557%	7946	-0.001%	6847	-0.143%	778	0.016%	755	-0.777%	121
hashing	100000	90155	0.000%	17019	0.217%	15893	0.001%	13537	1.000%	1557	-0.006%	1497	0.013%	139
$c = 0.1$	500000	450773	0.000%	85727	-0.522%	79465	-0.002%	67907	0.674%	7786	-0.001%	7457	0.408%	184
	5000	4552	0.011%	834	-0.017%	756	-0.030%	646	-0.173%	77	-0.073%	74	0.158%	74
asymm.	10000	9104	0.001%	1676	-0.570%	1511	0.002%	1289	-0.011%	154	-0.046%	148	0.019%	90
cuckoo	50000	45520	0.001%	8309	0.296%	7555	-0.001%	6476	-0.473%	769	0.008%	737	0.541%	133
hashing	100000	91041	-0.000%	16653	0.088%	15110	0.001%	12831	0.457%	1538	-0.012%	1478	0.340%	154
$c = 0.2$	500000	455201	0.000%	83163	0.209%	75551	-0.001%	64425	0.042%	7690	-0.000%	7473	-0.790%	205
	5000	4628	0.011%	792	0.956%	690	-0.034%	579	0.120%	75	-0.089%	72	0.202%	85
asymm.	10000	9257	0.007%	1588	0.674%	1379	-0.023%	1156	0.208%	150	-0.104%	146	-0.601%	106
cuckoo	50000	46290	-0.000%	7918	0.967%	6894	0.004%	5756	0.646%	750	0.024%	727	-0.136%	160
hashing	100000	92580	-0.000%	15997	-0.042%	13788	0.002%	11600	-0.124%	1500	-0.004%	1438	1.010%	186
$c = 0.3$	500000	462899	-0.000%	80093	-0.177%	68939	0.000%	57671	0.448%	7500	-0.004%	7245	0.219%	252
	5000	4743	0.007%	750	-0.071%	595	-0.010%	484	1.066%	72	-0.050%	70	-0.050%	107
asymm.	10000	9487	0.004%	1499	0.003%	1191	-0.011%	973	0.521%	143	-0.016%	140	-0.255%	137
cuckoo	50000	47438	0.001%	7446	0.645%	5955	-0.008%	4842	1.009%	717	-0.018%	695	0.407%	219
hashing	100000	94876	0.001%	14819	1.138%	11909	-0.004%	9711	0.725%	1434	-0.004%	1378	1.319%	259
$c = 0.4$	500000	474382	-0.000%	74816	0.174%	59543	0.001%	49020	-0.222%	7169	0.003%	6962	0.262%	364
	5000	4493	0.006%	855	0.528%	808	-0.020%	697	0.014%	78	-0.059%	75	0.660%	67
simplified	10000	8986	0.001%	1716	0.159%	1615	-0.003%	1394	-0.050%	156	-0.011%	150	0.145%	81
cuckoo	50000	44932	0.001%	8594	-0.030%	8076	-0.003%	6914	0.759%	782	-0.019%	752	-0.114%	117
hashing	100000	89864	0.002%	17149	0.199%	16152	-0.003%	13926	0.060%	1563	-0.020%	1492	0.654%	135
	500000	449327	0.000%	86128	-0.247%	80760	-0.002%	69241	0.617%	7815	-0.001%	7534	-0.324%	178

Table 6.2: Number of trees of sizes one, two, and five for $\boldsymbol{\varepsilon = 0.2}$. The table provides sample mean and sample variance of the number of trees obtained over a sample of size 10^5. Additionally we give the relative error with respect to the asymptotic approximations of Theorem 6.1. Finally, the table depicts the numerically obtained average size of the maximum tree component.

		isolated nodes				trees with 2 nodes				trees with 5 nodes				average
$\varepsilon = 0.1$	m	sample mean	rel. error	sample var.	rel. error	sample mean	rel. error	sample var.	rel. error	sample mean	rel. error	sample var.	rel. error	max. tree
	5000	4065	0.009%	916	0.925%	744	-0.015%	642	0.205%	76	-0.113%	75	-0.461%	132
standard	10000	8131	0.008%	1850	0.021%	1488	-0.013%	1290	-0.272%	152	-0.074%	149	-0.690%	173
cuckoo	50000	40657	0.000%	9279	-0.313%	7439	-0.004%	6406	0.372%	759	0.009%	741	0.073%	291
hashing	100000	81314	0.000%	18503	-0.015%	14877	-0.001%	12875	-0.114%	1519	-0.015%	1491	-0.505%	350
	500000	406568	0.000%	92629	-0.138%	74385	-0.001%	64291	0.018%	7592	0.004%	7466	-0.678%	507
	5000	4082	0.009%	913	0.455%	731	-0.022%	632	-0.338%	75	-0.066%	74	-0.085%	137
asymm.	10000	8164	0.000%	1841	-0.350%	1461	0.015%	1262	-0.207%	151	-0.007%	148	-0.667%	180
cuckoo	50000	40822	0.002%	9154	0.208%	7305	-0.006%	6279	0.260%	755	-0.017%	738	-0.062%	306
hashing	100000	81644	0.001%	18393	-0.252%	14609	-0.002%	12523	0.534%	1509	0.002%	1476	-0.074%	371
$c = 0.1$	500000	408224	-0.000%	91753	-0.019%	73044	0.001%	62936	0.024%	7545	-0.002%	7386	-0.160%	541
	5000	4132	0.011%	895	-0.124%	690	-0.011%	589	-0.014%	74	-0.023%	72	0.466%	152
asymm.	10000	8265	0.006%	1781	0.424%	1380	-0.015%	1181	-0.178%	148	-0.033%	145	-0.053%	205
cuckoo	50000	41328	0.001%	8935	0.064%	6901	-0.004%	5970	-1.324%	739	-0.002%	723	0.158%	365
hashing	100000	82656	0.001%	17714	0.936%	13802	-0.000%	11647	1.171%	1479	-0.007%	1449	-0.124%	450
$c = 0.2$	500000	413280	0.000%	88870	0.599%	69009	0.000%	58446	0.809%	7393	-0.003%	7142	1.323%	675
	5000	4219	0.020%	853	0.135%	623	-0.031%	523	0.012%	71	-0.060%	70	0.024%	185
asymm.	10000	8440	0.007%	1708	0.034%	1245	-0.006%	1049	-0.253%	142	-0.046%	140	-0.010%	260
cuckoo	50000	42202	0.002%	8498	0.528%	6226	-0.003%	5206	0.492%	711	-0.016%	699	-0.204%	519
hashing	100000	84406	0.000%	17104	-0.108%	12451	-0.003%	10441	0.204%	1422	-0.010%	1393	0.155%	663
$c = 0.3$	500000	422029	0.001%	85273	0.182%	62255	-0.002%	52370	-0.111%	7108	0.001%	6899	1.124%	1081
	5000	4348	0.037%	783	1.793%	528	-0.021%	435	-0.223%	67	-0.173%	66	-0.322%	248
asymm.	10000	8698	0.020%	1580	0.889%	1056	-0.011%	871	-0.378%	133	-0.117%	131	0.235%	378
cuckoo	50000	43498	0.003%	7936	0.459%	5280	-0.003%	4368	-0.701%	665	-0.024%	651	0.663%	968
hashing	100000	86997	0.001%	15969	-0.144%	10558	0.003%	8718	-0.491%	1331	-0.008%	1310	0.032%	1420
$c = 0.4$	500000	434989	0.000%	79799	-0.088%	52795	-0.002%	43558	-0.419%	6653	-0.002%	6534	0.280%	3165
	5000	4065	0.009%	918	0.712%	744	-0.013%	641	0.335%	76	-0.118%	74	0.146%	132
simplified	10000	8131	-0.000%	1837	0.729%	1488	0.001%	1281	0.362%	152	-0.024%	149	-0.488%	172
cuckoo	50000	40657	0.000%	9333	-0.891%	7439	-0.001%	6482	-0.803%	759	-0.011%	739	0.324%	290
hashing	100000	81314	0.000%	18580	-0.429%	14877	0.002%	12910	-0.386%	1519	-0.016%	1482	0.090%	350
	500000	406571	-0.000%	93055	-0.598%	74384	0.000%	64603	-0.468%	7592	0.001%	7415	0.022%	507

Table 6.3: Number of trees of sizes one, two, and five for $\boldsymbol{\varepsilon = 0.1}$. The table provides sample mean and sample variance of the number of trees obtained over a sample of size 10^5. Additionally we give the relative error with respect to the asymptotic approximations of Theorem 6.1. Finally, the table depicts the numerically obtained average size of the maximum tree component.

		isolated nodes				trees with 2 nodes				trees with 5 nodes				average
$\varepsilon = 0.06$	m	sample mean	rel. error	sample var.	rel. error	sample mean	rel. error	sample var.	rel. error	sample mean	rel. error	sample var.	rel. error	max. tree
	5000	3906	0.016%	935	1.160%	717	-0.016%	615	0.804%	74	-0.120%	73	-0.113%	182
standard	10000	7812	0.010%	1897	-0.245%	1434	-0.003%	1239	0.089%	148	-0.088%	145	0.292%	254
cuckoo	50000	39062	0.001%	9441	0.208%	7172	-0.002%	6216	-0.251%	740	-0.008%	722	0.582%	492
hashing	100000	78125	0.000%	18920	0.005%	14343	0.001%	12386	0.126%	1479	0.001%	1440	0.805%	626
	500000	390628	-0.000%	94640	-0.039%	71718	-0.001%	61910	0.159%	7396	0.008%	7225	0.449%	999
	5000	3923	0.017%	934	0.455%	704	-0.003%	607	-0.099%	74	-0.119%	71	1.180%	189
asymm.	10000	7847	0.005%	1859	0.919%	1407	0.003%	1207	0.524%	147	-0.049%	145	-0.429%	266
cuckoo	50000	39236	0.001%	9333	0.488%	7037	-0.000%	6022	0.708%	734	-0.023%	717	0.509%	530
hashing	100000	78473	-0.000%	18799	-0.218%	14074	0.001%	12120	0.087%	1469	0.006%	1444	-0.183%	681
$c = 0.1$	500000	392360	0.000%	93465	0.349%	70368	0.001%	60402	0.414%	7344	-0.008%	7210	-0.019%	1107
	5000	3976	0.026%	904	1.047%	663	-0.042%	568	-0.252%	72	-0.103%	70	0.832%	212
asymm.	10000	7952	0.011%	1816	0.580%	1326	-0.010%	1131	0.170%	144	-0.077%	141	0.060%	308
cuckoo	50000	39765	0.001%	9198	-0.712%	6631	0.000%	5723	-1.071%	717	0.000%	704	0.190%	671
hashing	100000	79530	0.001%	18310	-0.235%	13263	-0.003%	11350	-0.228%	1435	-0.014%	1414	-0.273%	895
$c = 0.2$	500000	397656	-0.000%	91033	0.329%	66313	0.001%	56521	0.178%	7173	-0.007%	7005	0.651%	1575
	5000	4066	0.039%	857	1.640%	596	-0.015%	500	0.126%	69	-0.147%	68	-0.091%	259
asymm.	10000	8134	0.021%	1730	0.741%	1191	0.007%	1011	-1.048%	137	-0.063%	134	0.658%	398
cuckoo	50000	40678	0.006%	8757	-0.466%	5955	-0.002%	5028	-0.478%	686	-0.020%	670	0.794%	1038
hashing	100000	81359	0.002%	17336	0.564%	11909	0.003%	9992	0.162%	1372	-0.025%	1346	0.426%	1549
$c = 0.3$	500000	406801	0.000%	87321	-0.174%	59550	-0.001%	50013	0.056%	6860	-0.002%	6735	0.328%	3606
	5000	3906	0.014%	950	-0.379%	717	-0.006%	623	-0.444%	74	-0.066%	73	-0.632%	181
simplified	10000	7812	0.007%	1888	0.202%	1434	-0.009%	1245	-0.419%	148	-0.065%	145	0.140%	254
cuckoo	50000	39063	-0.000%	9375	0.907%	7172	0.003%	6188	0.203%	740	0.012%	727	-0.212%	492
hashing	100000	78125	0.001%	18784	0.724%	14344	-0.001%	12424	-0.182%	1479	-0.006%	1453	-0.097%	627
	500000	390628	-0.000%	93826	0.821%	71718	-0.001%	61855	0.248%	7397	0.002%	7202	0.772%	1002

Table 6.4: Number of trees of sizes one, two, and five for $\boldsymbol{\varepsilon = 0.06}$. The table provides sample mean and sample variance of the number of trees obtained over a sample of size 10^5. Additionally we give the relative error with respect to the asymptotic approximations of Theorem 6.1. Finally, the table depicts the numerically obtained average size of the maximum tree component.

		isolated nodes				trees with 2 nodes				trees with 5 nodes				average
$\varepsilon = 0.04$	m	sample mean	rel. error	sample var.	rel. error	sample mean	rel. error	sample var.	rel. error	sample mean	rel. error	sample var.	rel. error	max. tree
	5000	3828	0.028%	955	0.030%	704	-0.025%	608	0.125%	73	-0.099%	72	-0.011%	216
standard	10000	7657	0.007%	1898	0.665%	1407	0.008%	1226	-0.768%	146	-0.060%	142	0.844%	313
cuckoo	50000	38289	0.002%	9591	-0.383%	7037	-0.001%	6120	-0.574%	728	-0.012%	723	-1.003%	681
hashing	100000	76578	0.001%	19038	0.367%	14074	0.000%	12138	0.262%	1456	-0.003%	1430	0.090%	907
	500000	382892	0.000%	95216	0.343%	70371	-0.000%	60441	0.667%	7281	-0.001%	7214	-0.793%	1602
	5000	3846	0.024%	943	0.432%	690	-0.011%	594	0.139%	72	-0.105%	72	-0.807%	223
asymm.	10000	7692	0.015%	1890	0.222%	1381	-0.014%	1184	0.514%	145	-0.044%	143	-0.386%	327
cuckoo	50000	38466	0.001%	9491	-0.214%	6902	-0.003%	5939	0.164%	723	-0.022%	712	-0.254%	737
hashing	100000	76932	0.002%	18974	-0.166%	13804	-0.003%	11953	-0.457%	1445	-0.009%	1414	0.507%	999
$c = 0.1$	500000	384665	-0.000%	95304	-0.626%	69019	0.000%	59716	-0.376%	7224	0.003%	7093	0.166%	1831
	5000	3899	0.037%	912	1.117%	650	-0.049%	552	0.407%	71	-0.201%	70	-0.600%	250
asymm.	10000	7800	0.023%	1819	1.353%	1299	-0.018%	1102	0.700%	141	-0.127%	139	-0.031%	381
cuckoo	50000	39006	0.004%	9219	0.004%	6496	0.001%	5548	-0.016%	705	-0.020%	694	-0.070%	951
hashing	100000	78013	0.002%	18393	0.248%	12993	-0.004%	11077	0.144%	1409	0.011%	1392	-0.320%	1375
$c = 0.2$	500000	390075	0.000%	91638	0.604%	64959	0.003%	54912	0.999%	7047	-0.000%	7004	-0.956%	2955
	5000	3828	0.021%	950	0.553%	704	-0.020%	612	-0.600%	73	-0.152%	71	0.460%	215
simplified	10000	7657	0.010%	1897	0.707%	1407	0.003%	1215	0.198%	146	-0.081%	143	0.279%	311
cuckoo	50000	38289	0.001%	9583	-0.304%	7037	-0.000%	6094	-0.152%	728	0.001%	717	-0.176%	677
hashing	100000	76579	-0.000%	19141	-0.168%	14074	0.005%	12181	-0.094%	1456	0.014%	1436	-0.327%	904
	500000	382894	-0.000%	96114	-0.597%	70371	0.001%	60605	0.397%	7281	0.004%	7172	-0.204%	1600

Table 6.5: Number of trees of sizes one, two, and five for $\boldsymbol{\varepsilon = 0.04}$. The table provides sample mean and sample variance of the number of trees obtained over a sample of size 10^5. Additionally we give the relative error with respect to the asymptotic approximations of Theorem 6.1. Finally, the table depicts the numerically obtained average size of the maximum tree component.

		nodes in cyclic components				number of cycles				average
		sample mean	rel. error	sample var.	rel. error	sample mean	rel. error	sample var.	rel. error	max. cyclic comp.
$\varepsilon = 0.4$	standard cuckoo hashing	1.41	-0.032%	16.5	-1.672%	0.223	0.109%	0.224	-0.204%	1.32
	asymmetric c.h., $c = 0.1$	1.44	0.064%	17.0	-0.376%	0.226	0.160%	0.226	-0.093%	1.35
	asymmetric c.h., $c = 0.2$	1.56	0.301%	19.7	-1.385%	0.234	0.537%	0.234	0.312%	1.46
	asymmetric c.h., $c = 0.3$	1.80	-0.332%	24.4	1.211%	0.254	-1.075%	0.253	-0.586%	1.68
	asymmetric c.h., $c = 0.4$	2.23	0.802%	35.7	1.858%	0.279	0.131%	0.280	0.033%	2.08
	simplified cuckoo hashing	1.85	1.316%	18.3	2.641%	0.456	0.438%	0.455	0.656%	1.67
$\varepsilon = 0.2$	standard cuckoo hashing	8.86	0.326%	420.9	0.898%	0.509	0.381%	0.507	0.732%	7.99
	asymmetric c.h., $c = 0.1$	9.32	0.329%	475.3	-1.828%	0.517	0.605%	0.522	-0.331%	8.39
	asymmetric c.h., $c = 0.2$	11.01	-0.050%	619.7	1.941%	0.549	0.043%	0.548	0.311%	9.89
	asymmetric c.h., $c = 0.3$	15.00	0.052%	1132.7	0.135%	0.608	-0.041%	0.610	-0.465%	13.37
	asymmetric c.h., $c = 0.4$	26.31	-0.283%	3324.6	-1.320%	0.717	0.053%	0.716	0.152%	23.30
	simplified cuckoo hashing	9.95	0.507%	445.3	1.048%	0.801	0.496%	0.802	0.377%	8.75
$\varepsilon = 0.1$	standard cuckoo hashing	42.36	0.649%	8191.8	1.593%	0.830	0.079%	0.827	0.415%	37.36
	asymmetric c.h., $c = 0.1$	46.75	1.056%	9775.5	3.762%	0.857	-0.511%	0.856	-0.453%	41.02
	asymmetric c.h., $c = 0.2$	65.66	1.940%	19211.2	3.892%	0.924	0.394%	0.917	1.234%	57.52
	asymmetric c.h., $c = 0.3$	143.88	1.865%	86487.5	6.546%	1.106	-0.144%	1.100	0.409%	125.48
	asymmetric c.h., $c = 0.4$	1059.13	32.367%	3394840.0	66.167%	1.610	3.343%	1.575	5.465%	913.83
	simplified cuckoo hashing	44.81	0.422%	8375.5	2.041%	1.149	0.241%	1.150	0.137%	38.89
$\varepsilon = 0.06$	standard cuckoo hashing	123.54	2.350%	64162.5	7.368%	1.069	0.625%	1.066	0.876%	107.83
	asymmetric c.h., $c = 0.1$	146.52	2.734%	90427.4	7.306%	1.110	0.426%	1.113	0.204%	127.33
	asymmetric c.h., $c = 0.2$	272.48	5.264%	292639.0	16.073%	1.257	0.680%	1.258	0.561%	236.17
	asymmetric c.h., $c = 0.3$	1353.49	42.292%	5116340.0	77.172%	1.689	4.558%	1.650	6.758%	1167.11
	simplified cuckoo hashing	127.24	2.543%	65382.4	7.068%	1.406	0.046%	1.406	0.041%	109.82
$\varepsilon = 0.04$	standard cuckoo hashing	279.76	4.803%	307422.0	15.485%	1.268	0.367%	1.270	0.269%	242.42
	asymmetric c.h., $c = 0.1$	357.38	6.960%	490130.0	20.621%	1.330	0.456%	1.333	0.235%	309.49
	asymmetric c.h., $c = 0.2$	897.51	25.207%	2590900.0	56.156%	1.568	2.576%	1.551	3.660%	773.91
	simplified cuckoo hashing	284.33	5.223%	305917.0	16.757%	1.612	-0.140%	1.603	0.380%	244.60

Table 6.6: The table shows sample mean and sample variance of the *number of nodes contained in cyclic components* and the *number of cycles*. We provide numerically obtained results using a sample of size 10^5 and give the relative error with respect to the asymptotic approximations of Theorem 6.1. Furthermore, we provide the numerically obtained average size of the largest cyclic component. All depicted values are obtained using a fixed table size determined by the parameter $m = 5 \cdot 10^5$, but our numerical data obtained using different table sizes are almost identical.

Chapter 7

Construction Cost

7.1 Introduction

So far, we analysed the failure probability of the various cuckoo hash algorithms and obtained some information on the structure of the related cuckoo graph. In this chapter, we investigate the average case behaviour of insertion operations. The cost of a single insertion is thereby measured by the number of moved keys during this procedure, hence it equals one plus the number of kick-out operations. Unfortunately, the exact behaviour is very complex to describe and no exact result is known so far. Hence, we cannot give an exact result, but we are looking for a suitable upper bound.

7.2 Simplified cuckoo hashing

As usual, we start with the analysis of the simplified algorithm.

Theorem 7.1. *Suppose that $\varepsilon \in (0,1)$ is fixed. Then, an upper bound for the expected number of steps to construct a simplified cuckoo hash data structure possessing a table of size $2m$ with $n = \lfloor (1-\varepsilon)m \rfloor$ keys is*

$$\min\left(C, \frac{-\log\varepsilon}{1-\varepsilon}\right) n + \mathcal{O}(1), \tag{7.1}$$

where the constant implied by $\mathcal{O}(1)$ depends on ε. By performing numerical calculations it turns out that this bound holds for $C = 4$.

Proof of Theorem 7.1

Denote the failure probability of a simplified cuckoo hashing attempt by p. Clearly, the expected number of attempts to construct the data structure is hence given by $1/(1-p)$. We have already shown that the equation $p = \mathcal{O}(1/m)$ holds. This implies that the expected number of rehashes to build the hash table, what we denote by N, is in $\mathcal{O}(1/m)$. Furthermore, the additional complexity of a failed attempt is $\mathcal{O}(n)$, because we detect an endless loop in the insertion procedure after at most $2n$ steps. Therefore, it is only left to show the claimed bound for the situation where cuckoo hashing succeeds, *i.e.* the cuckoo graph contains only trees and cyclic components. Using this result, we conclude that C_i, the number of steps required during the i-th unsuccessful construction, is in $\mathcal{O}(m)$ hence the equation

$$\mathbb{E}\sum_{i=1}^{N} C_i = \mathbb{E}N\,\mathbb{E}C_1 = \mathcal{O}(1) \tag{7.2}$$

holds, *cf.* Devroye and Morin [2003].

Consider the graph just before the insertion of the l-th edge (resp. key) and denote the node addressed by the first hash function by x_l and the second by y_l. Recall that a new key is always placed in x_l by the standard insertion procedure. The number of steps needed to perform this insertion is fully determined by the component containing x_l, and not affected by the component containing y_l, unless x_l belongs to a cyclic component or both nodes belong to the same component (including the situation where $x_l = y_l$ holds). In the latter case, the new edge creates a cycle. Hence we end up with a component containing a cycle anyway. But this is a very rare event, because as we know from Theorem 6.1, that the expected number of nodes contained in cyclic components is finite. More precisely, Lemma 7.1 shows that the expected cost caused by cyclic components is constant.

Lemma 7.1. *The expected number of all steps performed while inserting elements in cyclic components is constant.*

Proof. Let η denote the random variable that determines the number of nodes contained in cyclic components. Assume that η equals k. Then, the insertion of each of the k corresponding keys takes at most $2k$ steps, because during an insertion, no node is visited more than twice (*cf.* Figure 1.6) and each cyclic component holds at most k keys. The total number of expected steps is therefore bounded by

$$\sum_k 2k^2\mathbb{P}(\eta = k) = 2\mathbb{V}\eta + (\mathbb{E}\eta)^2 = \mathcal{O}(1), \tag{7.3}$$

what is constant for all fixed ε because of the results from Theorem 6.1. □

The cuckoo graph contains $2m-l+1$ trees before the insertion of the l-th node. Denote the number of steps needed for the insertion in tree T by $\nu(T)$ and the number of its nodes by $m(T)$. The probability, that a tree possessing $m(T)$ nodes is chosen equals $m(T)/(2m)$. Using $\tilde{t}(x)$, the generating function of unrooted trees defined in (4.5), we obtain the generating function $H(x)/(2m)$ that counts the maximum insertion cost of a tree component, weighted by the probability that this component is chosen as follows:

$$\tilde{t}(x) := \sum_T \frac{x^{m(T)}}{m(T)!} = \sum_k \tilde{t}_k \frac{x^k}{k!}, \tag{7.4}$$

$$H(x) := \sum_T m(T)\nu(T)\frac{x^{m(T)}}{m(T)!}. \tag{7.5}$$

Next, we extend this to a set consisting of k such trees:

$$\frac{1}{k!}\sum_{(T_1,\ldots,T_k)}\big(m(T_1)\nu(T_1)+\cdots+m(T_k)\nu(T_k)\big)$$
$$\times\binom{m(T_1)+\cdots+m(T_k)}{m(T_1),\ldots,m(T_k)}\frac{x^{m(T_1)+\cdots+m(T_k)}}{\big(m(T_1)+\cdots+m(T_k)\big)!}$$
$$=\frac{1}{(k-1)!}H(x)\tilde{t}(x)^{k-1}. \quad (7.6)$$

Neglecting costs caused by cyclic components, we obtain a result indicating the average complexity

$$C(l):=\frac{1}{2m\ \#G^{\circ}_{2m,l-1}}\frac{2^{l-1}(l-1)!(2m)!}{(2m-l)!}[x^{2m}]\frac{H(x)\tilde{t}(x)^{2m-l}}{\sqrt{1-t(x)}}$$
$$=\frac{2m-l+1}{2m}\frac{[x^{2m}]H(x)\tilde{t}(x)^{2m-l}/\sqrt{1-t(x)}}{[x^{2m}]\tilde{t}(x)^{2m-l+1}/\sqrt{1-t(x)}}, \quad (7.7)$$

for the insertion of the $l-$th key, *cf.* (4.9). We proceed similar to former calculations, using the saddle point method to extract the coefficient. A slight difference is the new occurring function $H(x)$, but it behaves as a constant factor, so we only need to know $H(x_0)$, what we consider as follows.

We give a first estimate using the tree size $m(T)$ as upper bound of the parameter $\nu(T)$, and obtain for real x the inequality

$$H(x)\le\sum_T m(T)^2\frac{x^{m(T)}}{m(T)!}=x\frac{\partial}{\partial x}\left(x\frac{\partial}{\partial x}\tilde{t}(x)\right)=x\frac{\partial}{\partial x}t(x)=\frac{t(x)}{1-t(x)}. \quad (7.8)$$

Recall that $t(x_0)=\frac{l-1}{m}$ and $\tilde{t}(x_0)=\frac{l-1}{2m}\left(2-\frac{l-1}{m}\right)$ hold. Altogether, this gives us

$$C(l)=\frac{(2m-l+1)H(x_0)}{2m\,\tilde{t}(x_0)}\left(1+\mathcal{O}\left(\frac{1}{2m-l}\right)\right)\le\frac{m}{m-l}\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right), \quad (7.9)$$

and further summation over all l leads to

$$\frac{1}{n}\sum_{l=1}^{n}C(l)\le\frac{1}{n}\sum_{l=1}^{n}\frac{1}{1-\frac{l}{m}}\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right)\to\frac{1}{1-\varepsilon}\int_{\varepsilon}^{1}\frac{da}{a}=\frac{\log\frac{1}{\varepsilon}}{1-\varepsilon}, \quad (7.10)$$

as m goes to infinity. Together with Lemma 7.1, this completes the proof of the first bound of Theorem 7.2.

Next, we try to obtain a better bound using a more suitable estimate for $\nu(T)$. Note that the selection of the node x_l in a tree component, transforms this component into a rooted tree. The insertion procedure starts at the root and the number of required steps is bounded by the height of this tree. We introduce the denotations

- t_n for the number of rooted trees with n nodes,
- $t_n^{[k]}$ for the number of rooted trees with n nodes and height less or equal k,

- and h_n for the sum of the heights of all rooted trees with n nodes.

Moreover, we introduce the corresponding generating functions:

$$t(x) = \sum_{n\geq 0} t_n x^n, \qquad t^{[k]}(x) = \sum_{n\geq 0} t_n^{[k]} x^n, \qquad h(x) = \sum_{n\geq 0} h_n x^n. \tag{7.11}$$

Due to Flajolet and Odlyzko [1982], we know that

$$t(x) - t^{[k]}(x) \sim 2\frac{\delta(x)\,(1-\delta(x))^n}{1-(1-\delta(x))^n}, \tag{7.12}$$

holds where $\delta(x) = \sqrt{2(1-ex)} + \mathcal{O}(1-ex)$ and further,

$$h(x) = \sum_{k\geq 0} \left(t(x) - t^{[k]}(x)\right) \sim -2\log\delta(x), \tag{7.13}$$

in a Δ-domain around its singularity e^{-1}, *cf.* Flajolet and Sedgewick [2008].

Now, we use the asymptotic approximation of $h(x)$ as upper bound of $H(x)$ and obtain similar to (7.9) the upper bound

$$C(l) \leq -\frac{m}{l-1}\log\left(2 - 2\frac{l-1}{m}e^{1-\frac{l-1}{m}}\right)\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right), \tag{7.14}$$

for the construction time. This is of course valid only near the singularity, that is for $1-(l-1)/m$ close to zero. Nevertheless, this result is suitable to prove the second bound stated in Theorem 7.2. This is because of the fact that the integral

$$\int_0^{1/2} \frac{-\log 2\,(1-(1-a)e^a)}{1-a}\,da \tag{7.15}$$

is obviously bounded for $\varepsilon \to 0$, in contrary to the corresponding integral of (7.10). See Figure 7.1 for a plot of the two bounds. It is easy to see, that the second bound is not valid if ε is not close to 0. Finally, we compute a numerical value for the constant C by combining this two bounds.

$$C = \frac{1}{1-\varepsilon}\left(\int_x^1 \frac{da}{a} + \int_0^x \frac{-\log 2\,(1-(1-a)e^a)}{1-a}\,da\right). \tag{7.16}$$

To be on the safe side, we may for instance set x equal to 0.025 and obtain a bound approximately equal to 4.

7.3 Standard cuckoo hashing

It is straightforward to generalise the ideas that lead us to the previous result. Surprisingly, it turns out that the same bound holds for the classical cuckoo hash algorithm too.

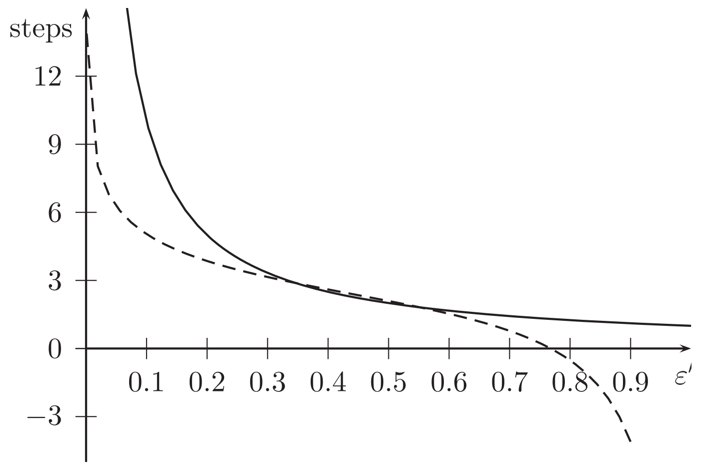


Figure 7.1: Bounds for the expected number of steps per insertion, depending on the momentary value of ε'. The continuous line corresponds to the bound based on the size of a tree component. Further, the dashed line indicates the bound obtained using the diameter as estimator, that is accurate for ε' close to 0 only.

Theorem 7.2. *Suppose that $\varepsilon \in (0,1)$ is fixed. Then, an upper bound for the expected number of steps to construct a cuckoo hash data structure possessing two tables of size m with $n = \lfloor (1-\varepsilon)m \rfloor$ keys is*

$$\min\left(C, \frac{-\log\varepsilon}{1-\varepsilon}\right) n + \mathcal{O}(1), \tag{7.17}$$

where the constant implied by $\mathcal{O}(1)$ depends on ε. By performing numerical calculations it turns out that this bound holds for $C = 4$.

Proof of Theorem 7.2

Note that all initial considerations stated in the proof of the simplified version can easily be adopted. For instance, the assertion of Lemma 7.1 holds, however the proof is now based on Theorem 6.2. The further proof continues in a similar way, though we have to use bivariate generating functions once more.

Consider the bipartite cuckoo graph before the insertion of the l-st node. At this moment, it contains exactly $2m - l + 1$ tree components. Denote the number of steps needed for the insertion in tree T by $\nu(T)$ and its number of nodes of first resp. second type by $m_1(T)$ resp. $m_2(T)$. The probability, that a tree possessing m_1 nodes of first kind is chosen equals m_1/m. Using the bivariate generating function $\tilde{t}(x,y)$ of unrooted bipartite trees, we obtain the generating function $H(x,y)/m$ counting the maximum insertion cost of a tree component times its selection probability as follows:

$$\tilde{t}(x,y) := \sum_T \frac{x^{m_1(T)} y^{m_2(T)}}{m_1(T)! m_2(T)!} = \sum_{m_1,m_2} \tilde{t}_{m_1,m_2} \frac{x^{m_1} y^{m_2}}{m_1! m_2!}, \tag{7.18}$$

$$H(x,y) := \sum_T m_1(T)\nu(T) \frac{x^{m_1(T)} y^{m_2(T)}}{m_1(T)! m_2(T)!}. \tag{7.19}$$

Again, we extend this to a set consisting of k such trees:

$$\frac{1}{k!}\sum_{(T_1,\dots,T_k)}\big(m_1(T_1)\nu(T_1)+\dots+m_1(T_k)\nu(T_k)\big)\binom{m_1(T_1)+\dots+m_1(T_k)}{m_1(T_1),\dots,m_1(T_k)}$$
$$\times\binom{m_2(T_1)+\dots+m_2(T_k)}{m_2(T_1),\dots,m_2(T_k)}\frac{x^{m_1(T_1)\dots m_1(T_k)}}{\big(m_1(T_1)+\dots+m_1(T_k)\big)!}\frac{y^{m_2(T_1)\dots m_2(T_k)}}{\big(m_2(T_1)+\dots+m_2(T_k)\big)!}$$
$$=\frac{1}{(k-1)!}H(x,y)\tilde{t}(x,y)^{k-1}. \quad (7.20)$$

By neglecting costs caused by cyclic components, we get a result indicating the average complexity

$$C(l):=\frac{1}{m\;\#G^\circ_{m,m,l-1}}\frac{(m!)^2(l-1)!}{(2m-l)!}[x^my^m]\frac{H(x,y)\tilde{t}(x,y)^{2m-l}}{\sqrt{1-t_1(x,y)t_2(x,y)}}$$
$$=\frac{2m-l+1}{m}\frac{[x^my^m]H(x,y)\tilde{t}(x,y)^{2m-l}/\sqrt{1-t_1(x,y)t_2(x,y)}}{[x^my^m]\tilde{t}(x,y)^{2m-l+1}/\sqrt{1-t_1(x,y)t_2(x,y)}} \quad (7.21)$$

for the insertion of the $l-$th key. As usual we proceed, using the saddle point method to extract the coefficient. The function $H(x,y)$ behaves as a constant factor, so we need to calculate respectively estimate $H(x_0,x_0)$ only, what can be done as follows.

The first estimate is again based on using the size of the tree component, that is now equal to $m_1(T)+m_2(T)$, as upper bound of the parameter $\nu(T)$. Hence we infer for real valued x and y the inequality

$$H(x,y)\le\sum_T m_1(T)\left(m_1(T)+m_2(T)\right)\frac{x^{m_1(T)}y^{m_2(T)}}{m_1(T)!m_2(T)!}. \quad (7.22)$$

Further, note that the relation

$$\sum_T m_1(T)\frac{x^{m_1(T)}y^{m_2(T)}}{m_1(T)!m_2(T)!}=x\frac{\partial}{\partial x}\tilde{t}(x,y)=t_1(x,y) \quad (7.23)$$

holds. Recall that $t_1(x,x)$ equals $t(x)$, so we establish

$$H(x,x)\le x\frac{\partial}{\partial x}t(x)=\frac{t(x)}{1-t(x)}=\frac{t_1(x,x)}{1-t_1(x,x)}. \quad (7.24)$$

From Chapter 4, we know that $t(x_0,x_0)=\frac{l-1}{m}$ and $\tilde{t}(x_0,x_0)=\frac{l-1}{m}\left(2-\frac{l-1}{m}\right)$ hold. Altogether, this provides us the bound

$$C(l)=\frac{(2m-l+1)H(x_0,x_0)}{m\,\tilde{t}(x_0,x_0)}\left(1+\mathcal{O}\left(\frac{1}{2m-l}\right)\right)\le\frac{m}{m-l}\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right), \quad (7.25)$$

and hence we obtain the same first bound as given for the simplified algorithm, see (7.10).

Again, we try to obtain a better result using a more suitable estimate for $\nu(T)$. The selection of the node x_l belonging to a tree component, transforms this component into a rooted bipartite tree. The insertion procedure starts at the root and the number of required steps is bounded by the height of this tree. Further, recall that we are only interested in the special case $x = y$. Because of this, we can once more consider usual (non-bipartite) rooted trees instead.

Hence we use the asymptotic approximation of $h(x)$ as upper bound of $H(x,x)$ and obtain the upper bound

$$C(l) \leq m \frac{-\log 2\left(1 - e^{1-l/m}\frac{l}{m}\right)}{l}\left(1 + \mathcal{O}\left(\frac{1}{m}\right)\right). \tag{7.26}$$

Note that this second bound also equals the bound obtained analysing the simplified algorithm. Thus the remaining details of this proof can be shown as before.

7.4 Asymmetric cuckoo hashing

Finally, we consider the generalised asymmetric data structure. We are still able to provide an estimate based on the component sizes, similar to the previous results. On the other hand, the derivation of the second bound is strongly based on the symmetry. Hence, it is not possible to adopt this result without further knowledge on the height of rooted bipartite trees.

Theorem 7.3. *Suppose that $c \in [0,1)$ and $\varepsilon \in (1 - \sqrt{1-c^2}, 1)$ are fixed. Then, an upper bound for the expected number of steps to construct an asymmetric cuckoo hash data structure possessing two tables of size $m_1 = \lfloor m(1+c) \rfloor$ respectively $m_2 = 2m - m_1$ with $n = \lfloor (1-\varepsilon)m \rfloor$ keys is*

$$\frac{1-c}{2(1-\varepsilon)}\left(\log\frac{1-c^2}{2\varepsilon(1-\varepsilon)-c^2} - \frac{2(1+c)}{\sqrt{1-c^2}}\operatorname{artanh}\frac{\varepsilon-1}{\sqrt{1-c^2}}\right)n + \mathcal{O}(1), \tag{7.27}$$

where the constant implied by $\mathcal{O}(1)$ depend on ε.

Proof. As mentioned above, the proof is related to the proofs of Theorem 7.1 and 7.2. The expected number of steps performed in cyclic components is still in $\mathcal{O}(1)$ due to Theorem 6.3. Further note that (7.22), the generating function $H(x,y)$ counting the maximum insertion cost of a tree component, still satisfies the inequality

$$H(x,y) \leq \sum_T m_1(T)\left(m_1(T) + m_2(T)\right)\frac{x^{m_1(T)}y^{m_2(T)}}{m_1(T)!m_2(T)!}. \tag{7.28}$$

Recall that multiplying the generating function by $m_1(T)$ corresponds to marking a node of first kind. Hence we obtain

$$\begin{aligned}H(x,y) &\leq x\frac{\partial}{\partial x}\left(x\frac{\partial}{\partial x}\tilde{t}(x,y)\right) + y\frac{\partial}{\partial y}\left(x\frac{\partial}{\partial x}\tilde{t}(x,y)\right) = x\frac{\partial}{\partial x}t_1(x,y) + y\frac{\partial}{\partial y}t_1(x,y)\\ &= \frac{t_1(x,y)}{1-t_1(x,y)t_2(x,y)} + \frac{t_1(x,y)t_2(x,y)}{1-t_1(x,y)t_2(x,y)} = \frac{t_1(x,y)(1+t_2(x,y))}{1-t_1(x,y)t_2(x,y)}.\end{aligned} \tag{7.29}$$

From Chapter 4, we know that $t_1(x_0,y_0) = \frac{l-1}{m(1-c)}$, $t_2(x_0,y_0) = \frac{l-1}{m(1+c)}$ and $\tilde{t}(x_0,y_0) = \frac{(l-1)(2m-l+1)}{m^2(1-c^2)}$ hold. Altogether, this gives us the bound

$$\begin{aligned}C(l) &= \frac{(2m-l+1)H(x_0,y_0)}{m(1+c)\,\tilde{t}(x_0,y_0)}\left(1 + \mathcal{O}\left(\frac{1}{2m-l}\right)\right)\\ &\leq \frac{m(1-c)(m(1+c)+l-1)}{m^2(1-c^2)-(l-1)^2}\left(1 + \mathcal{O}\left(\frac{1}{m}\right)\right).\end{aligned} \tag{7.30}$$

Similar to (7.9), we hence obtain the integral

$$\frac{1}{1-\varepsilon}\int_{\varepsilon}^{1}\frac{(1-c)(2+c-a)}{(2-a)a-c^2}\,da \\ = \frac{1-c}{2(1-\varepsilon)}\left(\log\frac{1-c^2}{2\varepsilon(1-\varepsilon)-c^2} - \frac{2(1+c)}{\sqrt{1-c^2}}\operatorname{artanh}\frac{e-1}{\sqrt{1-c^2}}\right), \quad (7.31)$$

that completes the proof. □

7.5 An improved insertion algorithm

We propose using a slightly modified insertion algorithm that is described in Listing 7.1. In contrast to the algorithm described by Pagh and Rodler [2004], we perform an additional test during the insertion of a key x under following circumstances: If the location $h_1(x)$ is already occupied, but $h_2(x)$ is empty, the algorithm places x in the second table. If both possible storage locations of x are already occupied, we proceed as usual and kick out the key stored in $h_1(x)$. This modification is motivated by two observations:

- We should check if the new key is already contained in the table. If we do not perform the check *a priori* and allow duplicates, we have to perform a check after each kick-out step, what degrades the performance. Further we have to inspect both possible storage locations to perform complete deletions. Because of this, it is not recommended to skip this test. Hence we have to inspect $h_2(x)$ anyway, and there are no negative effects on the average search time caused by this modification.
- The probability, that the position $h_2(x)$ is empty is relatively high. For the simplified algorithm possessing a load α, this probability equals $1-\alpha$. Moreover, we expect an even better behaviour for variants based on two tables, because of the unbalanced load, that is the majority of keys is usually stored in the first table. Denote the key stored at position $h_1(x)$ by y and assume that the corresponding component is a tree. Note that the previous considerations do not hold for $h_2(y)$, because we have to consider the probability conditioned on the property that the node belongs to a tree of size greater than one. This tree possesses exactly one free storage location only.

Experiments show, that this modified insertion algorithm reduces the number of required steps by about 5 to 10 percent.

7.6 Comparison and conclusion

Note that the bounds established in Theorems 7.1, 7.2, and 7.3 overestimate the number of required memory accesses per insertion, especially if ε is small. The attained numerical results, that are given in Table 7.1, show that the expected performance of both standard and simplified cuckoo hashing is by far below this upper bound. Blank entries refer to supercritical settings, where our asymptotic approximations are not applicable, *cf.* Table 4.1. See Chapter 9 for details on implementation and setup. We notice that the

Listing 7.1: Optimal insertion for cuckoo hashing.

```
function insert(s)
    if table1[h1(s)] == used
        if table1[h1(s)] == s
            return true
        end if
        if table2[h2(s)] != used
            table2[h2(q)] = s
            return true
        end if
        if table2[h2(s)] == s
            return true
        end if
    end if
    for i from 1 to maxloop do:
        p = table1[h1(s)]
        table1[h1(s)] = s
        if p != used
            % s did not kick out an element
            return true
        end if
        s = table2[h2(p)]
        table2[h2(p)] = p
        if s != used
            % p did not kick out an element
            return true
        end if
    end for
    % rehash required, insertion not possible
    return false
end function
```

performance depends on the size of the data structure, except for sparse tables. More precisely, the construction requires fewer steps on average if the size of the data structure increases.

Further, we notice that the simplified version offers the best average performance for all investigated settings, compared to all other variants of cuckoo hashing. This seems to be related to the higher probability of hitting an empty cell. Similarly, the asymmetric data structure offers better performance as the asymmetry increases, as long as we do not reach the critical load factor. However, the construction takes more steps on average for an asymmetric table close to its maximum fill ratio than for the standard algorithm possessing the same load factor.

Figure 7.2 displays the bound given in Theorem 7.3 for asymmetry factors $c = 0$ (*i.e.* the standard algorithm), $c = 0.2$, $c = 0.3$, and $c = 0.4$. We conclude that this upper bound shows a similar behaviour as the actual performance of insertions described above. That means that increasing asymmetry decreases the corresponding value for small loads, but the effect reverses near the critical fill ratio.

Next, we compare these results to the average number of memory accesses of an insertion into a table using a standard open addressing algorithm. Note that we do not consider the expected number of steps that are required to insert an element into a table holding a given number of keys, as it is often done for this algorithms. In contrast, we construct a whole table holding n keys, count the total number of required steps, and divide it by n. For linear probing and double hashing, these numbers equal the expected number of steps of an successful search. This is a well known parameter. Hence we provide the results of this analysis only, instead of performing experiments using this algorithms. It can be seen by our attained data, that the performance of simplified cuckoo hashing is approximately equal to the expected number of steps using linear probing and thus slightly higher than using double hashing.

It is also of interest to consider algorithms based on open addressing that try to speed up successful searches by more complicated insertion procedures that require additional steps. In particular, Brent's variation of double hashing is a popular algorithm that follows this approach. See Chapter 1 for a brief description and references. Unfortunately, no exact analysis of the behaviour of the insertion procedure of this algorithm is known. Hence, we provide the numerically obtained number of *different* memory cells inspected during an insertion as a measure of the insertion cost. Hereby, we include the additional memory cells, that are inspected during an initial search operation, that checks if the key is already contained in the data structure. However, note that the standard implementation of this insertion procedure does not use additional memory to hold keys that are accessed more than once, *cf.* Gonnet and Baeza-Yates [1991]. Thus, it is required to load the corresponding entries of the table again, but it is very likely that this repeated accesses do not produce a cache-miss and, hence, we do not expect high additional expenses, if the algorithm is running on a modern computer, *cf.* Heileman and Luo [2005]. From the data given in Table 7.1, we observe that the cost of an insertion using Brent's variation resp. simplified cuckoo hashing is approximately equal for large data structures, while the cuckoo hash algorithm takes a bit more steps for small tables.

Furthermore, the expected length of the longest insertion sequence is a practically relevant attribute, because it is a measure of the expected worst case. Table 7.2 shows the average of the maximum number of steps required during the construction of the whole table. For the two table version of cuckoo hashing, we notify that this number

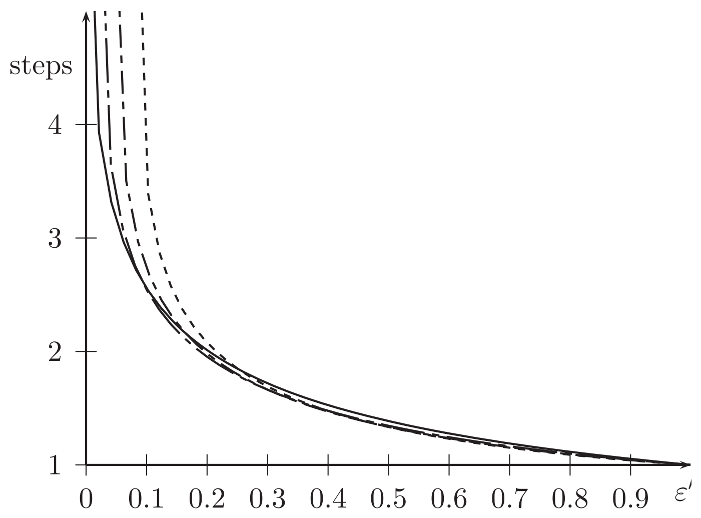


Figure 7.2: Bounds for the expected number of steps per insertion using asymmetric cuckoo hashing according to Theorem 7.3, depending on the momentary value of ε'. The depicted curves correspond to asymmetry factors $c = 0$ (*i.e.* the standard algorithm), $c = 0.2$, $c = 0.3$, and $c = 0.4$ from left to right.

increases if either the size of the data structure or the asymmetry increases. Further, we observe that the standard algorithm offers a slightly better behaviour than the simplified version. If we compare these results to the average size of the maximum tree resp. cyclic component that is given in the Tables 6.1 to 6.6, we observe that the numbers of Table 7.2 are by far below the maximum component sizes. That means, that the insertion procedure involves a limited part of the component only, even in the expected worst case.

Gonnet [1981] provides an asymptotic approximation of the length of the longest probing sequence for double hashing and linear probing, but it turns out that this results do not provide suitable estimates for the table sizes considered here. Hence we provide numerically obtained data for this algorithms too. We observe that double hashing offers the best performance under this aspect. For relatively sparse tables, cuckoo hash algorithms exhibit the next best behaviour, followed by linear probing and Brent's insertion algorithm. But we notice that the cuckoo hash algorithms possess the highest expected worst case otherwise.

Finally, Table 7.3 provides the maximum number of steps that occurred in an insertion operation in the whole sample of size 10^5. Using this data, we are able to choose suitable values for the maximum number of kick-out steps that is performed during an insertion, *i.e.* the choice of the parameter "maxloop" of Algorithm 7.1. Recall that this variable is used to stop insertion procedures, that are very unlikely to succeed. On the other hand, the failure rate increases if this parameter is to small. Hence, we suggest to leave a margin, especially for all settings that exhibit a low failure rate.

m	$\varepsilon = 0.7$	$\varepsilon = 0.4$	$\varepsilon = 0.2$	$\varepsilon = 0.1$	$\varepsilon = 0.06$	$\varepsilon = 0.04$
standard cuckoo hashing						
500	1.1382	1.2727	1.3972	1.5515	1.6782	1.7788
5000	1.1389	1.2718	1.3780	1.4759	1.5753	1.6815
50000	1.1390	1.2717	1.3749	1.4472	1.4982	1.5571
500000	1.1390	1.2717	1.3746	1.4424	1.4789	1.5055
bound	1.1889	1.5272	2.0118	2.5584	2.9930	3.3530
asymmetric cuckoo hashing, $c = 0.1$						
500	1.1269	1.2530	1.3746	1.5335	1.6692	1.7685
5000	1.1275	1.2518	1.3548	1.4529	1.5662	1.6786
50000	1.1275	1.2518	1.3506	1.4220	1.4769	1.5500
500000	1.1275	1.2518	1.3503	1.4166	1.4539	1.4840
asymmetric cuckoo hashing, $c = 0.2$						
500	1.1171	1.2366	1.3614	1.5419	1.7021	1.8299
5000	1.1177	1.2354	1.3363	1.4555	1.6144	1.7931
50000	1.1178	1.2352	1.3316	1.4070	1.4923	1.6450
500000	1.1178	1.2352	1.3311	1.3991	1.4438	1.5112
asymmetric cuckoo hashing, $c = 0.3$						
500	1.1091	1.2240	1.3588	1.6005	1.8381	-
5000	1.1096	1.2218	1.3263	1.4988	1.8415	-
50000	1.1096	1.2215	1.3181	1.4162	1.7007	-
500000	1.1096	1.2214	1.3171	1.3918	1.5367	-
asymmetric cuckoo hashing, $c = 0.4$						
500	1.1021	1.2141	1.3829	1.7749	-	-
5000	1.1025	1.2111	1.3279	1.7592	-	-
50000	1.1026	1.2104	1.3114	1.6090	-	-
500000	1.1026	1.2104	1.3092	1.4666	-	-
simplified cuckoo hashing						
500	1.0836	1.1979	1.3246	1.4669	1.5754	1.6591
5000	1.0840	1.1967	1.3093	1.4106	1.4994	1.5873
50000	1.0841	1.1967	1.3061	1.3881	1.4424	1.4950
500000	1.0841	1.1967	1.3059	1.3851	1.4285	1.4584
bound	1.1889	1.5272	2.0118	2.5584	2.9930	3.3530
linear probing						
asympt.	1.0882	1.2143	1.3333	1.4091	1.4434	1.4615
double hashing						
asympt.	1.0835	1.1889	1.2771	1.3285	1.3508	1.3623
double hashing using Brent's refinement						
997	1.0906	1.2207	1.3380	1.4097	1.4409	1.4574
10007	1.0910	1.2208	1.3378	1.4087	1.4400	1.4564
100003	1.0911	1.2211	1.3382	1.4094	1.4407	1.4571
1000003	1.0910	1.2211	1.3382	1.4093	1.4406	1.4570

Table 7.1: Average number of steps per insertion. We use random 32-bit integer keys and hash functions based on Carter and Wegman's universal family for cuckoo hashing, and consider the average taken over 10^5 successful constructed tables. Furthermore, the table depicts the asymptotic bound of Theorem 7.1 resp. 7.2 and the well known asymptotic results of linear probing and double hashing. Finally, we provide numerical data of the behaviour of Brent's algorithm using hash functions based on the division method. This functions require that the table size equals a prime number. Note that the parameter m equals the total number of memory cells for Brent's algorithm.

m	$\varepsilon = 0.7$	$\varepsilon = 0.4$	$\varepsilon = 0.2$	$\varepsilon = 0.1$	$\varepsilon = 0.06$	$\varepsilon = 0.04$
			standard cuckoo hashing			
500	2.355	4.195	7.424	10.886	13.215	14.710
5000	3.230	7.032	13.141	20.944	27.511	32.608
50000	4.519	10.321	20.105	33.704	46.577	58.502
500000	5.974	13.917	27.912	48.884	70.052	90.984
			asymmetric cuckoo hashing, $c = 0.1$			
500	2.338	4.166	7.424	10.968	13.366	14.909
5000	3.201	7.005	13.231	21.334	28.402	33.955
50000	4.474	10.337	20.281	34.578	48.660	62.207
500000	5.933	13.957	28.283	50.433	73.763	98.096
			asymmetric cuckoo hashing, $c = 0.2$			
500	2.324	4.188	7.602	11.502	14.188	16.013
5000	3.199	7.081	13.725	23.094	31.882	39.382
50000	4.475	10.505	21.230	38.068	57.242	79.108
500000	5.954	14.235	29.721	56.157	89.122	133.566
			asymmetric cuckoo hashing, $c = 0.3$			
500	2.323	4.274	8.054	12.715	16.126	-
5000	3.210	7.291	14.844	27.187	41.325	-
50000	4.522	10.881	23.222	47.033	87.219	-
500000	6.032	14.793	32.846	71.476	155.553	-
			asymmetric cuckoo hashing, $c = 0.4$			
500	2.329	4.420	8.927	15.230	-	-
5000	3.233	7.674	17.047	38.502	-	-
50000	4.610	11.559	27.350	79.395	-	-
500000	6.189	15.827	39.403	138.019	-	-
			simplified cuckoo hashing			
500	2.892	5.386	8.844	12.600	15.134	16.792
5000	4.251	8.331	14.750	22.876	29.830	35.348
50000	5.745	11.717	21.780	35.850	49.200	61.634
500000	7.329	15.340	29.636	51.045	72.865	94.370
			linear probing			
997	3.089	5.589	8.127	9.869	10.696	11.136
10007	4.552	8.530	12.733	15.670	17.077	17.819
100003	6.148	11.811	17.941	22.239	24.349	25.463
1000003	7.857	15.314	23.486	29.320	32.117	33.612
			double hashing			
997	2.805	4.399	5.695	6.457	6.807	6.967
10007	3.878	6.063	7.864	8.952	9.420	9.675
100003	4.977	7.792	10.129	11.538	12.166	12.482
1000003	6.102	9.546	12.425	14.168	14.953	15.347
			double hashing using Brent's refinement			
997	3.496	5.835	7.636	8.690	9.158	9.407
10007	5.027	7.894	10.287	11.725	12.345	12.695
100003	6.444	9.934	12.876	14.735	15.553	15.978
1000003	7.636	11.882	15.511	17.685	18.688	19.208

Table 7.2: Average *maximum* number of steps of an insertion. We use the same setup as described in Table 7.1. All entries of the table give the average taken over 10^5 successful constructed tables. Note that the parameter m equals the total number of memory cells for non-cuckoo hashing algorithms.

m	$\varepsilon = 0.7$	$\varepsilon = 0.4$	$\varepsilon = 0.2$	$\varepsilon = 0.1$	$\varepsilon = 0.06$	$\varepsilon = 0.04$
			standard cuckoo hashing			
500	13	29	59	77	87	83
5000	11	39	87	125	191	183
50000	11	41	79	165	273	357
500000	15	37	75	227	351	611
			asymmetric cuckoo hashing, $c = 0.1$			
500	13	31	59	77	79	91
5000	11	27	89	125	179	219
50000	13	41	73	227	285	345
500000	13	33	87	173	307	477
			asymmetric cuckoo hashing, $c = 0.2$			
500	9	35	63	81	87	93
5000	11	35	85	153	229	223
50000	11	35	91	177	363	433
500000	15	37	83	233	453	697
			asymmetric cuckoo hashing, $c = 0.3$			
500	13	29	51	97	97	-
5000	13	29	83	187	215	-
50000	13	37	91	273	497	-
500000	15	35	97	343	853	-
			asymmetric cuckoo hashing, $c = 0.4$			
500	11	33	75	113	-	-
5000	11	39	107	237	-	-
50000	13	39	171	461	-	-
500000	19	41	151	779	-	-
			simplified cuckoo hashing			
500	11	29	63	77	90	95
5000	13	29	74	165	198	190
50000	14	34	89	169	327	413
500000	16	34	83	219	368	530

Table 7.3: Maximum number of steps of an insertion during 10^5 successful constructed tables. We use random 32-bit integer keys and hash functions based on Carter and Wegman's universal family.

Chapter 8

Search

8.1 Introduction

Finally, we consider the average cost of both successful and unsuccessful search operations. Similar to the analysis of insertion operations, we count the number of memory cells accessed during the operation. Of course, we may perform a search in at most two steps. However, we are able to perform a lot of search operations in one step only under certain circumstances. Assume that we always start a search after a key x at the position $h_1(x)$. As a consequence, a successful search takes one step only, if the cell determined by the first hash function holds x. Further, it is guaranteed that a search operation is unsuccessful, if the position indicated by h_1 is empty, as long as our data structure meets the following rules:

- We always try to insert a key using h_1 first, the second hash function is used only if the inspected cell is already occupied.
- If we delete an element of the first table resp. addressed by the first hash function, it is not allowed to mark the cell "empty" instead we have to use a marker "previously occupied". This is similar to the deletions in hashing with open addressing, *cf.* Knuth [1998].

Listing 8.1 gives the details of the search algorithm. A C++ version of this algorithm can be found in Appendix A and it is also included in the attached CD-ROM.

Similar to the analysis of linear probing and uniform probing in Knuth [1998], our analysis considers hashing without deletions. Note that it is not possible to delete a key such that the table ends up as it would have been, if this key had never been inserted, except if we are willing to inspect the whole data structure. Because of this, deletions are likely to increase the number of keys stored using the second hash function what decreases the performance of search operations. However, it should be clear that our results can be applied to the situations where deletions are very rare.

Knuth [1998] suggests the usage of ideas originally developed for garbage collection to increase the performance of hash tables based on open addressing, if it is necessary to allow deletions. This suggestions are helpful for cuckoo hashing too. One might implement a counter telling how many keys try to occupy a given position. Whenever

Listing 8.1: Optimal search for cuckoo hashing.

```
function search(s)
    if table1[h1(s)] == s
        return true
    end if
    if table1[h1(s)] == empty
        return false
    end if
    if table2[h2(s)] == s
        return true
    end if
    return false
end function
```

this counter equals one, we might delete the key completely. Furthermore, it is a good idea to parse the whole table trying to move keys back to its primary position whenever a lot of "previously occupied" markers are present.

We want to emphasise that the notations used in the analysis of hashing with open addressing are a little bit different, so we state the results in terms of the load factor $\alpha = n/(2m)$. As a consequence, the results can be directly compared.

8.2 Simplified cuckoo hashing

Similar to the previous chapters, we start considering the simplified algorithm, although the analysis is in some sense more complicated. Because of this, we can not provide the exact behaviour of successful search operations in contrast to the other variants of cuckoo hashing.

Theorem 8.1. *Suppose that $\alpha \in (0, 1/2)$ is fixed and consider a successful simplified cuckoo hash of $n = \lfloor 2m\alpha \rfloor$ data points into a table of size $2m$ without deletions. Then the expected number of inspected cells C_n of an successful search satisfies the relation*

$$2 - \frac{1 - e^{-\alpha}}{\alpha} + \mathcal{O}\left(m^{-1}\right) \le C_n \le 2 - \frac{1 - e^{-2\alpha}}{2\alpha} + \mathcal{O}\left(m^{-1}\right). \tag{8.1}$$

Furthermore, the expected number of steps of an unsuccessful search is given by

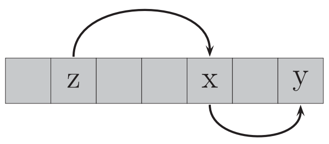
$$1 + \alpha, \tag{8.2}$$

and the variance equals

$$\alpha(1 - \alpha). \tag{8.3}$$

Proof. The number of steps of an unsuccessful search is determined by the number of empty cells which equals $2m - n$. Thus, we need one step only with probability $1 - \alpha$ and two steps otherwise.

Consider an arbitrary selected cell z of the first table. The probability, that none of the randomly selected values $h_1(x_1), \dots, h_1(x_n)$ equals z is given by $p = (1 - 1/(2m))^n$. Let p_s denote the probability that z is not addressed by the first hash function, conditioned



	x	y	z
h_1	2	5	2
h_2	5	7	5

Figure 8.1: Additional search costs in simplified cuckoo hashing. Two memory cells are accessible by h_1 for the current data, but only the key z can be found in a single step.

on the property that the construction of the hash table succeeds, and let p_a denote the probability that z is not addressed by the first hash function, conditioned on the property that the construction of the hash table is unsuccessful. By the law of total probability, we have $p = p_s + p_a$. Because of Theorem 4.1, the relation $p_a = \mathcal{O}(m^{-1})$ holds. Thus, we obtain that the probability that an arbitrary selected cell is not a primary storage location, equals

$$p_s = (1 - 1/(2m))^n + \mathcal{O}\left(m^{-1}\right) = e^{-\alpha} + \mathcal{O}\left(m^{-1}\right). \tag{8.4}$$

Hence, $2m(1 - p_s)$ is the expected number of cells accessed by h_1 if the table holds n keys. It is for sure that each of this memory slots stores a key, because an insertion always starts using h_1. However, assume that the primary position of one of this keys y equals the secondary storage position of an other key x. If x is kicked-out, it will subsequently kick-out y. Thus, the total number of steps to find all keys increases. Figure 8.1 gives an example of such a situation.

Let q denote the probability, that a cell z, which is addressed by both hash functions, is occupied by a key x such that $h_1(x) = z$ holds. Then, the expected number of keys reachable with one memory access equals

$$2m\left((1 - p_s)p_s + (1 - p_s)^2 q\right). \tag{8.5}$$

By setting $q = 1$ and $q = 0.5$ we get the claimed results. The latter value corresponds to a natural equilibrium. $\square$

8.3 Standard cuckoo hashing

Of course, the standard algorithm corresponds to asymmetric cuckoo hashing possessing a coefficient of asymmetry c equal to zero. In contrast to the analysis given in the previous sections, the analysis of the search performance of the asymmetric variant is similarly difficult to the standard algorithm. Hence, there it is obsolete to provide a separate proof. To make the results more approachable for the reader, we present the results separately.

Theorem 8.2. *Suppose that $\alpha \in (0, 1/2)$ is fixed and consider a successful cuckoo hash of $n = \lfloor 2m\alpha \rfloor$ data points into two tables of size m without deletions. Then, the expected number of inspected cells of an successful search is asymptotically given by*

$$2 - \frac{1}{2\alpha}\left(1 - e^{-2\alpha}\right) + \mathcal{O}\left(m^{-1}\right), \tag{8.6}$$

and its variance by

$$\frac{1}{2\alpha}\left(1-e^{-2\alpha}\right)-\left(\frac{1}{2\alpha}\left(1-e^{-2\alpha}\right)\right)^2+o(1). \tag{8.7}$$

Furthermore, the expected number of steps of an unsuccessful search is asymptotically given by

$$2-e^{-2\alpha}+\mathcal{O}\left(m^{-1}\right), \tag{8.8}$$

and the variance by

$$e^{-2\alpha}-e^{-4\alpha}+\mathcal{O}\left(m^{-1}\right). \tag{8.9}$$

Proof. See proof of Theorem 8.3. □

8.4 Asymmetric cuckoo hashing

Theorem 8.3. *Suppose that $c\in[0,1)$ and $\alpha\in(0,\sqrt{1-c^2}/2)$ are fixed and consider a successful asymmetric cuckoo hash of $n=\lfloor(1-\varepsilon)m\rfloor$ data points into two tables of size $m_1=\lfloor m(1+c)\rfloor$ respectively $m_2=2m-m_1$ without deletions. Then, the expected number of inspected cells of an successful search is asymptotically given by*

$$2-\frac{1+c}{2\alpha}\left(1-e^{-2\alpha/(1+c)}\right)+\mathcal{O}\left(m^{-1}\right), \tag{8.10}$$

and its variance by

$$\frac{1+c}{2\alpha}\left(1-e^{-2\alpha/(1+c)}\right)-\left(\frac{1+c}{2\alpha}\left(1-e^{-2\alpha/(1+c)}\right)\right)^2+o(1). \tag{8.11}$$

Furthermore, the expected number of steps of an unsuccessful search is asymptotically given by

$$2-e^{-2\alpha/(1+c)}+\mathcal{O}\left(m^{-1}\right), \tag{8.12}$$

and the variance by

$$e^{-2\alpha/(1+c)}-e^{-4\alpha/(1+c)}+\mathcal{O}\left(m^{-1}\right). \tag{8.13}$$

Proof. Consider an arbitrary selected cell z contained in the first table. The probability, that none of the randomly selected values $h_1(x_1),\dots,h_1(x_n)$ equals z is given by $p=\left(1-\lfloor m(1+c)\rfloor^{-1}\right)^n$. Let p_s denote the probability that z is empty, conditioned on the property that the construction of the hash table succeeds. Let p_a be the probability that z is empty, conditioned on the property that the construction of the hash table is unsuccessful. By the law of total probability, we have $p=p_s+p_a$. Using Theorem 4.3, we see that the relation $p_a=\mathcal{O}(m^{-1})$ holds. Thus, we obtain

$$p_s=\left(1-\lfloor m(1+c)\rfloor^{-1}\right)^n+\mathcal{O}\left(m^{-1}\right)=e^{-2\alpha/(1+c)}+\mathcal{O}\left(m^{-1}\right). \tag{8.14}$$

This equals the probability that the first inspected cell during a search is empty. The remaining unsuccessful searches take exactly two steps. Hence it is straightforward to calculate expectation and variance.

Similarly, we further obtain, that the expected number of occupied cells of the first table equals $\lfloor m(1+c)\rfloor\left(1-e^{-2\alpha/(1+c)}\right)+\mathcal{O}(1)$. This gives us the expected number of keys which might be found in a single step, while the search for any other key takes exactly two steps. Using this, it is straightforward to calculate the expectation, however we require a different approach to determine the variance. Instead of considering a successful constructed data structure, we assume that each of the m_1^n sequences of hash values of h_1 is equally likely. Note that we obtain the same distribution in limit because of to Theorem 4.3 and Lemma 6.1. Similar to the analysis of hashing with chaining (see, *e.g.*, Knuth [1998]) we use the generating function

$$P_n(z) = \sum_{k=0}^{n} \binom{n}{k} \frac{1}{m_1^k} \left(1 - \frac{1}{m_1}\right)^{n-k} z^k = \left(1 + \frac{z-1}{m_1}\right)^n . \tag{8.15}$$

If we are interested in the second moment M_2, we attach a "weight" of $4k - 3 + 3\delta_{k,0}$ to a list of length k where $\delta_{k,0}$ denotes Kronecker's delta and obtain

$$\begin{aligned} M_2 &= \frac{1}{n m_1^n} \sum_{k_1+\cdots+k_{m_1}=n} \binom{n}{k_1,\ldots,k_{m_1}} \Big(4k_1 - 3 + 3\delta_{k_1,0} + \cdots + 4k_{m_1} - 3 + 3\delta_{k_{m_1},0}\Big) \\ &= \frac{m_1}{n} \sum_{k=0}^{n} (4k - 3 + 3\delta_{k,0}) \binom{n}{k} \frac{1}{m_1^k} \left(1 - \frac{1}{m_1}\right)^{n-k} = \frac{m_1}{n} \big(4P_n'(1) - 3P_n(1) + 3P_n(0)\big) \\ &= 4 - 3\frac{m_1}{n} \left(1 - \left(1 - \frac{1}{m_1}\right)^n\right), \end{aligned} \tag{8.16}$$

which allows us to calculate the variance. □

8.5 Comparison and Conclusion

We start comparing the obtained asymptotic results. Figure 8.2 depicts the asymptotic behaviour of a successful search, depending on the load factor α. For simplified cuckoo hashing, we plot both bounds together with a numerically obtained curve. We conclude that the simplified algorithm offers the best performance, even if we compare it to linear probing and double hashing.

The good performance of successful search operations of cuckoo hash algorithms holds due to the fact, that the load is unbalanced, because the majority of keys will be usually stored using the first hash function. This explains the increased performance of the asymmetric variant too. However, recall that the algorithm possessing an asymmetry $c = 0.3$ allows a maximum load factor of approximately 0.477 according to Theorem 4.3, hence the corresponding curve stops at this load factor. Numerical results show that the failure rate increases dramatically if this bound will be exceeded.

Further, Figure 8.3 shows a plot according to an unsuccessful search. As an important result, simplified cuckoo hashing offers again the best performance. Compared to standard cuckoo hashing and the asymmetric algorithm, this can be explained by the higher probability of hitting an empty cell. Again, this is related to the fact, that the load is unbalanced, because the chance of hitting an empty cell in the first table of a variant using two separate tables is less than the overall percentage of empty cells. Compared to linear probing and double hashing, the chance of hitting a non-empty cell in the first step is identical. However, simplified cuckoo hashing needs exactly two steps in such a case, but there is a non-zero probability that the two other algorithms will need more than one additional step.

Again, we provide numerical results, which can be found in Table 8.1. Blank entries refer to supercritical settings, where our asymptotic approximations are not applicable, *cf.* Table 4.1. Since the cost of an unsuccessful search is deterministic for the simplified version and closely related to the behaviour of the successful search otherwise, we concentrate on the successful search. From the results given in the table, we find that our asymptotic results can be interpreted as a good approximation, even for hash tables of small size. Furthermore, the size of the tables does not seem to have hardly any influence on the behaviour of successful searches.

In particular, we notice that the simplified algorithm offers the best performance of all variants of cuckoo hashing for all investigated settings. This can be explained by the increased number of keys accessible in one step, although this increase is not guaranteed because of the effect described in Figure 8.1. However, our experiments show that the influence is noticeable, but not dramatic. As long as α is small, the actual behaviour is close to the lower bound of Theorem 8.1. Additionally, we compare the performance of simplified cuckoo hashing and the variants of double hashing offering improved retrieval power, namely Brent's variation and binary tree hashing. A brief description of these algorithms and references are given in Chapter 1. Note that the behaviour of the latter two algorithms is almost indistinguishable for such small load factors. Hence we do not provide results for both algorithms in Table 8.1. From our numerical results, we see that these three algorithms behave approximately equal regarding successful searches.

Concerning asymmetric cuckoo hashing, these results verify, that the performance of search operations increases, as the asymmetry increases. However, the simplified algorithm offers even better performance without the drawbacks of a lower maximal fill ratio and increasing failure probability, *cf.* Chapter 4.

Note that load balancing implemented by starting an insertion at a random selected table instead of always choosing the first, would decrease the performance of unsuccessful searches, although it would increase the number of empty cells of the first table. This is due to the fact, that this change requires the inspection of the second storage location, if the first is found empty. Furthermore, this modification would further have a negative influence on the performance for finding inserted keys and is therefore not recommended.

We conclude that simplified cuckoo hashing offers the best average search performance over all algorithms considered in this thesis, for all feasible load factors. Thus it is highly recommendable to use this variant instead of any other version of cuckoo hashing discussed here.

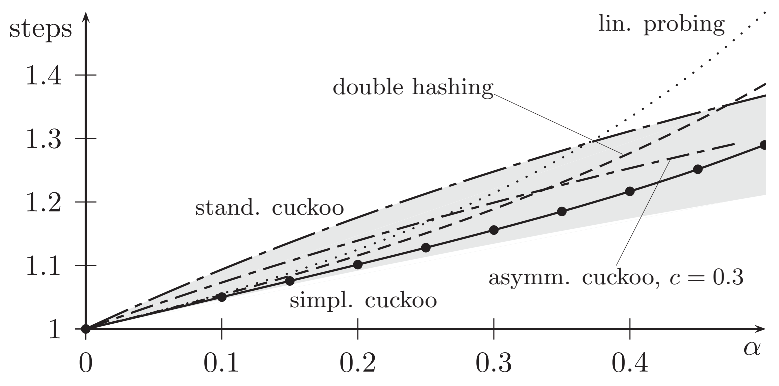


Figure 8.2: Comparison of successful search. The curves are plotted from the results of Theorem 8.1 resp. 8.2 together with the well known asymptotic results of the standard hash algorithms. For simplified cuckoo hashing, the grey area shows the span between the upper and lower bound. The displayed curve is obtained numerically with tables containing 10^5 cells and sample size $5 \cdot 10^4$.

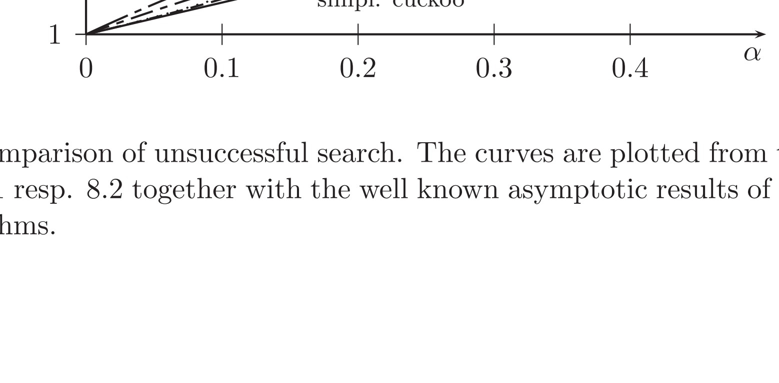


Figure 8.3: Comparison of unsuccessful search. The curves are plotted from the results of Theorem 8.1 resp. 8.2 together with the well known asymptotic results of the standard hash algorithms.

m	$\varepsilon = 0.7$	$\varepsilon = 0.4$	$\varepsilon = 0.2$	$\varepsilon = 0.1$	$\varepsilon = 0.06$	$\varepsilon = 0.04$
			standard cuckoo hashing			
500	1.1353	1.2474	1.3112	1.3400	1.3510	1.3563
5000	1.1360	1.2479	1.3116	1.3406	1.3517	1.3571
50000	1.1360	1.2480	1.3116	1.3406	1.3517	1.3572
500000	1.1361	1.2480	1.3117	1.3406	1.3517	1.3572
asympt.	1.1361	1.2480	1.3117	1.3406	1.3517	1.3572
			asymmetric cuckoo hashing, $c = 0.1$			
500	1.1241	1.2287	1.2890	1.3164	1.3269	1.3320
5000	1.1247	1.2292	1.2894	1.3170	1.3276	1.3328
50000	1.1248	1.2292	1.2894	1.3171	1.3277	1.3329
500000	1.1248	1.2292	1.2894	1.3171	1.3277	1.3329
asympt.	1.1248	1.2292	1.2894	1.3171	1.3277	1.3329
			asymmetric cuckoo hashing, $c = 0.2$			
500	1.1145	1.2125	1.2695	1.2958	1.3057	1.3105
5000	1.1151	1.2130	1.2701	1.2964	1.3065	1.3115
50000	1.1152	1.2131	1.2701	1.2965	1.3066	1.3116
500000	1.1152	1.2131	1.2701	1.2965	1.3067	1.3117
asympt.	1.1152	1.2131	1.2701	1.2965	1.3067	1.3117
			asymmetric cuckoo hashing, $c = 0.3$			
500	1.1064	1.1984	1.2526	1.2774	1.2868	-
5000	1.1069	1.1990	1.2532	1.2783	1.2879	-
50000	1.1070	1.1990	1.2532	1.2784	1.2881	-
500000	1.1070	1.1990	1.2532	1.2784	1.2881	-
asympt.	1.1070	1.1990	1.2532	1.2784	1.2881	-
			asymmetric cuckoo hashing, $c = 0.4$			
500	1.0994	1.1862	1.2376	1.2610	-	-
5000	1.0998	1.1867	1.2382	1.2621	-	-
50000	1.0999	1.1867	1.2383	1.2623	-	-
500000	1.0999	1.1867	1.2383	1.2623	-	-
asympt.	1.0999	1.1867	1.2383	1.2623	-	-
			simplified cuckoo hashing			
500	1.0750	1.1553	1.2163	1.2505	1.2651	1.2723
5000	1.0755	1.1557	1.2167	1.2516	1.2666	1.2742
50000	1.0756	1.1558	1.2168	1.2516	1.2667	1.2745
500000	1.0755	1.1558	1.2168	1.2516	1.2667	1.2745
lower bound	1.0714	1.1361	1.1758	1.1947	1.2021	1.2058
upper bound	1.1361	1.2480	1.3117	1.3406	1.3517	1.3572
			linear probing			
asympt.	1.0882	1.2143	1.3333	1.4091	1.4434	1.4615
			double hashing			
asympt.	1.0835	1.1889	1.2771	1.3285	1.3508	1.3623
			double hashing using Brent's refinement			
asympt.	1.0759	1.1573	1.2179	1.2510	1.2649	1.2721

Table 8.1: Average number of steps of a successful search. We use random 32-bit integer keys, hash functions based on Carter and Wegman's universal family, and consider the average taken over 10^5 successful constructed tables. Furthermore, the table depicts the asymptotic results of Theorem 8.1 resp. 8.3 and the well known asymptotic results of algorithms based on open addressing. See Chapter 9 for further details on implementation and setup.

Chapter 9

Experimental Settings

The major part of the analysis performed in the previous chapters leaded to asymptotic approximations instead of exact results. Consequently, one might argue that the actual behaviour of practical relevant settings might be different, because the parameters are to "small". To overcome this weak point, we provide numerically obtained results whenever possible. Hence, we can show that the asymptotic approximations provide suitable estimations. Further, our analysis of the construction cost of cuckoo hashing provides just an upper bound. We give numerical results to investigate the accuracy of this bound and to obtain a reliable estimation for the actual costs. Finally, the results provides a form of informal justification of our results. It is a good idea to check the calculation iteratively if the obtained results seem to be totally different.

The relevant numerical data for each chapter can be found at its end, together with a discussion of the results. This chapter does not focus on the results itself, but on the methods which are used to obtain them.

All numerical data are obtained using C++ programs on a workstation equipped with two Intel Xeon 5150 2.66Ghz dual core processors and 4GB RAM. The attached CD-ROM provides all relevant source codes. The most important listings can also be found in Appendix A. In principle we use two different kinds of programs, that are described in the next sections.

9.1 A random graph growth process

The first kind of program simulates the growth of the bipartite or usual cuckoo graph. It is based on an algorithm similarly constructed to the usual implementation of Kruskal's minimum spanning tree algorithm, see, *e.g.*, Cormen et al. [2001]. More precisely, we use a dynamic disjoint-set data structure to represent the connected components of the graph. Each set of this data structure corresponds to a unique component of the graph and it consists of the nodes contained in the corresponding component. Furthermore, each set is uniquely determined by an arbitrary selected representative, that is one member of the set. This information is internally stored in an array `p`. Additionally, we use a second array named `cyclic`, to store flags, that indicate if the corresponding component contains a cycle. The process starts with a graph consisting of isolated nodes only, hence each node is represented by itself. We continue successively inserting new randomly chosen

Listing 9.1: Dynamic disjoint-set data structure.

```
function make_set(u)
    p[u] = u
end function


function find_set(u)
    if u != p[u]
        u = find_set(p[u])
    end if
    return u
end function


function union(u,v)
    p[find_set(v)] = find_set(u)
end function
```

edges. There exist two different cases which we now treat separately:

- The new edge connects two different components. If both nodes belong to cyclic components, we obtain a bicyclic component and the process stops. Otherwise, we merge the corresponding sets by selecting one of the two representatives which becomes the new representative of the united set. Clearly, the new component is a tree if and only if both original components are trees.

- The new edge is incident on two nodes belonging to the same set, *i.e.* the nodes possess the same representative. Hence, the insertion of the last edge created a loop. If the cycle flag of this component is already set, a bicyclic component occurs and the process is stopped. Otherwise, we set the flag and continue the process.

Note that it is not required to update all of the array's `p` entries during a union operation. Instead of storing the representative for all nodes directly, we just store the parent node. The representative is the unique node of a component that equals its parent and there exists an easy recursive algorithm to determine this node. Listing 9.1 provides a pseudo code implementation of a dynamic disjoint-set, but it does not test for the occurrence of complex components. See the files tables.hpp and tables.cpp for a full implementation.

First of all, this programs allow us to calculate the percentage of successful constructed tables. Furthermore, they provide results about the structure of the cuckoo graph, such as size of the tree components, and the size of cyclic components. However, it is impossible to determine the size of a cycle contained in a unicyclic component without large additional effort. Since this number is not important for the analysis of cuckoo hashing, we did not implement this feature. Finally, we may not want to stop the creation of the graph if a fixed number of edges is reached, but at the occurrence of the first bicylce. This mode is supported by the software too. Numerical results obtained using this software are given in Chapters 5 and 6.

9.2 Implementation of cuckoo hashing

Further, we use programs that build up a concrete cuckoo hash table in memory. This software is based on the C++ template library developed in Kutzelnigg [2005]. Most of the routines have been rewritten and the support of d-cuckoo hashing, asymmetric table sizes and the simplified algorithm represent completely new features. The software provides different kinds of hash functions and it is easy to add further ones. In particular, the following modes are supported:

- Simulation mode: Instead of random keys, we use the numbers from 1 to n as keys and arrays filled with pseudo random numbers to provide the hash values. Hence, we achieve almost perfect random behaviour with high performance. This mode is used to provide the data given in Chapters 4 and 5.
- The universal class of hash functions of Carter and Wegman [1979], mentioned in Chapter 1. However, we use a slightly modified version that does not require that the size of the table equals a power of two. More precisely we calculate a 32-bit hash value as usual and obtain the actual hash value reducing this number modulo m. This mode is used to obtain the data concerning the runtime presented in Chapters 7 and 8.
- Polynomial hash functions. So far, the algorithm supports functions of type $ax + b \mod m$, where x denotes the key, a and b are random 32-bit numbers, m is a prime number, and the multiplication is performed without taking care of the overflow. As mentioned in Chapter 1, these functions are suitable for table sizes up to approximately 10^5.

This kind of program does not provide information about the cuckoo graph, but we may count the number of successful constructed tables and we obtain information about the behaviour of the insertion and search procedures of cuckoo hashing.

9.3 Random numbers

The quality of the output of all this programs depends heavily of the pseudo random number generator in use. This software includes support respectively implementations of two reliable random number generators.

- The KISS-generator by Marsaglia. It is based on the additive combination of three separate random number generators, namely a congruential generator, linear shift-register and a multiply-with-carry generator. The file random.hpp contains our implementation of this algorithms.
- The Mersenne Twister developed by Matsumoto and Nishimura [1998]. We use a C++ implementation of Richard J. Wagner, which is slightly modified for our purpose. See the file mt.hpp for further details.

All numerical data given in this thesis are obtained using the KISS-generator, because this algorithm is faster and we could not find significant differences in the output that would require the usage of the Mersenne Twister.

Furthermore, it is possible to use a C or C++ implementation of an arbitrary random number generator with very little afford. All that has to be done, is the implementation of an interface class, as it is provided for the usage of the Mersenne Twister.

Appendix A

Selected C++ code listings

This chapter gives some of the most important C++ code listings. All this files (among many others) are included on the attached CD-ROM. It also contains demo programs which show the usage of this library and contains the licence information, see readme.txt. A preliminary version of this library was developed for the master thesis Kutzelnigg [2005] using Microsoft Visual C. The current version is tested with various versions of gcc under Linux, but other compilers and operating systems should work too. In the following, we give a brief description of the design of the hash library, followed by some listings.

A.1 The hash library

`Hash_base` is an abstract class, that operates as an interface for the usage of all classes contained in the library. Hence, it is possible to use functions of the library without further knowledge on the implementation. All classes support the following operations:

- `search(S)`: search for a key `S`.
- `insert(S)`: insertion of the key `S`.
- `del(S)`: delete `S`.
- `get_n()`: number of keys currently stored in the table.
- `out(std::ofstream*)`: output of internal memory structure using a stream, for small tables only.

Furthermore, all classes support the operators `new` and `delete`. Thus, it is possible to exchange the used hashing method by modifying a single code line only. The hierarchic structure is depicted in Figure A.1. Additionally, Table A.1 gives an overview of the functionality. All classes related to cuckoo hashing are defined in `cuckoo.hpp` and implemented in `cuckoo.cpp`. Further, `cuckoo_main.cpp` provides the implementation of an executable file, that was used to generate the numerical results. The remaining part of the library follows a likewise structure.

name	functionality	data type
`Hash_cuck`	pure virtual base class for cuckoo hashing	arbitrary
`Hash_cuck_pol`	cuckoo hashing, polynomial hash functions	`unsigned int`
`Hash_cuck_cw`	cuckoo hashing, hash functions based on Carter and Wegman's universal family	`unsigned int`
`Hash_cuck_sim`	cuckoo hashing, pseudo random generator to simulate good hash functions	`unsigned int`
`Hash_d_cuck`	pure virtual base class for d-cuckoo hashing	arbitrary
`Hash_d_cuck_pol`	d-cuckoo hashing, polynomial hash functions	`unsigned int`
`Hash_d_cuck_cw`	d-cuckoo hashing, hash functions based on Carter and Wegman's universal family	`unsigned int`
`Hash_d_cuck_sim`	d-cuckoo hashing, pseudo random generator to simulate good hash functions	`unsigned int`
`Hash_open`	pure virtual base class for standard algorithms based on open addressing	arbitrary
`Hash_open_lin`	linear probing, hash function $S \mod m$	`unsigned int`
`Hash_open_dou`	double hashing, hash functions $S \mod m$, $1 + (S \mod (m-2)) \mod m$	`unsigned int`
`Hash_open_bre`	Brent's variation of double hashing, hash functions as for `Hash_open_dou`	`unsigned int`
`Hash_open_rob`	Robin Hood hashing, hash functions as for `Hash_open_dou`	`unsigned int`
`Hash_open_gen`	double hashing, arbitrary hash functions	arbitrary

Table A.1: Classes of the hash library.

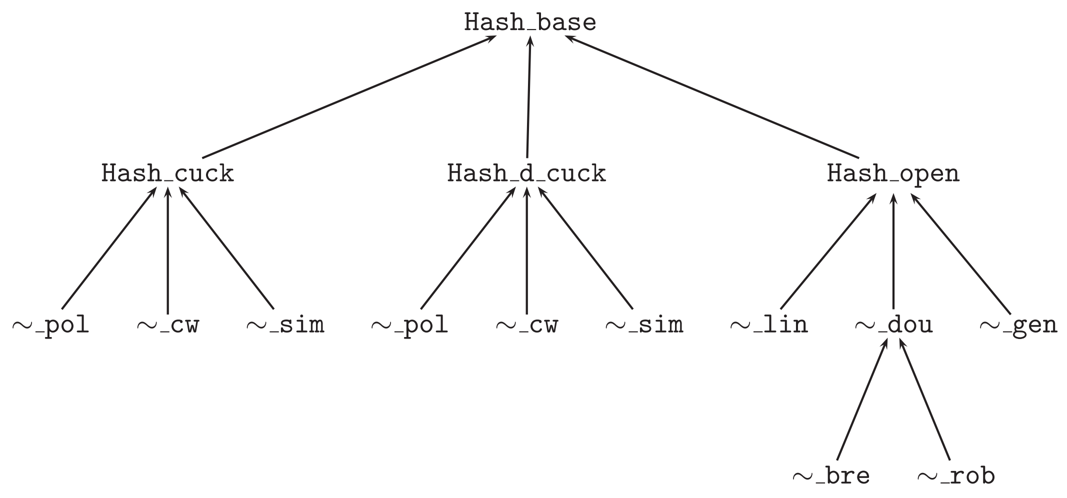


Figure A.1: Hierarchy of the C++ classes. The symbol ∼ refers to the name of the corresponding parent class, *i.e.* the full name of the leftmost class is `Hash_cuck_pol`.

Listing A.1: base.hpp

```
////////////////////////////////////////////////////////////
// Hashing Library Interface
// by Reinhard Kutzelnigg
// bug reports and comments to kutzelnigg@dmg.tuwien.ac.at
// www.dmg.tuwien.ac.at/kutzelnigg
// last updated: $Date: 2007/09/24 07:57:51 $
// distributed under the conditions of the LGPL
////////////////////////////////////////////////////////////


#ifndef HASH_BASE_H___
#define HASH_BASE_H___

#include <malloc.h>
#include <fstream>

typedef unsigned int uint;
typedef signed char schar;
typedef unsigned char uchar;

// Interface for tempate hash library
// all derived classes provide at least:
//   operators  new und delete
//   search(S): search for key  S
//   insert(S): insertion of key S
//   del(S):     delete key S
//   get_n():    return number of inserted keys
//   out(std::ostream*):
//               writes internal structure of the hash table in a text file
//               large tables appear truncated
//               intended for testing and generating of hash demos
template<class S> class Hash_base
{
public:
    virtual ~Hash_base() { }
    void* operator new(size_t s) { return malloc(s); }
    void operator delete(void* p) { free(p); }
    virtual int search(S) = 0;
    virtual int insert(S) = 0;
    virtual void del(S) = 0;
    virtual int get_n() = 0;
    virtual void out(std::ostream*) = 0;
};

#endif // HASH_BASE_H___
```

Listing A.2: cuckoo.hpp

```
///////////////////////////////////////////////////////////
// Cuckoo Hashing Library
// by Reinhard Kutzelnigg
// bug reports and comments to kutzelnigg@dmg.tuwien.ac.at
// www.dmg.tuwien.ac.at/kutzelnigg
// last updated: $Date: 2008/06/29 07:24:18 $
// distributed under the conditions of the LGPL
///////////////////////////////////////////////////////////

// References:
// Donald E. Knuth, The art of computer programming, volume III:
// Sorting and searching, second ed., Addison-Wesley, Boston, 1998.
// Rasmus Pagh and Flemming Friche Rodler, Cuckoo hashing, RS-01-32,
// BRICS, Department of Computer Science, University of Aarhus, 2001.

#ifndef CUCKOO_HASH_H___
#define CUCKOO_HASH_H___

#include <iostream>
#include <sstream>
#include <fstream>
#include "base.hpp"      // Interface of my Hash library
#include "random.hpp"    // my implementation of a random number generator


enum {empty, used, deleted};

// pure virtual base class for Cuckoo Hashing
// provides full functionality, except implemtation of the hash funct. h1, h2
template<class S> class Hash_cuck : public Hash_base<S>
{
protected:
    int m;             // size of 1 table
    int m2;            // size of 2 table (usually equal to m)
    int n;             // number of actual inserted elements
    int maxloop;       // max. number of loops for an insertion
    S *table1;         // both hash tables use dynamic allocated memory
    S *table2;
    virtual int h1(S s) = 0;    // hash function for table 1
    virtual int h2(S s) = 0;    // hash function for table 2
private:
    schar *status1;   // status table indicates if the cell is occupied
    schar *status2;   // 0: occupied, -1 empty, -2 deleted;
public:
    Hash_cuck(int m, int maxloop);              // standard constructor
    Hash_cuck(int m, int m2, int maxloop);      // different table sizes
    ~Hash_cuck();
    int search(S);
    int insert(S);
    void del(S);
    int get_n() { return n; }
    void out(std::ostream*);
    void occupancy(uint &n1);
};


// Implementation for unsigned int keys
// hash functions of type (s*a+b) %m where
//          a,b: random numberes 0,1..u-1
//         prime number table size strongly recommended!
//         (calculations with 32 bit, truncation possible)
class Hash_cuck_pol : public Hash_cuck<uint>
{
private:
    uint a1, a2, b1, b2;
    int h1(uint s) { return (a1*s+b1) % m; }
    int h2(uint s) { return (a2*s+b2) % m2; }

public:
    Hash_cuck_pol(int m, int maxloop, RandBase* random);
    Hash_cuck_pol(int m, int m2, int maxloop, RandBase* random);
    Hash_cuck_pol(int m, int maxloop, uint a1, uint b1, uint a2, uint b2);
    ~Hash_cuck_pol() { }
};


// Implementation for unsigned int keys
// Carter - Wegman hash functions
class Hash_cuck_cw : public Hash_cuck<uint>
{
private:
    int r1[128];
    int r2[128];
    int h1(uint s);
    int h2(uint s);
public:
    Hash_cuck_cw(int m, int maxloop, RandBase* random);
    Hash_cuck_cw(int m, int m2, int maxloop, RandBase* random);
    ~Hash_cuck_cw() { }
};


// Implementation ONLY for SIMULATION!!!
// (used to acquire the succes rate of table constructions)
// uses unsigned int keys 1,2,3,...,n_max
// and tables containing random numbers as hash functions
class Hash_cuck_sim : public Hash_cuck<uint>
{
private:
    int n_end;
    uint *hv1, *hv2;
    int h1(uint s) { return hv1[s%n_end]; }
    int h2(uint s) { return hv2[s%n_end]; }
public:
    Hash_cuck_sim(int m, int n_end, RandBase* random);
    Hash_cuck_sim(int m, int m2, int n_end, RandBase* random);
    ~Hash_cuck_sim();
};


// general case: arbitrary hash functions via function pointers
template<class S> class Hash_cuck_gen : public Hash_cuck<S>
{
private:
    int (*ph1)(S, int);
    int (*ph2)(S, int);
    int h1(S s) { return *ph1(s, Hash_cuck_gen::m); };
    int h2(S s) { return *ph2(s, Hash_cuck_gen::m); };

public:
    Hash_cuck_gen(int m, int maxloop, int (*h1)(S, int), int (*h2)(S, int));
};


// The definition of the template class is needed by the compiler to create
// instances of the class.
// Until the C++ keyword "export" is supported, the only (recommended) way
// is to include this definition in the header.
#include "cuckoo.cpp"
#endif //CUKCOO_HASH_H___
```

Listing A.3: cuckoo.cpp

```
//////////////////////////////////////////////////////////////
// Cuckoo Hashing Library
// by Reinhard Kutzelnigg
// bug reports and comments to kutzelnigg@dmg.tuwien.ac.at
// www.dmg.tuwien.ac.at/kutzelnigg
// last updated: $Date: 2008/06/29 07:24:18 $
// distributed under the conditions of the LGPL
//////////////////////////////////////////////////////////////

#include "cuckoo.hpp"

template<class S> Hash_cuck<S>::Hash_cuck(int m, int maxloop)
{
    Hash_cuck::m = m;
    Hash_cuck::m2 = m;
    Hash_cuck::maxloop = maxloop;
    n = 0;
    table1 = new S[m];
    table2 = new S[m];
    status1 = new schar[m];
    status2 = new schar[m];

    for (int i=0 ; i<m ; i++)
    {
        status1[i] = empty;
        status2[i] = empty;
    }
}


template<class S> Hash_cuck<S>::Hash_cuck(int m, int m2, int maxloop)
{
    Hash_cuck::m = m;
    Hash_cuck::m2 = m2;
    Hash_cuck::maxloop = maxloop;
    n = 0;
    table1 = new S[m];
    table2 = new S[m2];
    status1 = new schar[m];
    status2 = new schar[m2];

    int i=0;
    for (i=0 ; i<m ; i++)
    {
        status1[i] = empty;
    }
    for (i=0 ; i<m2 ; i++)
    {
        status2[i] = empty;
    }
}


template<class S> Hash_cuck<S>::~Hash_cuck()
{
    delete[] table1;
    delete[] table2;
    delete[] status1;
    delete[] status2;
}


template<class S> int Hash_cuck<S>::search(S s)
{
    int h = h1(s);
    schar stat = status1[h];
    if (stat==used && table1[h]==s)
        return 1;
    if (stat==empty)
        return -1;
    if (status2[h=h2(s)]==used && table2[h]==s)
        return 2;
    return -2;
}

template<class S> int Hash_cuck<S>::insert(S s)
{
    S p, q=s; // element q will be inserted in table 1
    uint h1_q = h1(q);
    uint h2_q;
    uint h2_p;

    if (status1[h1_q] == used)
    {
        if (table1[h1_q] == q)
            return 1;
        if (status2[h2_q=h2(q)] != used)
        {
            table2[h2_q] = q;
            status2[h2_q] = used;
               n++;
            return 2;
        }
        if (table2[h2_q] == q)
            return 2;
    }

    for (int i=1; i<=maxloop; i++)
    {
        p = table1[h1_q];     // element p will be inserted in table 2
        table1[h1_q] = q;
        if (status1[h1_q] != used)
        {   // q did not kick out an element
            status1[h1_q] = used;
            n++;
            if (i==1)
                return 1;
            return 2*i;
        }
        h2_p = h2(p);
        q = table2[h2_p];
        table2[h2_p] = p;
        if (status2[h2_p] != used)
        {   // p did not kick out an element
            status2[h2_p] = used;
            n++;
            return 2*i+1;
        }
        h1_q = h1(q);
    }
    return -2*maxloop-1;    // rehash required, insertion not possile
}


template<class S> void Hash_cuck<S>::del(S s)
{
    uint h = h1(s);
    if (status1[h]==used && table1[h]==s)
    {
        status1[h] = deleted;
        n--;
        return;
    }
    if (status1[h]!=empty && status2[h=h2(s)]==used && table2[h]==s)
    {
        status2[h] = deleted;
        n--;
    }
}


template<class S> void Hash_cuck<S>::out(std::ostream* stream)
{
    int i, j;

    *stream << "*******************************************\n";
    if (m == m2)
        *stream << "n=" << n << "  m1=m2=" << m << "\n";
    else
        *stream << "n=" << n << "  m1=" << m << "  m2=" << m2 << "\n";

    *stream << "*** Tabelle 1: ***\n";
    for (i=0; i<m && i<100; i+=7)
    {
        for (j=i; j<i+7 && j<m; j++)
        {
            stream->width(12);
            *stream << j;
        }
        *stream << "\n";
        for (j=i; j<i+7 && j<m; j++)
        {
            stream->width(12);
            if (status1[j] != used)
                *stream << " ";
            else
                *stream << table1[j];
        }
        *stream << "\n";
        for (j=i; j<i+7 && j<m; j++)
        {
            stream->width(12);
            switch (status1[j])
            {
                case empty: *stream << "empty"; break;
                case used: *stream << "used"; break;
                case deleted: *stream << "deleted"; break;
            }
        }
        *stream << "\n\n";
    }

    *stream << "*** Tabelle 2: ***\n";
    for (i=0; i<m2 && i<100; i+=7)
    {
        for (j=i; j<i+7 && j<m2; j++)
        {
            stream->width(12);
            *stream << j;
        }
        *stream << "\n";
        for (j=i; j<i+7 && j<m; j++)
        {
            stream->width(12);
            if (status2[j] != used)
                *stream << " ";
            else
                *stream << table2[j];
        }
        *stream << "\n";
        for (j=i; j<i+7 && j<m2; j++)
        {
            stream->width(12);
            switch (status2[j])
            {
                case empty: *stream << "empty"; break;
                case used: *stream << "used"; break;
                case deleted: *stream << "deleted"; break;
            }
        }
        *stream << "\n\n";
    }
}


template<class S> void Hash_cuck<S>::occupancy(uint &n1)
{
    int i;
    n1 = 0;
    for (i=0 ; i<m ; i++)
    {
        if (status1[i] == used)
            n1++;
    }
}


///////////////////////////////////////////////////////
///////////////////////////////////////////////////////


Hash_cuck_pol::Hash_cuck_pol(int m, int maxloop, RandBase* random)
    : Hash_cuck<uint>(m, m, maxloop)
{
    a1 = random->get(1<<31);
    a2 = random->get(1<<31);
    b1 = random->get(1<<31);
    b2 = random->get(1<<31);
}


Hash_cuck_pol::Hash_cuck_pol(int m, int m2, int maxloop, RandBase* random)
    : Hash_cuck<uint>(m, m2, maxloop)
{
    a1 = random->get(1<<31);
    a2 = random->get(1<<31);
    b1 = random->get(1<<31);
    b2 = random->get(1<<31);
}


Hash_cuck_pol::Hash_cuck_pol(int m, int maxloop,
    uint a1, uint b1, uint a2, uint b2) : Hash_cuck<uint>(m, maxloop)
{
    Hash_cuck_pol::a1 = a1;
    Hash_cuck_pol::a2 = a2;
    Hash_cuck_pol::b1 = b1;
    Hash_cuck_pol::b2 = b2;
}


///////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////


Hash_cuck_cw::Hash_cuck_cw(int m, int maxloop, RandBase* random)
    : Hash_cuck<uint>(m, m, maxloop)
{
    for (int i=0 ; i<128 ; i++)
    {
        r1[i] = random->get(1<<31);
        r2[i] = random->get(1<<31);
    }
}


Hash_cuck_cw::Hash_cuck_cw(int m, int m2, int maxloop, RandBase* random)
    : Hash_cuck<uint>(m, m2, maxloop)
{
    for (int i=0 ; i<128 ; i++)
    {
        r1[i] = random->get(1<<31);
        r2[i] = random->get(1<<31);
    }
}


inline int Hash_cuck_cw::h1(uint s)
{
    int addr = -1;
    uint h = 0;
    for (uint i=0 ; i<8 ; i++)
    {
        h ^= r1[addr += 1 + (s & 0xF)];
        s >>= 4;
    }
    return h % m;
}


inline int Hash_cuck_cw::h2(uint s)
{
    int addr = -1;
    uint h = 0;
    for (uint i=0 ; i<8 ; i++)
    {
        h ^= r2[addr += 1 + (s  & 0xF)];
        s >>= 4;
    }
    return h % m2;
}


////////////////////////////////////////////////////////
////////////////////////////////////////////////////////


Hash_cuck_sim::Hash_cuck_sim(int m, int n_end, RandBase* random)
    : Hash_cuck<uint>(m, n_end+1)
{
    //static RandKISS random(m,0);
    //random.setn(m);
    Hash_cuck_sim::n_end = n_end;
    hv1 = new uint[n_end];
    hv2 = new uint[n_end];

    for (int i=0 ; i<n_end ; i++)
    {
        hv1[i] = random->get(m);
        hv2[i] = random->get(m);
    }
}


Hash_cuck_sim::Hash_cuck_sim(int m, int m2, int n_end, RandBase* random)
    : Hash_cuck<uint>(m, m2, n_end+1)
{
    //static RandKISS random(m,0);
    //random.setn(m);
    Hash_cuck_sim::n_end = n_end;
    hv1 = new uint[n_end];
    hv2 = new uint[n_end];

    for (int i=0 ; i<n_end ; i++)
    {
        hv1[i] = random->get(m);
        hv2[i] = random->get(m2);
    }
}


Hash_cuck_sim::~Hash_cuck_sim()
{
    delete[] hv1;
    delete[] hv2;
}


//////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////


template<class S> Hash_cuck_gen<S>::Hash_cuck_gen(
    int m, int maxloop, int (*h1)(S, int), int (*h2)(S, int))
    : Hash_cuck<S>(m, maxloop)
{
    ph1 = h1;
    ph2 = h2;
}
```

Listing A.4: cuckoo_main.cpp

```
////////////////////////////////////////////////////////////
// Simulation Programm for the Cuckoo Hashing Library
// by Reinhard Kutzelnigg
// bug reports and comments to kutzelnigg@dmg.tuwien.ac.at
// www.dmg.tuwien.ac.at/kutzelnigg
// last updated: $Date: 2008/09/23 14:45:29 $
// distributed under the conditions of the GPL
////////////////////////////////////////////////////////////

// This program requests a file "input.txt", consisting of lines
// : repetitions numthread
// . type percentage rerun search
// m n
// m... total memory (both tables)
// percentage... size of the first table m1=m*perc/100, m2=m-m1
// n... number of (random) elements to insert
// repetitions... number of experiments using this input
//                (here we count only successful experiments)
// type ... 0 for Simulation with random numbers, 1 for pol. hash functions,
//          2 for Carter - Wegman style hash functions


#include "mt.hpp"
#include "random.hpp"
#include "base.hpp"
#include "cuckoo.hpp"
#include "threadparam.hpp"
#include <iostream>
#include <sstream>
#include <fstream>
#include <time.h>
#include <gmp.h>
#include <pthread.h>
#include <sys/time.h>



#define MAX_THREADS 8
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;


void *simulate(void* params);



uint m;
uint m2;
uint n;
int type = 0;           // type of hash function
uint numthreads = 1;
bool search = false;    // determine search time
bool rerun = false;     // repeat unsuccessful construction


int main()
{
    uint perc;
    int rep;
    int i;

    std::ofstream outfile;
    std::ifstream infile ("input.txt");
    if (!infile.good())
        return 1;

    time_t rawtime;
    struct tm * timeinfo;

    outfile.open("cuckoo.txt", std::ofstream::app);
    time ( &rawtime );
    timeinfo = localtime ( &rawtime );
    outfile << "Current date and time are: " << asctime (timeinfo) << "\n";
    outfile.close();
        timeval start_tval, end_tval;

    while (true)
    {
        gettimeofday(&start_tval, 0);

        char buff[100];
        infile.getline(buff,100);
        if (infile.eof())
            break;
        std::istringstream ist(buff);

        switch(buff[0])
        {
            case '%':
            case '\r':
            case  0 : continue;
            case ':': ist.ignore(1);
                       ist >> rep;
                       ist >> numthreads;
                       if (numthreads > MAX_THREADS)
                              numthreads = MAX_THREADS;
                       continue;
            case '.': ist.ignore(1);
                       ist >> type;
                       ist >> perc;
                       ist >> rerun;
                       ist >> search;
                       continue;
        }
        ist >> m;
        ist >> n;
        m2 = m - (m*perc)/100;
        m = (m*perc)/100;

        if (ist.bad())
            break;

        if (m==m2)
            std::cout << "\nm1=m2: " << m ;
        else
            std::cout << "\nm1: " << m << "  m2: " << m2;
        std::cout << "  n: " << n;
        switch (type)
        {
            case 0: std::cout << "  simulation\n"; break;
            case 1: std::cout << "  polynomial\n"; break;
            case 2: std::cout << "  Carter-Wegman\n"; break;
            default: std::cout << "  error\n"; break;
        }

        int error = 0;
        double search_steps_d = 0;
        double total_steps_d = 0;
        double error_steps_d = 0;
        double max_steps_d = 0;
        int max_global = 0;

        pthread_t tid[numthreads];
    Threadparam_c param[numthreads];

    int rc;
    int thread_rep = rep / numthreads;

    for (i = 0; i < numthreads; ++i)
    {
        if (i == numthreads)
            thread_rep = rep - (numthreads * thread_rep);
        param[i] = Threadparam_c(i, thread_rep, &error, &search_steps_d,
                                 &total_steps_d, &error_steps_d,
                                 &max_steps_d, &max_global);

        rc = pthread_create(&tid[i], NULL, simulate, &param[i]);
        if (rc != 0)
        {
            std::cout << "Could␣not␣create␣thread.␣Abort.\n";
            return 1;
        }
    }

    for (i = 0; i < numthreads; ++i)
    {
        rc = pthread_join(tid[i], NULL);
        if (rc != 0)
        {
             std::cout << "Could␣not␣join␣threads.␣Abort.\n";
             return 1;
         }
    }

    time ( &rawtime );
    timeinfo = localtime ( &rawtime );
    gettimeofday(&end_tval, 0);
    unsigned long runtime_sec = end_tval.tv_sec-start_tval.tv_sec;
    long runtime_usec = end_tval.tv_usec-start_tval.tv_usec;
    if (runtime_usec < 0)
    {
            runtime_usec += 1000000;
            runtime_sec -= 1;
    }

    outfile.open("cuckoo.txt", std::ofstream::app);
    if (m==m2)
        outfile << "␣m1=m2:␣" << m ;
    else
        outfile << "␣m1:␣" << m << "␣␣m2:␣" << m2;
    outfile << "␣␣n:␣" << n;
    switch (type)
    {
        case 0: outfile << "␣␣simulation\n"; break;
        case 1: outfile << "␣␣polynomial\n"; break;
        case 2: outfile << "␣␣Carter-Wegman\n"; break;
        default: outfile << "␣␣error\n"; break;
    }
    uint rep_performed;
    if (rerun)
        rep_performed = rep + error;
    else
        rep_performed = rep;

    outfile << "Number␣of␣errors:␣" << error << "␣during␣"
            << rep_performed << "␣experiments.\n"
            << "Average␣number␣of␣steps␣per␣insertion␣"
            << "(only␣\"good\"␣graphs):␣"
            << total_steps_d/(n*(double)(rep_performed-error)) << "\n"
```

```
                << "Average number of steps per insertion (all): "
                << error_steps_d/(n*(double)(rep_performed-error)) << "\n"
                << "Average max number of steps per insertion: "
                << max_steps_d/(double)(rep_performed-error) << "\n"
                << "Total max number of steps of an insertion: "
                << max_global << "\n";
        if (search)
            outfile << "Average number of steps per successful search: "
                    << search_steps_d/(n*(double)(rep_performed-error)) << "\n";
        outfile << "runtime in seconds: " << runtime_sec
                << "." << runtime_usec << "\n";
        outfile << "Current date and time are: " << asctime (timeinfo) << "\n";
        outfile.close();
    }
    std::cout << "\n";

    pthread_mutex_destroy(&mutex);
    return 0;
}



void* simulate(void* params)
{
    Threadparam_c* p = (Threadparam_c*)params;
    int rep = p->rep;

    Hash_cuck<uint> *table;
    int i, j;

    int error_local = 0;
    mpz_t total_steps;
    mpz_init (total_steps);
    mpz_t error_steps;
    mpz_init (error_steps);
    mpz_t search_steps;
    mpz_init (search_steps);
    mpz_t max_steps_sum;
    mpz_init (max_steps_sum);
    int max_thread = 0;

    time_t ltime;
    time (& ltime);
    int init = ltime * 1234567 *(1+p->id);
    RandKISS random(init);
    //MTRand random(init);

    for (i=0 ; i<rep ; i++)
    {
        unsigned int current_steps = 0;
        int max_steps = 0;

        switch (type)
        {
        case 0:  // simulation, (pseudo) random hash functions
                 table = new Hash_cuck_sim (m, m2, n, &random); break;
        case 1:  // polynomial hash functions
                 table = new Hash_cuck_pol (m, m2, n+1, &random); break;
        default: // Carter-Wegman like hash functions
                 table = new Hash_cuck_cw (m, m2, n+1, &random);
        }

        int steps;
        for (j=0 ; j<n ; j++)
        {
            if (type == 0)
```

```
                steps = table->insert(j);
            else
            {
                int key;
                do
                {
                     key = random.get(1<<31);
                }    while (table->search(key) > 0);
                steps = table->insert(key);
            }
            if (steps <= 0)
            {
                    error_local++;
                    current_steps -= steps; // steps are negative
                    mpz_add_ui(error_steps,error_steps,current_steps);
                    if (rerun)
                        i--;     // repeat unsuccessful construction
                    break;  // construction not successful
            }
            current_steps += steps;
            if (steps > max_steps)
                max_steps = steps;
        }

        mpz_add_ui(max_steps_sum,max_steps_sum,max_steps);
        if (max_steps > max_thread)
            max_thread = max_steps;

        if (search && steps>0)
        {   // determine average succ. search cost
            uint n1;
            table->occupancy(n1);
            mpz_add_ui(search_steps,search_steps,2*n-n1);
        }

        // output progress information
        if (rep/10 > 0 && p->id == 0)
            if ((i+1) % (rep/10) == 0 )
                std::cout << ((i+1)*100) / rep
                    << "␣percent␣done␣by␣first␣thread.\n";

        mpz_add_ui(total_steps,total_steps,current_steps);
        /*std::ofstream outfile;
        outfile.open("cuckoo.txt", std::ofstream::app);
        table->out(&outfile);
        outfile.close();*/
        delete table;
    }
    mpz_add(error_steps,error_steps,total_steps);

    pthread_mutex_lock(&mutex);
    *(p->error) += error_local;
    *(p->total_steps_d) += mpz_get_d(total_steps);
    *(p->error_steps_d) += mpz_get_d(error_steps);
    *(p->search_steps_d) += mpz_get_d(search_steps);
    *(p->max_steps_d) += mpz_get_d(max_steps_sum);
    if (*(p->max_global) < max_thread)
        *(p->max_global) = max_thread;
    pthread_mutex_unlock(&mutex);

    mpz_clear(total_steps);
    mpz_clear(error_steps);
    mpz_clear(search_steps);
}
```

Listing A.5: d-cuckoo.hpp

```
/////////////////////////////////////////////////////////////
// d - Cuckoo Hashing Library
// by Reinhard Kutzelnigg
// bug reports and comments to kutzelnigg@dmg.tuwien.ac.at
// www.dmg.tuwien.ac.at/kutzelnigg
// last updated: $Date: 2008/06/29 07:24:18 $
// distributed under the conditions of the LGPL
/////////////////////////////////////////////////////////////

// References:
// Donald E. Knuth, The art of computer programming, volume III:
// Sorting and searching, second ed., Addison-Wesley, Boston, 1998.
// D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis, Space Efficient Hash
// Tables with Worst Case Constant Access Time, LNCS 2607, pp. 271-282, 2003.

#ifndef DCUCKOO_HASH_H___
#define DCUCKOO_HASH_H___

#include <iostream>
#include <sstream>
#include <fstream>
#include "base.hpp"       // Interface of my Hash library
#include "random.hpp"     // my implementation of a random number generator


// pure virtual base class for d - Cuckoo Hashing
template<class S> class Hash_d_cuck : public Hash_base<S>
{
protected:
    int m;            // size of whole table
    int n;            // number of actual inserted elements
    int d;            // d <=126
    bool separate;    // hash functions adress whole table or not
    int maxloop;      // max. number of loops for an insertion
    S *table;         // arry holding (all) table data
    virtual int h(S s, int nr) = 0;    //   hash function
private:
    int *status;      // arry holding (all) table status data
    uint *hv;         // buffer holding all hash positions for a fixed key
public:
    Hash_d_cuck(int m, int d, bool separate, int maxloop);
    ~Hash_d_cuck();
    int search(S);
    int insert(S);
    void del(S);
    int get_n() { return n; }
    void out(std::ostream*);
    uint* occupancy();
};


// Implementation for unsigned int keys
// hash functions of type (s*a+b) %m where
//          a,b: random numberes 0,1..u-1
//         prime number table size strongly recommended!
//         (calculations with 32 bit, truncation possible)
class Hash_d_cuck_pol : public Hash_d_cuck<uint>
{
private:
    int M;
    uint *a;
    uint *b;
    int h(uint s, int nr) { return (a[nr]*s+b[nr]) % M + separate*nr*M; }
public:
    Hash_d_cuck_pol(int m, int d, bool separate, int maxloop, RandBase* random);
    ~Hash_d_cuck_pol();
};


// Implementation for unsigned int keys
// Carter - Wegman hash functions
class Hash_d_cuck_cw : public Hash_d_cuck<uint>
{
private:
    int M;
    int **r;
    int h(uint s, int nr);
public:
    Hash_d_cuck_cw(int m, int d, bool separate, int maxloop, RandBase* random);
    ~Hash_d_cuck_cw();
};


// Implementation ONLY for SIMULATION!!!
// (used to acquire the succes rate of table constructions)
// uses unsigned int keys 1,2,3,...,n_max
// and tables containing random numbers as hash functions
class Hash_d_cuck_sim : public Hash_d_cuck<uint>
{
private:
    int n_end;
    uint **hv;
    int h(uint s, int nr) { return hv[nr][s%n_end]; }
public:
    Hash_d_cuck_sim(int m, int d, int n_end, bool seperate, RandBase* random);
    ~Hash_d_cuck_sim();
};


// The definition of the template class is needed by the compiler to create
// instances of the class.
// Until the C++ keyword "export" is supported, the only (recommended) way
// is to include this definition in the header.
#include "d-cuckoo.cpp"
#endif //DCUKCOO_HASH_H___
```

Listing A.6: d-cuckoo.cpp

```
/////////////////////////////////////////////////////////////
// d - Cuckoo Hashing Library
// by Reinhard Kutzelnigg
// bug reports and comments to kutzelnigg@dmg.tuwien.ac.at
// www.dmg.tuwien.ac.at/kutzelnigg
// last updated: $Date: 2008/06/29 07:24:18 $
// distributed under the conditions of the LGPL
/////////////////////////////////////////////////////////////

template<class S> Hash_d_cuck<S>::Hash_d_cuck
    (int m, int d, bool separate, int maxloop)
{
    Hash_d_cuck::m = m;
    Hash_d_cuck::maxloop = maxloop;
    Hash_d_cuck::separate = separate;
    if(d > 126)
        d = 126;
    Hash_d_cuck::d = d;
    n = 0;
    table = new S[m];
    status = new int[m];
    hv = new uint[d];

    for (int i=0 ; i<m ; i++)
    {
        status[i] = -1;
    }
}


template<class S> Hash_d_cuck<S>::~Hash_d_cuck()
{
    delete[] table;
    delete[] status;
    delete[] hv;
}


template<class S> int Hash_d_cuck<S>::search(S s)
{
    for (int i=0 ; i<d ; i++)
    {
        if (status[h(s,i)]>=0 && table[h(s,i)]==s)
            return i+1;
    }
    return -d;
}


template<class S> int Hash_d_cuck<S>::insert(S s)
{
    S p, q=s;
    int i, j;
    int nr_q;
    int nr_p;

    hv[0] = h(q,0);
    if (status[hv[0]] == -1)
    {    // first inspected cell is empty
        table[hv[0]] = q;
        status[hv[0]] = 0;
        n++;
        return 1;
    }

    for (j=1 ; j!=d ; j++)
    {
        hv[j] = h(q,j);
        if (status[hv[j]] == -1  ||  table[hv[j]]==q)
        {    // found an empty cell or q already in table
            table[hv[j]] = q;
            status[hv[j]] = j;
            n++;
            return 1+j;
        }
    }
    // all possible cells for q are occupied
    p = table[hv[0]];    // p will be kicked out
    nr_p = status[hv[0]];
    table[hv[0]] = q;
    status[hv[0]] = 0;
    q = p;
    nr_q = (nr_p+1) % d;

    for (i=1; i<maxloop; i++)
    {
        for (j=nr_q ; j!=nr_p ; j=(j+1)%d)
        {
            hv[j] = h(q,j);
            if (status[hv[j]] == -1  ||  table[hv[j]]==q)
            {    // found an empty cell or q already in table
                table[hv[j]] = q;
                status[hv[j]] = j;
                n++;
                int steps = j - nr_q + 1;
                if (steps < 0)
                    steps += d;
                  return i*(d-1)+1+steps;
            }
        }
        // all possible cells for q are occupied
        p = table[hv[nr_q]];    // p will be kicked out
        nr_p = status[hv[nr_q]];
        table[hv[nr_q]] = q;
        status[hv[nr_q]] = nr_q;
        q = p;
        nr_q = (nr_p+1) % d;
    }
    return -maxloop*(d-1)+1;
    // rehash required, insertion not possile
}


template<class S> void Hash_d_cuck<S>::del(S s)
{
    int r = search(s);
    if (r >= 0)
    {
        status[h(s,r)] = -1;
        n--;
    }
}


template<class S> void Hash_d_cuck<S>::out(std::ostream* stream)
{
    int i, j;

    *stream << "*******************************************\n";
    *stream << "n=" << n << "  m=" << m << "  d=" << d << "\n";
    for (i=0; i<m && i<100; i+=7)
    {
        for (j=i; j<i+7 && j<m; j++)
        {
            stream->width(12);
            *stream;
        }
        *stream << "\n";

        for (j=i; j<i+7 && j<m; j++)
        {
            stream->width(12);
            if (status[j] < 0)
                *stream << "␣";
            else
                *stream << table[j];
        }
        *stream << "\n";
        for (j=i; j<i+7 && j<m; j++)
        {
            stream->width(12);
            *stream << status[j];
        }
        *stream << "\n\n";
    }
}


template<class S> uint* Hash_d_cuck<S>::occupancy()
{
    uint i;
    int p;
    uint* o;
    o = new uint[d];
    for (i=0 ; i<d ; i++)
        o[i] = 0;
    for (i=0 ; i<m ; i++)
    {
        p = status[i];
        if (p >= 0)
            o[p]++;
    }
    return o;
}


////////////////////////////////////////////////////////
////////////////////////////////////////////////////////


Hash_d_cuck_pol::Hash_d_cuck_pol
    (int m, int d, bool separate, int maxloop, RandBase* random)
    : Hash_d_cuck<uint>(m, d, separate, maxloop)
{
    a = new uint[d];
    b = new uint[d];

    if (separate)
        M = m/d;
    else
        M = m;

    for (int i=0 ; i<d ; i++)
    {
        a[i] = random->get(1<<31);
        b[i] = random->get(1<<31);
    }
}

Hash_d_cuck_pol::~Hash_d_cuck_pol()
{
    delete[] a;
    delete[] b;
}


///////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////


Hash_d_cuck_cw::Hash_d_cuck_cw
    (int m, int d, bool separate, int maxloop, RandBase* random)
    : Hash_d_cuck<uint>(m, d, separate, maxloop)
{
    if (separate)
            M = m/d;
        else
            M = m;
    r = new int*[d];
    for (int j=0 ; j<d ; j++)
    {
        r[j] = new int[128];
        for (int i=0 ; i<128 ; i++)
        {
            r[j][i] = random->get(1<<31);
        }
    }
}


Hash_d_cuck_cw::~Hash_d_cuck_cw()
{
    for (int j=0 ; j<d ; j++)
    {
        delete[] r[j];
    }
    delete[] r;
}


inline int Hash_d_cuck_cw::h(uint s, int nr)
{
    int addr = -1;
    uint h = 0;
    for (uint i=0 ; i<8 ; i++)
    {
        h ^= r[nr][addr += 1 + (s & 0xF)];
        s >>= 4;
    }
    return h % M + separate*nr*M;;
}


///////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////


Hash_d_cuck_sim::Hash_d_cuck_sim
    (int m, int d, int n_end, bool separate, RandBase* random)
    : Hash_d_cuck<uint>(separate ? d*(m/d) : m, d, separate, d*n_end+1)
{
    Hash_d_cuck_sim::n_end = n_end;
    hv = new uint*[d];

    if(!separate)
    {
        for (int j=0 ; j<d ; j++)
        {
            hv[j] = new uint[n_end];
            for (int i=0 ; i<n_end ; i++)
            {
                hv[j][i] = random->get(m);
            }
        }
    }
    else
    {
        for (int j=0 ; j<d ; j++)
        {
            hv[j] = new uint[n_end];
            for (int i=0 ; i<n_end ; i++)
            {
                hv[j][i] = random->get(m/d) + j*(m/d);
            }
        }
    }
}


Hash_d_cuck_sim::~Hash_d_cuck_sim()
{
    for (int j=0 ; j<d ; j++)
    {
        delete[] hv[j];
    }
    delete[] hv;
}
```

Listing A.7: tables.hpp

```
////////////////////////////////////////////////////////////
// (Bipartite) Disjoint Set-Forest Library
// by Reinhard Kutzelnigg
// bug reports and comments to kutzelnigg@dmg.tuwien.ac.at
// www.dmg.tuwien.ac.at/kutzelnigg
// last updated: $Date: 2007/09/24 07:57:51 $
// distributed under the conditions of the LGPL
////////////////////////////////////////////////////////////

// References:
// T. C. Cormen, C. E. Leiserson, and R. L. Rivest,
// Introduction to Algorithms, MIT Press, London, 2000.

#ifndef TABLES_HPP___
#define TABLES_HPP___

#include <malloc.h>

class Tables
{
    bool bipartite;
    int *set1;
    int *size1;
    bool *cyclic1;
    int *set2;
    int *size2;
    bool *cyclic2;

public:
    Tables (int m, int m2=0);
    ~Tables ();

    // returns the parent of u
    int parent (int u);

    // unions the components with roots u and v (new root v)
    void merge (int u, int v);

    // returns the size of the component containing u, only for root nodes!
    int getsize (int u);

    // sets the size of the component with root u to s
    void setsize (int u, int s);

    // returns if u is in a cycle, use only for root nodes!
    int getcyclic (int u);

    // marks the component with root u as cyclic
    void setcyclic (int u);

    // returns the root node of u
    int findset (int u);

    void* operator new(size_t s) { return malloc(s); }
    void operator delete(void* p) { free(p); }
};

#endif /*TABLES_HPP___*/
```

Listing A.8: tables.cpp

```
/////////////////////////////////////////////////////////////
// Bipartite Disjoint Set-Forest Library
// by Reinhard Kutzelnigg
// bug reports and comments to kutzelnigg@dmg.tuwien.ac.at
// www.dmg.tuwien.ac.at/kutzelnigg
// last updated: $Date: 2008/07/07 14:57:49 $
// distributed under the conditions of the LGPL
/////////////////////////////////////////////////////////////

#include "tables.hpp"

Tables::Tables (int m, int m2)
{
    if (m2 <= 0)
        bipartite = false;
    else
        bipartite = true;

    set1 = new int[m+1];
    size1 = new int[m+1];
    cyclic1 = new bool[m+1];

    for (int j=1 ; j<m+1 ; j++)
    {
        set1[j] = j;
        size1[j] = 1;
        cyclic1[j] = false;
    }

    if(bipartite)
    {
        set2 = new int[m2+1];
        size2 = new int[m2+1];
        cyclic2 = new bool[m2+1];

        for (int j=1 ; j<m2+1 ; j++)
        {
            set2[j] = -j;
            size2[j] = 1;
            cyclic2[j] = false;
        }
    }
}

Tables::~Tables ()
{
    delete[] set1;
    delete[] size1;
    delete[] cyclic1;

    if (bipartite)
    {
        delete[] set2;
        delete[] size2;
        delete[] cyclic2;
    }
}


int Tables::parent (int u)
{
    if (u > 0)
    {   // first table
        return set1[u];
    }
    //second table
    return set2[-u];
}

void Tables::merge (int u, int v)
{
    if (u > 0)
    {    // first table
        set1[u] = v;
        return;
    }
    // second table
    set2[-u] = v;
}


int Tables::getsize (int u)
{
    if (u>0)
    {   // first table
        return size1[u];
    }
    // second table
    return size2[-u];
}


void Tables::setsize (int u, int s)
{
    int h;

    if (u>0)
    {   // first table
        h = size1[u];
        size1[u] = s;
        return;
    }
    // second table
    h = size2[-u];
    size2[-u] = s;
}


int Tables::getcyclic (int u)
{
    if (u>0)
    {   // first table
        return cyclic1[u];
    }
    return cyclic2[-u];
}


void Tables::setcyclic (int u)
{
    int h;

    if (u>0)
    {   // first table
        h = cyclic1[u];
        cyclic1[u] = true;
        return;
    }
    // second table
    h = cyclic2[-u];
    cyclic2[-u] = true;
}


int Tables::findset (int u)
{    // standard version
    while (parent(u) != u)
        u = parent(u);
    return u;
}

/*
int Tables::findset (int u)
{    // shorten path length
    int v = parent(u);
    int w = u;
    if (v != u)
    {
        w = findset(v);
        merge(u,w);
    }
    return w;
}
*/
```

Appendix B

Selected Maple worksheets

In this chapter, we present some of the most important Maple worksheets used to perform the calculations decribed in the previous chapters. All files (and some others) are included on the attached CD-ROM.

In general, each worksheet starts with the derivation of the saddle point theorem that is required to obtain the desired results. After this initial calculation, the actual derivation of special results start. At this point, is should be easy to exchange the involved functions and hence adopt the worksheet to perform different calculations that are however based on the same saddle point method.

```
> # Saddle Point Method, general case
  # We calculate the leading term and the coefficients of
```
$\frac{1}{m}$ **and** $\frac{1}{m^2}$ *of the asymptotic expansion of* $[x^{2m}]f(x)^{2m-n}g(x)$

```
  restart:
> with(LinearAlgebra) :
  with(plots) :
  with(stats) :
```

substitution

```
> # cummulants
```
$h3 := -\frac{I\cdot t^3}{6}\cdot k(3)\cdot x :$

$h4 := \frac{t^4}{24}\cdot k(4)\cdot x^2 :$

$h5 := \frac{I\cdot t^5}{5!}\cdot k(5)\cdot x^3 :$

$h6 := -\frac{t^6}{6!}\cdot k(6)\cdot x^4 :$

$> hg1 := I\cdot t\cdot kg(1)\cdot x :$

$hg2 := -\frac{t^2}{2}\cdot kg(2)\cdot x^2 :$

$hg3 := -\frac{I\cdot t^3}{3!}\cdot kg(3)\cdot x^3 :$

$hg4 := \frac{t^4}{4!}\cdot kg(4)\cdot x^4 :$

> # coefficient of $\frac{1}{2m-n}$
```
  series(exp(h3 + h4 + hg1 + hg2), x, 3) :
  convert(%, polynom) :
  h := collect( coeff( %, x^2), t) :
```

> # coefficient of $\frac{1}{(2m-n)^2}$
```
  series(exp(h3 + h4 + h5 + h6 + hg1 + hg2 + hg3 + hg4), x, 5) :
  convert(%, polynom) :
  h_ := collect( coeff( %, x^4), t) :
> # substitution
```
$subs\left(t = \frac{u}{\sqrt{k(2)}}, h\right) :$
```
  h := collect(%, u);
```
$$h := -\frac{1}{72}\frac{u^6 k(3)^2}{k(2)^3} + \frac{\left(\frac{1}{6}k(3)\,kg(1) + \frac{1}{24}k(4)\right)u^4}{k(2)^2} \tag{1.1}$$

$$+ \frac{\left(-\frac{1}{2}kg(1)^2 - \frac{1}{2}kg(2)\right)u^2}{k(2)}$$

$> subs\left(t = \frac{u}{\sqrt{k(2)}}, h\_\right) :$
```
  h_ := collect(%, u);
```
$$\begin{aligned} h\_ := {} & \frac{1}{31104}\frac{u^{12}k(3)^4}{k(2)^6} + \frac{\left(-\frac{1}{1728}k(3)^2k(4) - \frac{1}{1296}k(3)^3kg(1)\right)u^{10}}{k(2)^5} \\ & + \frac{1}{k(2)^4}\left(\left(\frac{1}{144}k(3)^2kg(2) + \frac{1}{720}k(3)\,k(5)\right.\right. \\ & \left.\left. + \frac{1}{144}k(3)\,kg(1)\,k(4) + \frac{1}{144}k(3)^2kg(1)^2 + \frac{1}{1152}k(4)^2\right)u^8\right) \\ & + \frac{1}{k(2)^3}\left(\left(-\frac{1}{720}k(6) - \frac{1}{36}k(3)\,kg(3) - \frac{1}{48}k(4)\,kg(2)\right.\right. \\ & - \frac{1}{36}k(3)\,kg(1)^3 - \frac{1}{48}kg(1)^2k(4) - \frac{1}{12}k(3)\,kg(1)\,kg(2) \\ & \left.\left. - \frac{1}{120}kg(1)\,k(5)\right)u^6\right) + \frac{1}{k(2)^2}\left(\left(\frac{1}{6}kg(1)\,kg(3) + \frac{1}{8}kg(2)^2\right.\right. \\ & \left.\left. + \frac{1}{24}kg(4) + \frac{1}{4}kg(1)^2kg(2) + \frac{1}{24}kg(1)^4\right)u^4\right) \end{aligned} \tag{1.2}$$

integration

$> int\left(\exp\left(-\frac{t^2}{2}\right)\cdot t\char`^n,\ t = -\infty..\infty\right) :$
```
  i := unapply(%, n);
```
$$i := n \rightarrow \Gamma\left(\frac{1}{2}n + \frac{1}{2}\right)2^{-\frac{1}{2}+\frac{1}{2}n}\left(1 + (-1)^n\right) \tag{2.1}$$

```
> # calculate integrals
  H := 0 :
  for a from 0 to 6 do:
    if a > 0 then
      coeff(h, u^a)
    else
      subs(u = 0, h)
    fi:
    H := H + %* i(a) :
  od:
> H_ := 0 :
  for a from 0 to 12 do:
    if a > 0 then
      coeff(h_, u^a)
    else
```

$subs(u = 0, h\_)$
fi:
$H\_ := H\_ + \%{*}\,i(a)$:
od:

output expansion

> $\#$ *coefficient of* $\frac{1}{2m-n}$
> $simplify\left(\frac{H}{i(0)}\right)$;

$$-\frac{1}{24}\,\frac{1}{k(2)^3}\big(12\,k(2)^2\,kg(1)^2 + 12\,k(2)^2\,kg(2) - 12\,k(2)\,k(3)\,kg(1) - 3\,k(2)\,k(4) + 5\,k(3)^2\big) \tag{3.1}$$

> $\#$ *coefficient of* $\frac{1}{(2m-n)^2}$
> $simplify\left(\frac{H\_}{i(0)}\right)$;

$$\begin{aligned}\frac{1}{1152}\,\frac{1}{k(2)^6}\big(&576\,k(2)^4\,kg(1)\,kg(3) + 432\,k(2)^4\,kg(2)^2 + 144\,k(2)^4\,kg(4)\\&+ 864\,k(2)^4\,kg(1)^2\,kg(2) + 144\,k(2)^4\,kg(1)^4 - 24\,k(2)^3\,k(6)\\&- 480\,k(2)^3\,k(3)\,kg(3) - 360\,k(2)^3\,k(4)\,kg(2) - 480\,k(2)^3\,k(3)\,kg(1)^3\\&- 360\,k(2)^3\,kg(1)^2\,k(4) - 1440\,k(2)^3\,k(3)\,kg(1)\,kg(2)\\&- 144\,k(2)^3\,kg(1)\,k(5) + 840\,k(2)^2\,k(3)^2\,kg(2) + 168\,k(2)^2\,k(3)\,k(5)\\&+ 840\,k(2)^2\,k(3)\,kg(1)\,k(4) + 840\,k(2)^2\,k(3)^2\,kg(1)^2\\&+ 105\,k(2)^2\,k(4)^2 - 630\,k(3)^2\,k(2)\,k(4) - 840\,k(3)^3\,k(2)\,kg(1)\\&+ 385\,k(3)^4\big)\end{aligned} \tag{3.2}$$

definition of funtions f and g, calculation of cummulants

> $f := \ln(ts(e^u))$;
> $g := \ln\left(\frac{1}{\sqrt{1-t(e^u)}}\right)$;
> $\#$ *saddle point,* $e = 1 - \frac{n}{m}$
> $x0 := (1-e)\cdot e^{e-1}$;

$$f := \ln(ts(e^u))$$

$$g := -\frac{1}{2}\ln(1 - t(e^u))$$

$$x0 := (1-e)\,e^{e-1} \tag{4.1}$$

> $\#$ *main procedure to calculate cummulants*
> $k\_ :=$**proc**(s, f)
> **local** fl, i:
>
> $fl := f$:
>
> **for** i **from** 1 **to** s **do**
> $fl := diff(fl, u)$:
> $fl := subs\left(\mathrm{D}[1](ts)(e^u) = \frac{t(e^u)}{e^u}, fl\right)$:
> $fl := subs\left(\mathrm{D}[1](t)(e^u) = \frac{t(e^u)}{(1-t(e^u))\cdot e^u}, fl\right)$:
> **od**:
>
> $fl := subs\left(ts(e^u) = \frac{1-e^2}{2}, fl\right)$:
> $fl := subs(t(e^u) = 1-e, fl)$:
> $fl := subs(e^u = (1-e)\cdot e^{e-1}, fl)$:
> $fl := simplify(fl)$:
>
> **return** fl;
> **end proc**:

> $k :=$**proc**(s)
> **global** f, w:
> **return** $k\_(s, f)$:
> **end proc**:

> $kg :=$**proc**(s)
> **global** g, w:
> **return** $k\_(s, g)$:
> **end proc**:

> $k(1)$;
> $k(2)$;
> $k(3)$;
> $k(4)$;

$$\frac{2}{e+1}$$

$$-\frac{2\,(e-1)}{(e+1)^2\,e}$$

$$\frac{2\,(2\,e^3 - 5\,e^2 + 2\,e + 1)}{(e+1)^3\,e^3}$$

$$-\frac{2\,(6\,e^5 - 26\,e^4 + 25\,e^3 + 5\,e^2 - 7\,e - 3)}{(e+1)^4\,e^5} \tag{4.2}$$

> $kg(1)$;
> $kg(2)$;

$$-\frac{1}{2}\,\frac{e-1}{e^2}$$

$$\frac{1}{2}\,\frac{(e-1)\,(e-2)}{e^4} \qquad (4.3)$$

▼ asymptotic (main result)

> \# *leading term*
> $simplify\left(\frac{i(0)\cdot\exp(k(0))^{m\cdot(1+e)}\cdot\exp(kg(0))}{2\cdot\pi\cdot x0^{2\cdot m}\cdot\sqrt{m\cdot(1+e)\cdot k(2)}}\right);$

$$\frac{1}{2}\,\frac{\left(\frac{1}{2}-\frac{1}{2}e^2\right)^{m(e+1)}\left(-(e-1)\,e^{e-1}\right)^{-2m}}{\sqrt{\pi}\,\sqrt{e}\,\sqrt{-\frac{m\,(e-1)}{(e+1)\,e}}} \qquad (5.1)$$

> \# $+\frac{H}{m\cdot(1+e)}$
> $H := simplify\left(\frac{H}{i(0)}\right);$

$$H := -\frac{1}{48}\,\frac{e^4-12e^3-4e^2+12e-5}{e^3\,(e-1)} \qquad (5.2)$$

> \# $+\frac{H\_}{m^2\cdot(1+e)^2}$
> $H\_ := simplify\left(\frac{H\_}{i(0)}\right);$

$$H\_ := \frac{1}{4608}\,\frac{1}{e^6\,(e-1)^2}\,(e^8+264e^7+712e^6-2760e^5+6e^4+4920e^3 -2120e^2-2424e+1465) \qquad (5.3)$$

▼ additional calculations

> \# *asymptotic expansion of* $\frac{(2m)!n!}{(2m-n)!}$ *using Stirling's formula*
> $\left(1+\frac{1}{12\cdot 2}\cdot x+\frac{1}{288\cdot 4}\cdot x^2\right)\cdot\left(1+\frac{1}{12\cdot(1-e)}\cdot x+\frac{1}{288\cdot(1-e)}\cdot x^2\right):$
> $\left(1-\frac{1}{12\cdot(1+e)}\cdot x+\frac{1}{288\cdot(1+e)^2}\cdot x^2\right)\cdot\left(1+\frac{H}{1+e}\cdot x+\frac{H\_}{(1+e)^2}\cdot x^2\right):$
> $simplify(expand(\%\%\cdot\%)):$
> $factor(simplify(coeff(\%, x)));$
> $factor(simplify(coeff(\%\%, x^2)));$
> $f := unapply(\%\%, e):$
> $g := unapply(\%\%, e):$

$$\frac{1}{48}\,\frac{(2e-5)\,(e-1)^2}{e^3}$$

$$\frac{1}{4608}\,\frac{1}{e^6\,(e-1)^2}\,(4e^8-60e^7+205e^6-242e^5+835e^4-3992e^3 +7123e^2-5354e+1465) \qquad (6.1)$$

> \# *expected number of failures during the construction of N cuckoo hash tables,*
> \# *a is based on the first order expansion, a2 makes use of the second order term too;*
> $N := 5\cdot 10^5:$
> $a := (m, e)\rightarrow piecewise\left(-evalf\left(\frac{f(e)}{m}\right)\cdot N < N, -evalf\left(\frac{f(e)}{m}\right)\cdot N, N\right):$
> $a2 := (m, e)\rightarrow piecewise\left(-evalf\left(\frac{f(e)}{m}+\frac{g(e)}{m^2}\right)\cdot N < 0, 0, -evalf\left(\frac{f(e)}{m}+\frac{g(e)}{m^2}\right)\cdot N < N, -evalf\left(\frac{f(e)}{m}+\frac{g(e)}{m^2}\right)\cdot N, N\right):$

▶ probabilities

> *# Double Saddle Point Method, general case*
> *# We calculate the leading term and the coefficients of*
> $\frac{1}{m}$ **and** $\frac{1}{m^2}$ *of the asymptotic expansion of* $[x^m y^m] f(x,y)^{2m-n} g(x,y)$
> *restart:*

> *with(LinearAlgebra) :*
> *with(plots) :*
> *with(stats) :*

calculations for variable – substitution

> *# We perform a variable substitution*
> *to eleminate the term "st" of the function* $-\frac{1}{2} s^2 k(0,2) - t\,s\,k(1,1) - \frac{1}{2} t^2 k(2,0)$

$N := \langle\langle a \mid b\rangle, \langle c \mid d\rangle\rangle$:
N^%T . N;
$P := \langle\langle k(2,0) \mid k(1,1)\rangle, \langle k(1,1) \mid k(0,2)\rangle\rangle$;

$$\begin{bmatrix} a^2+c^2 & a\,b+c\,d \\ a\,b+c\,d & b^2+d^2 \end{bmatrix}$$

$$P := \begin{bmatrix} k(2,0) & k(1,1) \\ k(1,1) & k(0,2) \end{bmatrix} \qquad (1.1)$$

> $N := \langle\langle$sqrt$(k(2,0) - k(1,1)$^2 / $k(0,2)) \mid 0\rangle,$
> $\langle k(1,1)$ / sqrt$(k(0,2)) \mid$ sqrt$(k(0,2))\rangle\rangle$;
> N^%T . N;
> *Determinant*(N);

$$N := \begin{bmatrix} \sqrt{k(2,0) - \frac{k(1,1)^2}{k(0,2)}} & 0 \\ \frac{k(1,1)}{\sqrt{k(0,2)}} & \sqrt{k(0,2)} \end{bmatrix}$$

$$\begin{bmatrix} k(2,0) & k(1,1) \\ k(1,1) & k(0,2) \end{bmatrix}$$

$$\sqrt{k(2,0) - \frac{k(1,1)^2}{k(0,2)}}\,\sqrt{k(0,2)} \qquad (1.2)$$

> $sub :=$ *MatrixInverse*(N) . $\langle\langle u\rangle, \langle v\rangle\rangle$;

(1.3)

$$sub := \begin{bmatrix} \frac{u}{\sqrt{\frac{k(2,0)\,k(0,2) - k(1,1)^2}{k(0,2)}}} \\ -\frac{k(1,1)\,u}{k(0,2)\sqrt{k(2,0) - \frac{k(1,1)^2}{k(0,2)}}} + \frac{v}{\sqrt{k(0,2)}} \end{bmatrix} \qquad (1.3)$$

> *# perform test*
> $S := 0$:
> **for** i **from** 0 **to** 2 **do**:
> $j := 2 - i$:
> $S := S + m \cdot \left(\frac{I \cdot t}{\sqrt{m}}\right)^i \cdot \left(\frac{I \cdot s}{\sqrt{m}}\right)^j \cdot \frac{k(i,j)}{i! \cdot j!}$;
> **od**:
> S;
> *subs*$(t = sub[1,1], S)$:
> *subs*$(s = sub[2,1], \%)$:
> *simplify*$(\%)$;

$$-\frac{1}{2} s^2 k(0,2) - t\,s\,k(1,1) - \frac{1}{2} t^2 k(2,0)$$

$$-\frac{1}{2} v^2 - \frac{1}{2} u^2 \qquad (1.4)$$

substitution

> *# expansion of* $(2m-n) \cdot \log(f)$*, calculate terms involving* $\frac{1}{\sqrt{2m-n}^{\,l}}$
>
> $hf :=$ **proc**(l)
> **local** i, j, S: $S := 0$:
> **for** i **from** 0 **to** l **do**:
> $j := l - i$:
> $S := S + \frac{1}{x^2} \cdot (I \cdot t \cdot x)^i \cdot (I \cdot s \cdot x)^j \cdot \frac{k(i,j)}{i! \cdot j!}$:
> **od**:
> **return** *collect*(*simplify*$(S), x)$:
> **end proc**:

> *# expansion of* $\log(g)$*, calculate terms involving* $\frac{1}{\sqrt{2m-n}^{\,l}}$
>
> $hg :=$ **proc**(l)
> **local** i, j, S: $S := 0$:
> **for** i **from** 0 **to** l **do**:
> $j := l - i$:
> $S := S + (I \cdot t \cdot x)^i \cdot (I \cdot s \cdot x)^j \cdot \frac{kg(i,j)}{i! \cdot j!}$:
> **od**:
> **return** *collect*(*simplify*$(S), x)$:

```
  end proc:
> series(exp(hf(3) + hf(4) + hg(1) + hg(2)), x, 3) :
  convert(%, polynom) :
  h := collect(coeff(%, x^2), [s, t], distributed) :
> series(exp(hf(3) + hf(4) + hf(5) + hf(6) + hg(1) + hg(2) + hg(3)
      + hg(4)), x, 5) :
  convert(%, polynom) :
  h_ := collect(coeff(%, x^4), [s, t], distributed) :
> subs(t = sub[1, 1], h) :
  subs(s = sub[2, 1], %) :
  h := collect(%, [u, v], distributed) :
> subs(t = sub[1, 1], h_) :
  subs(s = sub[2, 1], %) :
  h_ := collect(%, [u, v], distributed) :
```

▼ integration

```
> int(int(e^(-1/2·(t^2 + s^2))·t^n·s^m, t = -∞..∞), s = -∞..∞) :
  i := unapply(%, n, m);
```

$$i := (n, m) \to \Gamma\left(\frac{1}{2}n + \frac{1}{2}\right)\Gamma\left(\frac{1}{2}m + \frac{1}{2}\right)2^{-1+\frac{1}{2}n+\frac{1}{2}m}\left(1 + (-1)^n + (-1)^m + (-1)^{n+m}\right) \quad (3.1)$$

```
> # calculate integrals
  H := 0 :
  for a from 0 to 6 do:
   for b from 0 to 6 do:
    if a > 0 then
     coeff(h, u^a)
    else
     subs(u = 0, h)
    fi:
    if b > 0 then
     coeff(%, v^b)
    else
     subs(v = 0, %)
    fi:
    H := H + %·i(a, b) :
   od:
  od:
> H_ := 0 :
  for a from 0 to 12 do:
   for b from 0 to 12 do:
    if a > 0 then
     coeff(h_, u^a)
    else
     subs(u = 0, h_)
    fi:
    if b > 0 then
     coeff(%, v^b)
    else
     subs(v = 0, %)
    fi:
    H_ := H_ + %·i(a, b) :
   od:
  od:
```

▶ additional calculations (α,β,γ,...)

▼ definition of funtions f and g, calculation of cummulants

```
> f := ln(ts(e^u, e^v));
  g := ln(1 / sqrt(1 - t1(e^u, e^v)·t2(e^u, e^v)));
  # saddle point,  e = 1 - n/m
  x0 := (1 - e)·e^(e-1);
  y0 := (1 - e)·e^(e-1);
```

$$f := \ln(ts(e^u, e^v))$$

$$g := -\frac{1}{2}\ln(1 - t1(e^u, e^v)\, t2(e^u, e^v))$$

$$x0 := (1 - e)\, e^{e-1}$$

$$y0 := (1 - e)\, e^{e-1} \quad (5.1)$$

```
> # main procedure to calculate cummulants
  k_ :=proc(r, s, f)
   local fl, i, j :

   fl := f :

   for i from 1 to s do
    fl := diff(fl, v) :
    fl := subs(D[2](ts)(e^u, e^v) = t2(e^u, e^v)/e^v, fl) :
    fl := subs(D[2](t1)(e^u, e^v) = t1(e^u, e^v)·t2(e^u, e^v)/(e^v·(1 - t1(e^u, e^v)·t2(e^u, e^v))), fl) :
    fl := subs(D[2](t2)(e^u, e^v) = t2(e^u, e^v)/(e^v·(1 - t1(e^u, e^v)·t2(e^u, e^v))), fl) :
   od:

   for j from 1 to r do
    fl := diff(fl, u) :
    fl := subs(D[1](ts)(e^u, e^v) = t1(e^u, e^v)/e^u, fl) :
```

$$fl := subs\left(\mathrm{D}[1](t1)(\mathrm{e}^u, \mathrm{e}^v) = \frac{t1(\mathrm{e}^u, \mathrm{e}^v)}{\mathrm{e}^u \cdot (1 - t1(\mathrm{e}^u, \mathrm{e}^v) \cdot t2(\mathrm{e}^u, \mathrm{e}^v))}, fl\right):$$

$$fl := subs\left(\mathrm{D}[1](t2)(\mathrm{e}^u, \mathrm{e}^v) = \frac{t1(\mathrm{e}^u, \mathrm{e}^v) \cdot t2(\mathrm{e}^u, \mathrm{e}^v)}{\mathrm{e}^u \cdot (1 - t1(\mathrm{e}^u, \mathrm{e}^v) \cdot t2(\mathrm{e}^u, \mathrm{e}^v))}, fl\right):$$

```
od:

fl := subs(ts(exp(u), exp(v)) = 1-e^2, fl) :
fl := subs(t1(exp(u), exp(v)) = 1-e, fl) :
fl := subs(t2(exp(u), exp(v)) = 1-e, fl) :
fl := subs(exp(u) = (1-e)*exp(e-1), exp(v) = (1-e)*exp(e-1), fl) :
fl := simplify(fl) :

return fl;
end proc:
> k :=proc(r, s)
  global f:
  return k_(r, s, f) :
  end proc:
> kg :=proc(r, s)
  global g :
  return k_(r, s, g) :
  end proc:
> k(2, 0);
  k(1, 1);
  k(3, 0);
  k(2, 1);
```

$$-\frac{e^2 - e + 1}{(e+1)^2 e (-2+e)}$$
$$\frac{-1 + 2e}{(e+1)^2 e (-2+e)}$$
$$\frac{(e-1)(2e^4 - 2e^3 + 4e^2 - 2e - 1)}{(e+1)^3 (-2+e)^2 e^3}$$
$$-\frac{(e-1)(3e^3 - 5e^2 + 2e + 1)}{(e+1)^3 (-2+e)^2 e^3} \qquad (5.2)$$

asymptotic (main result)

> # leading term

$$simplify\left(\frac{\exp(k(0,0))^{m \cdot (1+e)} \cdot \exp(kg(0,0))}{2 \cdot \pi \cdot (x0 \cdot y0)^m \cdot m \cdot (1+e) \cdot \mathrm{sqrt}(\Delta)}\right);$$

$$\frac{1}{2} \frac{(1-e^2)^{m(e+1)} ((e-1)^2 e^{2e-2})^{-m}}{\sqrt{-e(-2+e)} \sqrt{\frac{e-1}{e(-2+e)(e+1)^3}} \pi m (e+1)} \qquad (6.1)$$

$$> \# + \frac{H}{m \cdot (1+e)}$$
$$H := simplify\left(\frac{H}{2 \cdot \pi}\right);$$

$$H := -\frac{1}{12} \frac{e^6 - 10e^5 + 21e^4 - 2e^3 - 27e^2 + 20e - 5}{e^3 (-2+e)^2 (e-1)} \qquad (6.2)$$

$$> \# + \frac{H\_}{m^2 \cdot (1+e)^2}$$
$$H\_ := simplify\left(\frac{H\_}{2 \cdot \pi}\right);$$

$$H\_ := \frac{1}{288} \frac{1}{e^6 (e^2 - 3e + 2)^2 (-2+e)^2} (e^{12} + 52e^{11} + 34e^{10} - 1288e^9 + 4315e^8 - 5984e^7 + 404e^6 + 10552e^5 - 13241e^4 + 2468e^3 + 7186e^2 - 5960e + 1465) \qquad (6.3)$$

additional calculations

> # *asymptotic expansion of* $\frac{(m!)^2 n!}{(2m-n)!}$ *using Stirling's formula*

$$\left(1 + \frac{1}{12} \cdot x + \frac{1}{288} \cdot x^2\right)^2 \cdot \left(1 + \frac{1}{12 \cdot (1-e)} \cdot x + \frac{1}{288 \cdot (1-e)} \cdot x^2\right):$$
$$\left(1 - \frac{1}{12 \cdot (1+e)} \cdot x + \frac{1}{288 \cdot (1+e)^2} \cdot x^2\right) \cdot \left(1 + \frac{H}{1+e} \cdot x + \frac{H\_}{(1+e)^2} \cdot x^2\right):$$

```
simplify(expand(%·%%)) :
factor(simplify(coeff(%, x)));
factor(simplify(coeff(%%, x^2)));
f := unapply(%%, e) :
g := unapply(%%, e) :
```

$$\frac{1}{12} \frac{(2e^2 - 5e + 5)(e-1)^3}{e^3 (-2+e)^2}$$

$$\frac{1}{288} \frac{1}{(e-1)^2 (-2+e)^4 e^6} (4e^{12} - 53e^{11} + 325e^{10} - 1146e^9 + 2865e^8 - 6192e^7 + 12886e^6 - 24068e^5 + 35026e^4 - 35724e^3 + 23501e^2 - 8890e + 1465) \qquad (7.1)$$

> # *expected number of failures during the construction of N cuckoo hash tables,*
> # *a is based on the first order expansion, a2 makes use of the second order term too;*

$$N := 5 \cdot 10^5:$$

$$a := (m, e) \rightarrow piecewise\left(-evalf\left(\frac{f(e)}{m}\right) \cdot N < N, -evalf\left(\frac{f(e)}{m}\right) \cdot N, N\right):$$

$$a2 := (m, e) \rightarrow piecewise\left(-evalf\left(\frac{f(e)}{m} + \frac{g(e)}{m^2}\right) \cdot N < 0, 0, -evalf\left(\frac{f(e)}{m} + \frac{g(e)}{m^2}\right) \cdot N < N, -evalf\left(\frac{f(e)}{m} + \frac{g(e)}{m^2}\right) \cdot N, N\right):$$

▶ ***probabilities***

> *# Double Saddle Point Method, general case*
# We calculate the leading term and the coefficients of
$\frac{1}{m}$ *of the asymptotic expansion of* $\left[x^{m_1} y^{m_2}\right] f(x, y)^{2m-n} g(x, y)$

# we define $m_1 = m(1+c)$, $m_2 = m(1-c)$ *and* $e = 1 - \frac{n}{m}$

restart :

> *with*(*LinearAlgebra*) :
with(*plots*) :

calculations for variable substitution

> *# We perform a variable substitution*
to eleminate the term "st" of the function $-\frac{1}{2} s^2 k(0,2) - t\,s\,k(1,1) - \frac{1}{2} t^2 k(2,0)$

$N := \langle\langle a \mid b\rangle, \langle c \mid d\rangle\rangle$:
N^%T . N;
$P := \langle\langle k(2,0) \mid k(1,1)\rangle, \langle k(1,1) \mid k(0,2)\rangle\rangle$;

$$\begin{bmatrix} a^2+c^2 & a\,b+c\,d \\ a\,b+c\,d & b^2+d^2 \end{bmatrix}$$

$$P := \begin{bmatrix} k(2,0) & k(1,1) \\ k(1,1) & k(0,2) \end{bmatrix} \qquad (1.1)$$

> $N := \langle\langle \text{sqrt}(k(2,0) - k(1,1)\text{^}2 / k(0,2)) \mid 0\rangle,$
$\langle k(1,1) / \text{sqrt}(k(0,2)) \mid \text{sqrt}(k(0,2))\rangle\rangle$;
N^%T . N;
Determinant(N);

$$N := \begin{bmatrix} \sqrt{k(2,0) - \frac{k(1,1)^2}{k(0,2)}} & 0 \\ \frac{k(1,1)}{\sqrt{k(0,2)}} & \sqrt{k(0,2)} \end{bmatrix}$$

$$\begin{bmatrix} k(2,0) & k(1,1) \\ k(1,1) & k(0,2) \end{bmatrix}$$

$$\sqrt{k(2,0) - \frac{k(1,1)^2}{k(0,2)}}\,\sqrt{k(0,2)} \qquad (1.2)$$

> $sub :=$ *MatrixInverse*(N) . $\langle\langle u\rangle, \langle v\rangle\rangle$;

(1.3)

$$sub := \begin{bmatrix} \dfrac{u}{\sqrt{\dfrac{k(2,0)\,k(0,2) - k(1,1)^2}{k(0,2)}}} \\ -\dfrac{k(1,1)\,u}{k(0,2)\sqrt{k(2,0) - \dfrac{k(1,1)^2}{k(0,2)}}} + \dfrac{v}{\sqrt{k(0,2)}} \end{bmatrix} \qquad (1.3)$$

> *# perform test*
$S := 0$:
for i **from** 0 **to** 2 **do**:
 $j := 2 - i$:
 $S := S + m \cdot \left(\frac{I \cdot t}{\sqrt{m}}\right)^i \cdot \left(\frac{I \cdot s}{\sqrt{m}}\right)^j \cdot \frac{k(i,j)}{i! \cdot j!}$;
od:
S;
subs($t = sub[1,1], S$) :
subs($s = sub[2,1], \%$) :
simplify(%);

$$-\frac{1}{2} s^2 k(0,2) - t\,s\,k(1,1) - \frac{1}{2} t^2 k(2,0)$$

$$-\frac{1}{2} v^2 - \frac{1}{2} u^2 \qquad (1.4)$$

variable substitution

> *# expansion of* $(2m-n) \cdot \log(f)$, *calculate terms involving* $\frac{1}{\sqrt{2m-n}^{\,l}}$

hf :=**proc**(l)
 local i, j, S: $S := 0$:
 for i **from** 0 **to** l **do**:
  $j := l - i$:
  $S := S + \frac{1}{x^2} \cdot (I \cdot t \cdot x)^i \cdot (I \cdot s \cdot x)^j \cdot \frac{k(i,j)}{i! \cdot j!}$:
 od:
 return *collect*(*simplify*(S), x) :
 end proc:

> *# expansion of* $\log(g)$, *calculate terms involving* $\frac{1}{\sqrt{2m-n}^{\,l}}$

hg :=**proc**(l)
 local i, j, S: $S := 0$:
 for i **from** 0 **to** l **do**:
  $j := l - i$:
  $S := S + (I \cdot t \cdot x)^i \cdot (I \cdot s \cdot x)^j \cdot \frac{kg(i,j)}{i! \cdot j!}$:
 od:
 return *collect*(*simplify*(S), x) :
 end proc:

```
> series(exp(hf(3) + hf(4) + hg(1) + hg(2)), x, 3) :
  convert(%, polynom) :
  h := collect(coeff(%, x^2), [s, t], distributed) :
> series(exp(hf(3) + hf(4) + hf(5) + hf(6) + hg(1) + hg(2) + hg(3)
      + hg(4)), x, 5) :
  convert(%, polynom) :
  h_ := collect(coeff(%, x^4), [s, t], distributed) :
> subs(t = sub[1, 1], h) :
  subs(s = sub[2, 1], %) :
  h := collect(%, [u, v], distributed) :
> subs(t = sub[1, 1], h_) :
  subs(s = sub[2, 1], %) :
  h_ := collect(%, [u, v], distributed) :
```

integration

```
> int(int(e^(-1/2·(t^2 + s^2))·t^n·s^m, t = -∞..∞), s = -∞..∞) :
  i := unapply(%, n, m);
```

$$i := (n, m) \rightarrow \Gamma\left(\frac{1}{2}\,n+\frac{1}{2}\right)\Gamma\left(\frac{1}{2}\,m+\frac{1}{2}\right)2^{-1+\frac{1}{2}n+\frac{1}{2}m}\left(1+(-1)^n+(-1)^m + (-1)^{n+m}\right) \qquad (3.1)$$

```
> # calculate integrals
  H := 0 :
  for a from 0 to 6 do:
   for b from 0 to 6 do:
    if a > 0 then
     coeff(h, u^a)
    else
     subs(u = 0, h)
    fi:
    if b > 0 then
     coeff(%, v^b)
    else
     subs(v = 0, %)
    fi:
    H := H + %·i(a, b) :
   od:
  od:
> H_ := 0 :
  for a from 0 to 12 do:
   for b from 0 to 12 do:
    if a > 0 then
     coeff(h_, u^a)
    else
     subs(u = 0, h_)
    fi:
    if b > 0 then
     coeff(%, v^b)
    else
     subs(v = 0, %)
    fi:
    H_ := H_ + %·i(a, b) :
   od:
  od:
```

calculate x0 and y0 (saddle point)

```
> solve({A/(A+B-A·B) = m1/(m1+m2-n), B/(A+B-A·B) = m2/(m1+m2-n)}, {A, B});
  assign(%);
```

$$\left\{B=\frac{n}{m_1},\ A=\frac{n}{m_2}\right\} \qquad (4.1)$$

```
> x0 := A·exp(-B);
```

$$x_0 := \frac{n\,e^{-\frac{n}{m_1}}}{m_2} \qquad (4.2)$$

```
> y0 := A·exp(-B);
```

$$y_0 := \frac{n\,e^{-\frac{n}{m_1}}}{m_2} \qquad (4.3)$$

```
> C := simplify(A + B - A·B);
```

$$C := -\frac{n\,(-m_1-m_2+n)}{m_2\,m_1} \qquad (4.4)$$

definition of funtions f and g, calculation of cummulants

```
> f := ln(ts(e^u, e^v));
  g := ln(1/sqrt(1 - t1(e^u, e^v)·t2(e^u, e^v)));
```

$$f := \ln(ts(e^u, e^v))$$

$$g := -\frac{1}{2}\ln(1 - t1(e^u, e^v)\,t2(e^u, e^v)) \qquad (5.1)$$

```
> # main procedure to calculate cummulants
  k_ :=proc(r, s, f)
   local fl, i, j:

   fl := f:

   for i from 1 to s do
    fl := diff(fl, v) :
```

$fl := subs\left(\mathrm{D}[2](ts)(\mathrm{e}^u, \mathrm{e}^v) = \frac{t2(\mathrm{e}^u, \mathrm{e}^v)}{\mathrm{e}^v}, fl\right):$

$fl := subs\left(\mathrm{D}[2](t1)(\mathrm{e}^u, \mathrm{e}^v) = \frac{t1(\mathrm{e}^u, \mathrm{e}^v)\cdot t2(\mathrm{e}^u, \mathrm{e}^v)}{\mathrm{e}^v\cdot(1 - t1(\mathrm{e}^u, \mathrm{e}^v)\cdot t2(\mathrm{e}^u, \mathrm{e}^v))}, fl\right):$

$fl := subs\left(\mathrm{D}[2](t2)(\mathrm{e}^u, \mathrm{e}^v) = \frac{t2(\mathrm{e}^u, \mathrm{e}^v)}{\mathrm{e}^v\cdot(1 - t1(\mathrm{e}^u, \mathrm{e}^v)\cdot t2(\mathrm{e}^u, \mathrm{e}^v))}, fl\right):$

od:

for j **from** 1 **to** r **do**

$fl := diff(fl, u):$

$fl := subs\left(\mathrm{D}[1](ts)(\mathrm{e}^u, \mathrm{e}^v) = \frac{t1(\mathrm{e}^u, \mathrm{e}^v)}{\mathrm{e}^u}, fl\right):$

$fl := subs\left(\mathrm{D}[1](t1)(\mathrm{e}^u, \mathrm{e}^v) = \frac{t1(\mathrm{e}^u, \mathrm{e}^v)}{\mathrm{e}^u\cdot(1 - t1(\mathrm{e}^u, \mathrm{e}^v)\cdot t2(\mathrm{e}^u, \mathrm{e}^v))}, fl\right):$

$fl := subs\left(\mathrm{D}[1](t2)(\mathrm{e}^u, \mathrm{e}^v) = \frac{t1(\mathrm{e}^u, \mathrm{e}^v)\cdot t2(\mathrm{e}^u, \mathrm{e}^v)}{\mathrm{e}^u\cdot(1 - t1(\mathrm{e}^u, \mathrm{e}^v)\cdot t2(\mathrm{e}^u, \mathrm{e}^v))}, fl\right):$

od:

$fl := subs(ts(\exp(u), \exp(v)) = C, fl):$
$fl := subs(t1(\exp(u), \exp(v)) = A, fl):$
$fl := subs(t2(\exp(u), \exp(v)) = B, fl):$
$\#fl := subs(\exp(u) = (1-e)*\exp(e-1), \exp(v) = (1-e)*\exp(e-1), fl):$
$fl := simplify(fl):$

return fl;
end proc:

> $k :=$**proc**(r, s)
global f:
return $k\_(r, s, f):$
end proc:

> $kg :=$**proc**(r, s)
global g:
return $k\_(r, s, g):$
end proc:

> $k(2, 0):$
$simplify(subs(m_1 = (1+c)\cdot m, m_2 = m\cdot(1-c), n = m\cdot(1-e), \%));$
$k(1, 1):$
$simplify(subs(m_1 = (1+c)\cdot m, m_2 = m\cdot(1-c), n = m\cdot(1-e), \%));$

$$-\frac{(1+c)^2\,(1-c+c^2-e-c\,e+e^2)}{(c^2-2\,e+e^2)\,(1+e)^2}$$

$$-\frac{(-1+2\,e-c^2)\,(-1+c^2)}{(c^2-2\,e+e^2)\,(1+e)^2} \quad \mathbf{(5.2)}$$

> $\Delta := simplify(k(2,0)\cdot k(0,2) - k(1,1)^2):$
$simplify(subs(m_1 = (1+c)\cdot m, m_2 = m\cdot(1-c), n = m\cdot(1-e), \%));$

$$\frac{(-1+e)\,(1+c)^2\,(-1+c)^2}{(1+e)^3\,(c^2-2\,e+e^2)} \quad \mathbf{(5.3)}$$

asymptotic (main result)

> $\#\ calculate\ x_0^{m_1}\cdot y_0^{m_2}$

$\frac{n^{m_1+m_2}\cdot\exp(-2\cdot n)}{m_1^{m_2}\cdot m_2^{m_1}}:$

$F := simplify(subs(m_1 = (1+c)\cdot m, m_2 = m\cdot(1-c), n = m\cdot(1-e), \%));$

$$F := (-m\,(-1+e))^{2\,m}\,\mathrm{e}^{2\,m\,(-1+e)}\,((1+c)\,m)^{m\,(-1+c)}\,(-m\,(-1+c))^{-(1+c)\,m} \quad \mathbf{(6.1)}$$

> $\#\ leading\ term$

$simplify\left(\frac{\exp(k(0,0))^{m_1+m_2-n}\cdot\exp(kg(0,0))}{2\cdot\pi\cdot F\cdot(m_1+m_2-n)\cdot\sqrt{\Delta}}\right):$

$L := simplify(subs(m_1 = (1+c)\cdot m, m_2 = m\cdot(1-c), n = m\cdot(1-e), \%));$

$$L := \frac{1}{2}\left(\left(\frac{-1+e^2}{-1+c^2}\right)^{m\,(1+e)}(-m\,(-1+e))^{-2\,m}\,\mathrm{e}^{-2\,m\,(-1+e)}\,((1+c)\,m)^{-m\,(-1+c)}\,(-m\,(-1+c))^{(1+c)\,m}\right)\Bigg/\left(\sqrt{\frac{c^2-2\,e+e^2}{-1+c^2}}\,\sqrt{\frac{(-1+e)\,(1+c)^2\,(-1+c)^2}{(1+e)^3\,(c^2-2\,e+e^2)}}\,\pi\,m\,(1+e)\right) \quad \mathbf{(6.2)}$$

> $\#\ +\ \frac{H}{m_1+m_2-n}$

$simplify\left(\frac{H}{2\cdot\pi}\right):$

$H := simplify(subs(m_1 = (1+c)\cdot m, m_2 = m\cdot(1-c), n = m\cdot(1-e), \%));$

$$H := -\frac{1}{12}\,\frac{1}{(-1+e)\,(c^2-2\,e+e^2)^3}\,(10-45\,e-16\,e^3\,c^2+74\,e^2-6\,e\,c^2+41\,e^5-12\,e^6+e^7-2\,e\,c^6-8\,e^5\,c^2+24\,e^4\,c^2+2\,c^4-6\,e^3\,c^4+10\,e^2\,c^4-44\,e^4-23\,e^3) \quad \mathbf{(6.3)}$$

> $\#\ +\ \frac{H\_}{m^2\cdot(1+e)^2}$

$simplify\left(\frac{H\_}{2\cdot\pi}\right):$

$H\_ := simplify(subs(m_1 = (1+c)\cdot m, m_2 = m\cdot(1-c), n = m\cdot(1-e), \%));$

$$H\_ := \frac{1}{288}\,\frac{1}{(-1+e)^2\,(c^2-2\,e+e^2)^6}\,(556\,e^{10}\,c^4+5860-29700\,e-26552\,e^3\,c^2-11520\,c^2+54049\,e^2+49848\,e\,c^2+97640\,e^5-53833\,e^6-15000\,e^7+8000\,e\,c^6-112160\,e^5\,c^2+140616\,e^4\,c^2+8860\,c^4+21640\,e^3\,c^4+39916\,e^2\,c^4-55650\,e^4-24832\,e^3 \quad \mathbf{(6.4)}$$

$$+14244\,c^4 e^6-6980\,c^6 e^6-1312\,c^6 e^8-63756\,c^2 e^2+20584\,c^4 e^8$$
$$-37144\,c^4 e^7+5648\,c^6 e^7-28396\,e^9+9603\,e^{10}-170\,e^{12}-1216\,e^{11}$$
$$-40\,e^3 c^{10}-21836\,c^2 e^6+87104\,c^2 e^7-50228\,c^2 e^8+212\,c^8 e^6$$
$$-2412\,c^6-288\,c^8 e+148\,c^8+2912\,e^9 c^2+9032\,e^{10} c^2-4080\,e^{11} c^2$$
$$-8\,c^{10} e+41600\,e^8+53680\,c^4 e^5-81700\,c^4 e^4-5188\,e^9 c^4-80\,e^2 c^8$$
$$-5292\,e^2 c^6+552\,e^3 c^8+24\,c^{10} e^4+17036\,e^4 c^6-35388\,c^4 e+20\,c^8 e^4$$
$$-4208\,e^5 c^6+48\,e^{13}+596\,e^{12} c^2+e^{14}+4\,c^{12} e^2-10560\,e^3 c^6$$
$$-504\,c^8 e^5)$$

> *# check case* $m_1 = m_2$
simplify(*subs*($c = 0, L$));

$$\frac{1}{2}\,\frac{(1-e^2)^{m(1+e)}\,(-m(-1+e))^{-2m}\,\mathrm{e}^{-2m(-1+e)}\,m^{2m-1}}{\sqrt{-e(-2+e)}\,\sqrt{\frac{-1+e}{(1+e)^3\,e\,(-2+e)}}\,\pi\,(1+e)} \qquad \textbf{(6.5)}$$

> *simplify*(*subs*($c = 0, H$));

$$-\frac{1}{12}\,\frac{e^6-10\,e^5+21\,e^4-2\,e^3-27\,e^2+20\,e-5}{(-2+e)^2\,e^3\,(-1+e)} \qquad \textbf{(6.6)}$$

▼ calculate failure probability

> $x := 'x'$:
$$\left(1+\frac{1}{12\cdot(1+c)}\cdot x+\frac{1}{288\cdot(1+c)^2}\cdot x^2\right)\cdot\left(1+\frac{1}{12\cdot(1-c)}\cdot x+\frac{1}{288\cdot(1-c)^2}\cdot x^2\right)\cdot\left(1+\frac{1}{12\cdot(1-e)}\cdot x+\frac{1}{288\cdot(1-e)}\cdot x^2\right):$$
$$\left(1-\frac{1}{12\cdot(1+e)}\cdot x+\frac{1}{288\cdot(1+e)^2}\cdot x^2\right)\cdot\left(1+\frac{H}{1+e}\cdot x+\frac{H\_}{(1+e)^2}\cdot x^2\right):$$
simplify(*expand*(%%·%)) :
factor(*simplify*(*coeff*(%, x)));
factor(*simplify*(*coeff*(%%, x^2)));
$f :=$ *unapply*(%%, e, c) :
$g :=$ *unapply*(%%, e, c) :

$$-\frac{1}{12}\,\frac{(-1+e)^3\,(2\,e^3-9\,e^2+3\,c^2 e^2-9\,e\,c^2+15\,e-2\,c^4-10+10\,c^2)}{(c^2-2\,e+e^2)^3\,(1+c)\,(-1+c)}$$

$$\frac{1}{288}\,(-799\,e^{10} c^4+5860-c^{16} e-41420\,e+784724\,e^3 c^2-6\,e^3 c^{14} \qquad \textbf{(7.1)}$$
$$-23240\,c^2+144\,c^{12}+131029\,e^2+155728\,e\,c^2-272100\,e^5$$
$$+182842\,e^6+2\,e\,c^{14}-100380\,e^7-2700\,c^{10}+191948\,e\,c^6$$
$$+623492\,e^5 c^2-856318\,e^4 c^2+37760\,c^4-1039662\,e^3 c^4$$
$$+665285\,e^2 c^4+306501\,e^4+12\,c^{14} e^2-245790\,e^3-577\,e\,c^{12}$$
$$-624\,e^3 c^{12}+8646\,e^5 c^{10}-5622\,e^9 c^6-110\,e^{11} c^4+14\,e^{13} c^2$$
$$-6328\,e^7 c^8+204\,e^4 c^{12}-1284\,e^6 c^{10}+660\,e^{10} c^6+21\,e^{12} c^4$$
$$+628\,e^8 c^8-15\,e^5 c^{12}-20\,e^7 c^{10}-6\,e^{11} c^6-e^{13} c^4+170082\,c^4 e^6$$
$$-30552\,c^{10} e^2-65536\,c^6 e^6+19340\,c^6 e^8-15\,e^9 c^8-459590\,c^2 e^2$$
$$-24846\,c^4 e^8+6452\,c^4 e^7-21972\,c^6 e^7-22236\,e^9+8749\,e^{10}$$
$$+553\,e^{12}-2658\,e^{11}+35970\,e^3 c^{10}-309184\,c^2 e^6+108520\,c^2 e^7$$
$$-32224\,c^2 e^8+35092\,c^8 e^6-31612\,c^6-79232\,c^8 e+13792\,c^8$$
$$+11584\,e^9 c^2-4606\,e^{10} c^2+1318\,e^{11} c^2+14040\,c^{10} e+49114\,e^8$$
$$-575736\,c^4 e^5+993789\,c^4 e^4+8281\,e^9 c^4+194908\,e^2 c^8$$
$$-501568\,e^2 c^6-267188\,e^3 c^8-24044\,c^{10} e^4-637728\,e^4 c^6$$
$$-240544\,c^4 e+221600\,c^8 e^4+321032\,e^5 c^6-69\,e^{13}-210\,e^{12} c^2$$
$$+4\,e^{14}+840\,c^{12} e^2+731120\,e^3 c^6-113327\,c^8 e^5)\Big/\big((c^2-2\,e$$
$$+e^2)^6\,(-1+e)^2\,(1+c)^2\,(-1+c)^2\big)$$

> *# check case* $c = 0$
simplify($f(e, 0)$);

$$\frac{1}{12}\,\frac{(2\,e^2-5\,e+5)\,(-1+e)^3}{(-2+e)^2\,e^3} \qquad \textbf{(7.2)}$$

> *# expected number of failures during the construction of N cuckoo hash tables,*
# a is based on the first order expansion, a2 makes use of the second order term too;
$N := 5\cdot 10^5$:
$$a := (m, e, c) \rightarrow \mathit{piecewise}\left(-\mathit{evalf}\left(\frac{f(e,c)}{m}\right)\cdot N < N, -\mathit{evalf}\left(\frac{f(e,c)}{m}\right)\cdot N, N\right):$$
$$a2 := (m, e, c) \rightarrow \mathit{piecewise}\left(-\mathit{evalf}\left(\frac{f(e,c)}{m}+\frac{g(e,c)}{m^2}\right)\cdot N < 0, 0, -\mathit{evalf}\left(\frac{f(e,c)}{m}+\frac{g(e,c)}{m^2}\right)\cdot N < N, -\mathit{evalf}\left(\frac{f(e,c)}{m}+\frac{g(e,c)}{m^2}\right)\cdot N, N\right):$$

> *# leading coefficient of* $\frac{m_1!\cdot m_2!\cdot n!}{(2m-n)!}$ (*Stirling's formula*)
$$\frac{2\cdot\pi\cdot m_1^{m_1-n}\cdot m_2^{m_2-n}\cdot n^n}{(m_1+m_2-n)^{m_1+m_2-n}\cdot\exp(2\cdot n)}\sqrt{\frac{m_1\cdot m_2\cdot n}{m_1+m_2-n}}:$$
simplify(*subs*($m_1 = (1+c)\cdot m$, $m_2 = m\cdot(1-c)$, $n = m\cdot(1-e)$, %));

$$2\,\pi\,((1+c)\,m)^{m(c+e)}\,(-m(-1+c))^{m(-c+e)}\,(-m(-1+e))^{-m(-1+e)}\sqrt{\frac{m^2\,(-1+e)\,(-1+c^2)}{1+e}}\,(m(1+e))^{-m(1+e)}\,\mathrm{e}^{2m(-1+e)} \qquad \textbf{(7.3)}$$

▼ calculate max. n depending on c

> *subs*($m_1 = (1+c)\cdot m$, $m_2 = m\cdot(1-c)$, $\exp(kg(0,0))$);

$solve\left(\frac{1}{\%} = 0,\ n\right);$

$$e^{g} \qquad (8.1)$$

> $emin := c \rightarrow 1 - \sqrt{1 - c^2};$
$nmax := unapply((1 - emin(c)) \cdot m,\ m,\ c);$

$$emin := c \rightarrow 1 - \sqrt{1 - c^2}$$

$$nmax := (m,\ c) \rightarrow \sqrt{1 - c^2}\ m \qquad (8.2)$$

▶ ***probabilities***

Index

Bibliography

Milton Abramowitz and Irena A. Stegun, editors. *Handbook of Mathematical Functions*. Dover Publications, Washington, DC, 1970.

Tsiry Andriamampianina and Vlady Ravelomanana. Enumeration of connected uniform hypergraphs. In *FPSAC 05*, 2005.

Nikolas Askitis. *Efficient Data Structures for Cache Architectures*. PhD thesis, School of Computer Science and Information Technology, RMIT University, Australia, August 2007.

Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999.

Andrew Binstock. Hashing rehashed: Is ram speed making your hashing less efficient? *Dr. Dobb's Journal*, 4(2), 1996.

John R. Black, Charles U. Martel, and Hongbin Qi. Graph and hashing algorithms for modern architectures: Design and performance. In *WAE '92, Saarbrücken, Germany, August 20-22, 1998, Proceedings*, pages 37–48. Max-Planck-Institut für Informatik, 1998.

Jonah Blasiak and Rick Durrett. Random oxford graphs. *Stochastic Process. Appl.*, 115 (8):1257–1278, 2005.

Béla Bollobás. *Random Graphs*. Cambridge University Press, Cambridge, UK, second edition, 2001.

Richard P. Brent. Reducing the retrieval time of scatter storage techniques. *Commun. ACM*, 16(2):105–109, 1973.

Andrei Z. Broder and Michael Mitzenmacher. Using multiple hash functions to improve ip lookups. In *INFOCOM*, pages 1454–1463, 2001.

Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

Pedro Celis, Per-Åke Larson, and J. Ian Munro. Robin hood hashing (preliminary report). In *26th Annual Symposium on Foundations of Computer Science, 21-23 October 1985, Portland, Oregon, USA*, pages 281–288. IEEE, 1985.

John B. Conway. *Functions of One Complex Variable*. Springer, New York, second edition, 1978.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, Mass., London, England, second edition, 2001.

Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. Perfect hashing. *Theoretical Computer Science*, 182(1-2):1–143, 1997.

Jurek Czyzowicz, Wojciech Fraczak, and Feliks Welfed. Notes about d-cuckoo hashing. Technical Report RR 06/05-1, Université du Québec en Outaouais, 2006.

Ketan Dalal, Luc Devroye, Ebrahim Malalla, and Erin McLeis. Two-way chaining with reassignment. *SIAM J. Comput.*, 35(2):327–340, 2005.

Luc Devroye and Pat Morin. Cuckoo hashing: Further analysis. *Information Processing Letters*, 86(4):215–219, 2003.

Reinhard Diestel. *Graph Theory*. Springer, Berlin, 2005.

Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, 380(1-2):47–68, 2007.

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. ACM, 2003.

Martin Dietzfelbinger, Joseph Gil, Yossi Matias, and Nicholas Pippenger. Polynomial hash functions are reliable (extended abstract). In *ICALP '92*, volume 623 of *LNCS*, pages 235–246. Springer, 1992. ISBN 3-540-55719-9.

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

Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1):19–51, 1997.

Michael Drmota. A bivariate asymptotic expansion of coefficients of powers of generating functions. *European Journal of Combinatorics*, 15(2):139–152, 1994.

Michael Drmota and Reinhard Kutzelnigg. A precise analysis of cuckoo hashing. *preprint*, 2008.

Michael Drmota and Michèle Soria. Marking in combinatorial constructions: Generating functions and limiting distributions. *Theoretical Computer Science*, 144(1&2):67–99, 1995.

Michael Drmota and Michèle Soria. Images and preimages in random mappings. *SIAM Journal on Discrete Mathematics*, 10(2):246–269, 1997.

Paul Erdős and Alfréd Rényi. On random graphs i. *Publ. Math. Debrecen*, 6:290–297, 1959.

Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hungar. Acad. Sci.*, 5:17–61, 1960.

Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Bull. Inst. Internat. Statist.*, 38:343–347, 1961.

Úlfar Erlingsson, Mark Manasse, and Frank McSherry. A cool and practical alternative to traditional hash tables. In *WDAS '06: 7th Workshop on Distributed Data and Structures*, 2006.

Philippe Flajolet and Andrew M. Odlyzko. The average height of binary trees and other simple trees. *Journal of Computer and System Sciences*, 25(2):171–213, 1982.

Philippe Flajolet and Andrew M. Odlyzko. Random mapping statistics. *LNCS*, 434: 329–354, 1990.

Philippe Flajolet and Robert Sedgewick. *An introduction to the analysis of algorithms.* Addison-Wesley, Boston, Mass., 2001.

Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics, web edition.* To be published by Cambridge University Press, Cambridge, UK, 2008. available from the authors web sites.

Philippe Flajolet, Donald E. Knuth, and Boris Pittel. The first cycles in an evolving graph. *Discrete Mathematics*, 75(1-3):167–215, 1989.

Philippe Flajolet, Zhicheng Gao, Andrew M. Odlyzko, and L. Bruce Richmond. The distribution of heights of binary trees and other simple trees. *Combinatorics, Probability & Computing*, 2:145–156, 1993.

Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. In *STACS '03*, volume 2607 of *LNCS*, pages 271–282. Springer, 2003.

Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory Comput. Syst.*, 38(2):229–248, 2005.

Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with O(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

Danièle Gardy. Some results on the asymptotic behaviour of coefficients of large powers of functions. *Discrete Mathematics*, 139(1-3):189–217, 1995.

Omer Gimenez, Anna de Mier, and Marc Noy. On the number of bases of bicircular matroids. *Annals of Combinatorics*, 9(1):35–45, 2005.

G. H. Gonnet and R. Baeza-Yates. *Handbook of algorithms and data structures: in Pascal and C.* Addison-Wesley, Boston, MA, USA, second edition, 1991. ISBN 0-201-41607-7.

Gaston H. Gonnet. Expected length of the longest probe sequence in hash code searching. *J. ACM*, 28(2):289–304, 1981.

Gaston H. Gonnet and J. Ian Munro. The analysis of an improved hashing technique. In *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing, 2-4 May 1977, Boulder, Colorado, USA*, pages 113–121. ACM, 1977.

I. J. Good. Saddle-point methods for the multinomial distribution. *Ann. Math. Stat.*, 28 (4):861–881, 1957.

Ian P. Goulden and David M. Jackson. *Combinatorial Enumeration.* Dover, New York, 1983.

Daniel H. Greene and Donald E. Knuth. *Mathematics for the Analysis of Algorithms.* Birkhäuser, Boston, second edition, 1982.

Gregory L. Heileman and Wenbin Luo. How caching affects hashing. In *ALENEX /ANALCO 2005, Vancouver, BC, Canada, 22 January 2005*, pages 141–154. SIAM, 2005.

Hsien-Kuei Hwang. Large deviations for combinatorial distributions. i. central limit theorems. *Annals of Applied Probability*, 6(1):297–319, 1996.

Svante Janson, Donald E. Knuth, Tomasz Łuczak, and Boris Pittel. The birth of the giant component. *Random Structures and Algorithms*, 4(3):233–359, 1993.

Svante Janson, Tomasz Łuczak, and Andrzej Rucinski. *Random Graphs.* Wiley, New York, 2000.

I. B. Kalugin. The number of components of a random bipartite graph. *Discrete Mathematics and Applications*, 1(3):289–299, 1991.

Alfons Kemper and Antré Eickler. *Datenbanksysteme.* Oldenburg, München, Wien, sixth edition, 2006.

Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. In *Proceedings of the 16th Annual European Symposium on Algorithms*, 2008.

Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, Boston, second edition, 1998.

Reinhard Kutzelnigg. Analyse von Hash-Algorithmen. Master's thesis, TU-Wien, 2005.

Reinhard Kutzelnigg. Bipartite random graphs and cuckoo hashing. In *Proceedings of the 4th Colloquium on Mathematics and Computer Science*, Discrete Mathematics and Theoretical Computer Science, pages 403–406, 2006.

Reinhard Kutzelnigg. An improved version of cuckoo hashing: Average case analysis of construction cost and search operations. In *Proceedings of the 19th internatinal workshop on combinatorial algoritms*, pages 253–266, 2008.

Chuck Lever. Linux kernel hash table behavior: Analysis and improvements. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 13–26, 2000.

M. Loéve. *Probability Theory 1.* Springer, New York, fourth edition, 1977.

Tomasz Łuczak. Random trees and random graphs. *Random Structures and Algorithms*, 13(3-4):485–500, 1998.

Wenbin Luo and Gregory L. Heileman. Improved exponential hashing. *IEICE Electronic Express*, 1(7):150–155, 2004.

G. Marsaglia. The marsaglia random number cdrom including the diehard battery of tests of randomness. http://stat.fsu.edu/pub/diehard/.

Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998. ISSN 1049-3301. doi: http://doi.acm.org/10.1145/272991.272995.

A. Meir and J. W. Moon. On the altitude of nodes in random trees. *Canadian Journal of Mathematics*, 30:997–1015, 1978.

J. Ian Munro and Pedro Celis. Techniques for collision resolution in hash tables with open addressing. In *Proceedings of the Fall Joint Computer Conference*, pages 601–610. IEEE Computer Society, 1986. ISBN 0-8186-0743-2.

Moni Naor, Gil Segev, and Udi Wieder. History-independent cuckoo hashing. In *ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings*, volume 5126 of *LNCS*, pages 631–642. Springer, 2008. ISBN 978-3-540-70582-6.

Anna Pagh, Rasmus Pagh, and Milan Ruzic. Linear probing with constant independence. In *STOC '07*, pages 318–327, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-631-8. doi: http://doi.acm.org/10.1145/1250790.1250839.

Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. Research Series RS-01-32, BRICS, Department of Computer Science, University of Aarhus, 2001a.

Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Algorithms - ESA 2001, 9th Annual European Symposium*, volume 2161 of *LNCS*, pages 121–133. Springer, 2001b.

Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51 (2):122–144, 2004.

Athanasios Papoulis and S. Unnikrishna Pillai. *Probability, Random Variables and Stochastic Processes.* McGraw-Hill, Boston, fourth edition, 2002.

A. P. Prudnikov, Yu. A. Brychkov, and O. I. Marichev. *Integrals and Series, Volume 3: More Special Functions.* Gordon and Breach, New York and London, 1989.

William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990. ISSN 0001-0782.

M. V. Ramakrishna and Justin Zobel. Performance in practice of string hashing functions. In *DASFAA*, Advanced Database Research and Development Series, pages 215–224. World Scientific, 1997.

John Riordan. *Combinatorial Identities.* Wiley, New York, 1968.

John Riordan and N. J. A. Sloane. The enumeration of rooted trees by total height. *J. Australian Math. Soc.*, 10:278–282, 1969.

Kenneth A. Ross. Efficient hash probes on modern processors. IBM Research Report RC24100, IBM, 2006.

H. I. Scoins. The number of trees with nodes of alternate parity. *Proc. Cambridge Philos. Soc.*, 58:12–16, 1962.

Alan Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications (extended abstract). In *FOCS*, pages 20–25. IEEE, 1989.

Bradley J. Smith, Gregory L. Heileman, and Chaouki T. Abdallah. The exponential hash function. *ACM Journal of Experimental Algorithms*, 2:3, 1997.

B. Stroustrup. *Die C++ Programmiersprache.* Adisson-Wesley, München, 2000.

Thinh Ngoc Tran and Surin Kittitornkun. Fpga-based cuckoo hashing for pattern matching in nids/nips. In *APNOMS*, pages 334–343, 2007.

Berthold Vöcking. How asymmetry helps load balancing. *J. ACM*, 50(4):568–589, 2003.

Herbert S. Wilf. *generatingfunctionology.* Acad. Press, San Diego, Calif., second edition, 1999.

Semyom B. Yakubovich. Integral transformations by the index of lommel's function. *Periodica Mathematica Hungarica*, 46(2):223– 233, 2003.

Marcin Zukowski, Sándor Héman, and Peter A. Boncz. Architecture-conscious hashing. In *DaMoN Proceedings of the 2nd international workshop on Data management on new hardware.* ACM Press, 2006.

Daniel Zwillinger. *Handbook of differential equations.* Academic Press, Boston, second edition, 1992.

Table of Symbols and Notations

Symbol	Interpretation
$\Re z$	real part of the complex number z
$\Im z$	imaginary part of the complex number z
$\mathbb{P}(A)$	probability of an event A
$\mathbb{P}(A\|B)$	probability of an event A, conditioned on the occurrence of event$B$
$\mathbb{E}\eta$	expectation of a random variable η
$\mathbb{V}\eta$	variance of a random variable η
$\delta_{i,j}$	Kronecker's delta, $\delta_{i,j} = \begin{cases} 1 \text{ if } i \text{ equals } j \\ 0 \text{ otherwise} \end{cases}$
$\lfloor x \rfloor$	floor of x, greatest integer k satisfying $k \le x$
$\{x\}$	fractional part of x (*i.e.* $x - \lfloor x \rfloor$)
$\oplus$	bitwise exclusive or operation
$x^{\overline{k}}$	$x(x+1)(x+2)\dots(x+k-1)$
$x^{\underline{k}}$	$x(x-1)(x-2)\dots(x-k+1)$
$f(n) = o(g(n))$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$f(n) = \mathcal{O}(g(n))$	$\|f(n)\| \le c\|g(n)\|$ holds for all suff. large n and a constant $c > 0$
$f(n) = \Theta(g(n))$	$c_1\|g(n)\| \le \|f(n)\| \le c_2\|g(n)\|$ for suff. large n and const. $c_1, c_2 > 0$
$f(n) = \Omega(g(n))$	$\|f(n)\| \ge c\|g(n)\|$ holds for all suff. large n and a constant $c > 0$
$\mathbf{x}$	bold font indicates vectors, $\mathbf{x} = (x_1, \dots, x_d)$
$\mathbf{x}^{\mathbf{k}}$	multipower, $\mathbf{x}^{\mathbf{k}} = x_1^{k_1} \dots x_d^{k_d}$
$[x^n]a(x)$	n-th coefficient of the series $a(x) = \sum_{n\ge0} a_n x^n$
$t(x)$	generating function of rooted labelled trees
$\tilde{t}(x)$	generating function of unrooted labelled trees
$t(x,y)$	generating function of rooted labelled bipartite trees
$\tilde{t}(x,y)$	generating function of unrooted labelled bipartite trees

List of Figures

List of Tables

List of Listings

Lebenslauf

Ich wurde am 18. Mai 1980, als Sohn von Rosemarie Kutzelnigg, geborene Otter, und Alfred Kutzelnigg, in Vorau geboren. Von 1986 bis 1990 besuchte ich die Volksschule in Vorau, und von 1990 bis 1994 die Hauptschule, ebenfalls in meinem Geburtsort. Anschließend besuchte ich von 1994 bis 1999 die höhere technische Bundeslehranstalt für Elektrotechnik in Pinkafeld, mit Schwerpunkt Steuerungs- und Regelungstechnik. Dort legte ich am 16. Juni 1999 die Reifeprüfung mit ausgezeichneten Erfolg ab. In den darauf folgenden Monaten leiste ich den Präsenzdienst beim österreichischen Bundesheer in Pinkafeld ab.

Im September 2000 immatrikulierte ich an der Technischen Universität Wien und begann mit dem Studium der Technischen Mathematik, Studienrichtung Mathematische Computerwissenschaften. Dieses schloss ich mit der Ablegung der zweiten Diplomprüfung am 23. November 2005 mit ausgezeichnetem Erfolg ab. Während des Studiums war ich im Sommersemester 2003 für das Institut für Computergraphik und Algorithmen, und ab dem darauf folgenden Wintersemester für das Institut für Analysis und Scientific Computing, als Tutor tätig.

Im März 2006 inskribierte ich das Doktoratsstudium der technischen Wissenschaften. Gleichzeitig nahm ich ein Anstellung als Projektassistent im von Prof. Michael Drmota geleiteten FWF-Projekt S9604 „Analytic and Probabilistic Methods in Combinatorics“ an, welches dem nationalen Forschungsnetzwerk „Analytic Combinatorics and Probabilistic Number Theory“ zugehört. Nach einem schweren Sportunfall im März 2008 und mehrmonatigen Krankenstand, wechselte ich im Juli 2008 zum Forschungsprojekt „Network Models, Governance and R&D collaboration networks“, Projekt Nummer 028875 des EU FP6-NEST-Adventure Programms. Für dieses, an der TU Wien ebenfalls von Prof. Michael Drmota geleitete Projekt, bin ich derzeit als Projektassistent tätig.