Fabiano Cupertino Botelho
Supervisor - Nivio Ziviani

Near-Optimal Space Perfect Hashing Algorithms

PhD. dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais as a partial requirement to obtain the PhD. degree in Computer Science.

Belo Horizonte
September 29, 2008

To my dear wife *Janaína*.
To my dear parents *Maria Lúcia* and *José Vítor*.
To my dear sisters *Gleiciane* and *Cristiane*.

Acknowledgements

To God for having granted me life and wisdom to realize a dream of childhood and for the great help in difficult moments.

To my dear wife Janaína Marcon Machado Botelho for the love, understanding by several times when I could not give her the attention she deserves, companionship and encouragement during moments in which I desired to give up everything. Jana thank you for sharing your life with me and the victories won during the entire doctorate. With the grace of God in our lives we will continue to be very happy.

To my dear parents Maria Lúcia de Lima Botelho and José Vitor Botelho for sacrifices made in the past that have given support for this achievement.

To my dear sisters Cristiane Cupertino Botelho and Gleiciane Cupertino Botelho for the love of the best two sisters in the world.

To my dear aunt Márcia Novaes Alves and my dear uncle Sudário Alves for always welcome me with affection, giving me much support throughout my doctorate.

To Prof. Nivio Ziviani for the excellent work of supervision and for being an example of professionalism and dedication to work. His extensive experience in academic research, and particularly in the areas of information retrieval and algorithms have been of extreme importance to realize this work. In addition, his excellent support, attention and encouragement were of great importance not only for completing the doctorate, but also for my academic and professional life.

To Prof. Rasmus Pagh with whom I've learned a lot about techniques for designing and analyzing hashing algorithms, being crucial his participation in this thesis.

To Prof. Yoshiharu Kohayakawa for the attention dedicated to the discussions that contributed to improve the quality of this work. Thanks also to receive me at the

Institute of Mathematics and Statistics at the University of São Paulo and for all the support given to my work during the time I spent in São Paulo.

To Prof. Edleno Silva de Moura for trusting on me and for always encouraging me. Thanks also to receive me at the Department of Computer Science at the Federal University of Amazonas during the time I spent in Manaus.

To the other Professors that evaluated this thesis, namely, Gaston Gonnet, Antônio Alfredo Loureiro, Wagner Meira Jr. and Jayme Luiz Szwarcfiter for having accepted to participate of the PhD. defense and for the relevant criticisms and suggestions.

To Djamal Belazzougui for the intelligent suggestions and contributions made to this thesis and to the CMPH library.

To Davi Reis for having conceived the idea of the CMPH library, which was fundamental to disseminate the results obtained in this thesis.

To my colleague and friend Marco Antônio Pinheiro de Cristo for the fun moments we spent together during our English classes and for always encoraging me.

To my colleague and friend Thierson Couto for his friendship, and to be always ready to cooperate.

To my colleague and friend David Menotti for the discussions, suggestions and criticisms that contributed much in the beginning of this work.

To my colleague and friend David Fernandes for having received me in your home during the time I spent in Manaus and for his endless friendship.

To my colleagues and friends of our great and unforgettable soccer team Curucu and their wives for the friendship conquered during the period we spent together. Thanks Pedro Neto, Maurício Figueiredo, Eduardo Freire Nakamura, Ruiter Caldas, André Lins, José Pinheiro, Guillermo Camara Chavez, Martin Gomez Ravetti, David Patricio Viscarra del Pozo and David Menotti for the amazing and fun moments that served to relieve the stress of this difficult period of doctorate.

To colleagues and friends from that period of our undergraduate course that, through the mailing list intrigas99, always supported me being close or distant. I thank also for all the good laughs that I gave when I was reading some posts of the list, which certainly helped a lot to ease the tension in difficult times.

To my colleagues and friends of the Laboratory for Treating Information (LATIN) Anísio Mendes Lacerda, Álvaro Pereira Jr., Charles Ornelas Almeida, Claudine Santos Badue, Daniel Galinkin, Denilson Pereira, Guilherme Vale Menezes, Hendrickson R. Langbehn, Humberto Mossri, Marco Antônio Pinheiro de Cristo, Marco Aurélio Barreto Modesto, Pável Calado and Wladmir Cardoso Brandão for the criticism and suggestions provided during the defense preparation and for the climate of friendship we have established within LATIN.

To Professors and employees of the Department of Computer Science at the Federal University of Minas Gerais that in various ways contributed to the completion of this work.

To Professors and employees of the Department of Computer Engineering at the Federal Center for Technological Education of Minas Gerais for having received me so well and in a so respectful manner to integrate the department team.

To the scholarships granted by CAPES (Coordination of Improvement of Higher Education) and CNPq (National Council for Scientific and Technological Development), which served as subsidy for the time dedicated to this thesis.

Abstract

A perfect hash function (PHF) $h : S \to [0, m-1]$ for a key set $S \subseteq U$ of size n, where $m \geq n$ and U is a key universe, is an injective function that maps the keys of S to unique values. A minimal perfect hash function (MPHF) is a PHF with $m = n$, the smallest possible range. Minimal perfect hash functions are widely used for memory efficient storage and fast retrieval of items from static sets, such as words in natural languages, reserved words in programming languages or interactive systems, universal resource locations (URLs) in web search engines, or item sets in data mining techniques.

In this thesis we present a simple, highly scalable and near-space optimal perfect hashing algorithm. Evaluation of a PHF on a given element of S requires constant time, and the dominating phase in the construction algorithm consists of sorting n fingerprints of $O(\log n)$ bits in $O(n)$ time. The space usage depends on the relation between m and n. For $m = n$ the space usage is in the range $2.62n$ to $3.3n$ bits, depending on the constants involved in the construction and in the evaluation phases. For $m = 1.23n$ the space usage is in the range $1.95n$ to $2.7n$ bits. In all cases, this is within a small constant factor from the information theoretical minimum of approximately $1.44n$ bits for MPHFs and $0.89n$ bits for PHFs, something that has not been achieved by previous algorithms, except asymptotically for very large n. This small space usage opens up the use of MPHFs to applications for which they were not useful in the past.

We demonstrate the scalability of our algorithm by constructing an MPHF for a set of 1.024 billion URLs from the World Wide Web of average length 64 characters in approximately 50 minutes, using a commodity PC. We also present a distributed and parallel implementation of the algorithm, which generates an MPHF for the same URL set, using a 14 computer cluster, in approximately 4 minutes, achieving an almost linear speedup. Also, for 14.336 billion 16-byte random integers distributed among the 14 participating machines, the algorithm outputs an MPHF in approximately 50 minutes, with a performance degradation of 20%.

Resumo

Uma função *hash* perfeita (FHP) $h : U \to [0, m-1]$ para um conjunto de chaves $S \subseteq U$ de tamanho n, onde $m \geq n$ e U é um universo de chaves, é uma função injetora que mapeia as chaves de S para valores únicos. Uma função *hash* perfeita mínima (FHPM) é uma FHP com $m = n$, o menor intervalo possível. Funções *hash* perfeitas mínimas são amplamente utilizadas para armazenamento eficiente e recuperação rápida de itens de conjuntos estáticos, como palavras em linguagem natural, palavras reservadas em linguagens de programação ou sistemas interativos, URLs (*universal resource locations*) em máquinas de busca, ou conjuntos de itens em técnicas de mineração de dados.

Nesta tese nós apresentamos um algoritmo de *hashing* perfeito altamente escalável e de espaço quase ótimo. A avaliação de uma FHP sobre um dado elemento de S requer tempo constante, e a fase dominante no algoritmo de construção consiste da ordenação de n *fingerprints* de $O(\log n)$ *bits* em tempo $O(n)$. A utilização de espaço depende da relação entre m e n. Para $m = n$ a utilização de espaço está dentro do intervalo $2,62n$ à $3,3n$ *bits*, dependendo das constantes envolvidas nas fases de construção e avaliação. Para $m = 1,23n$ a utilização de espaço está dentro do intervalo $1,95n$ à $2,7n$ *bits*. Em todos os casos, isto está distante por um pequeno fator constante do mínimo teórico de aproximadamente $1,44n$ *bits* para FHPMs e $0,89n$ *bits* para FHPs, uma coisa que não foi alcançada por algoritmos anteriores, exceto assintóticamente para valores de n muito grandes. Esta pequena utilização de espaço permitiu o uso de FHPMs em aplicações para as quais elas não eram úteis no passado.

Nós demonstramos a escalabilidade do nosso algoritmo ao construir uma FHPM para um conjunto de $1,024$ bilhões de URLs da *World Wide Web* de tamanho médio igual a 64 caracteres em aproximadamente 50 minutos, usando um PC comodite. Nós também apresentamos uma implementação distribuída e paralela do algoritmo, a qual gera uma FHPM para o mesmo conjunto de URLs, usando um cluster de 14 computadores, em aproximadamente 4 minutos, alcançando um *speedup* quase linear. Além disso, para $14,336$ bilhões de números inteiros de 16 *bytes* gerados aleatoriamente e distribuídos entre as 14 máquinas participantes, o algoritmo gera uma FHPM em aproximadamente 50 minutos, com uma degradação de desempenho de 20%.

Papers

Published

1. F.C. Botelho, Y. Kohayakawa, and N. Ziviani. A practical minimal perfect hashing method. In Proceedings of the 4th International Workshop on Efficient and Experimental Algorithms (WEA'05), pages 488–500. Springer LNCS vol. 3503, 2005.

2. F.C. Botelho, R. Pagh, and N. Ziviani. Simple and Space-Efficient Minimal Perfect Hash Functions. In Proceedings of the 10th Workshop on Algorithms and Data Structures (WADS'07), pages 139–150. Springer LNCS vol. 4619, 2007.

3. F.C. Botelho, and N. Ziviani. External Perfect Hashing for Very Large Key Sets. In Proceedings of the 16th Conference on Information and Knowledge Management (CIKM'07), pages 653–662, ACM Press, 2007.

4. F.C. Botelho, D. Galinkin, W. Meira Jr., and N. Ziviani. Distributed Perfect Hashing for Very Large Key Sets. In Proceedings of the 3rd International ICST Conference on Scalable Information Systems (InfoScale'08), Naples, Italy, June 2008.

5. F.C. Botelho, H.R. Langbehn, G.V. Menezes, and N. Ziviani. Indexing Internal Memory with Minimal Perfect Hash Functions. In Proceedings of the 23rd Brazilian Symposium on Database (SBBD'08), Campinas, Brazil, October 2008.

Accepted After The PhD. Defense

1. F.C. Botelho, and N. Ziviani. Near-optimal space perfect hashing algorithms. In Proceedings of the 22nd SBC Theses and Dissertations Contest (CTD'09), Bento Gonçalves, Brazil, July 2009.

2. D. Belazzougui, F.C. Botelho, and M. Dietzfelbinger. Hash, Displace, and Compress. In Proceedings of the 17th European Symposium on Algorithms (ESA'09), Copenhagen, Denmark, September 2009.

Submitted After The PhD. Defense

1. F.C. Botelho, A. Lacerda, G.V. Menezes, and N. Ziviani. Minimal perfect hashing: A competitive method for indexing internal memory. Submitted to a Special Issue of Information Sciences Journal, 2009.

2. F.C. Botelho, R. Pagh, and N. Ziviani. A scalable and near-optimal space perfect hashing algorithm. Submitted to ACM Transactions on Algorithms, 2009.

3. F.C. Botelho, N. Wormald, and N. Ziviani. Cores of Random r-Partite Hypergraphs. Submitted to Information Processing Letters, 2009.

4. F.C. Botelho, R. Pagh, and N. Ziviani. Perfect Hashing in Practice. Submitted to a Special Issue of Software Practice and Experience Journal, 2009.

5. F.C. Botelho, W.C. Brandão, and N. Ziviani. Minimal Perfect Hashing and Bloom Filters Made Practical. Submitted to 24th Brazilian Symposium on Database (SBBD'09), Fortaleza, Brazil, October 2009.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The need to access items based on the value of a key is ubiquitous in areas including artificial intelligence, data structures, database, data mining and information retrieval. Some types of databases are updated only rarely, typically by periodic batch updates. This is true, for example, for most data warehousing applications (see [71] for more examples and discussion). In such scenarios it is possible to improve query performance by creating very compact representations of keys by minimal perfect hash functions.

In applications where the key set is fixed for a long period of time the construction of a minimal perfect hash function can be done as part of the preprocessing phase. For example, On-Line Analytical Processing (OLAP) applications use extensive preprocessing of data to allow very fast evaluation of certain types of queries. More formally, given a *static* key set $S \subseteq U$ of size n from a key universe U of size u, where each key is associated with satellite data, the question we are interested in is: what are the data structures that provide the best trade-off between space usage and lookup time?

An efficient way to represent a key set in terms of lookup time is using a table indexed by a hash function. Considering $S \subseteq U$ and given a key $x \in S$, a hash function h computes an integer in $[0, m-1]$ for the storage or retrieval of x in a *hash table*. Hashing methods for *non-static key sets* can be used to construct data structures storing S and supporting membership queries of the type "$x \in S$?" in expected $O(1)$ time. However, they involve a certain amount of wasted space owing to unused locations in the table and wasted time to resolve collisions when two or more keys are hashed to the same table location.

Perfect hashing is a space-efficient way of creating compact representation for a static set S of n keys. For applications with successful searches, the representation of a key $x \in S$ is simply the value $h(x)$, where h is a perfect hash function (PHF) for the set S of values considered. The word "perfect" refers to the fact that the function will map the elements of S to unique values (is identity preserving). *Minimal perfect hash function* (MPHF) produces values that are integers in the range $[0, n-1]$, which is the smallest possible range. Figure 1.1(a) illustrates a perfect hash function and Figure 1.1(b) illustrates a minimal perfect hash function.

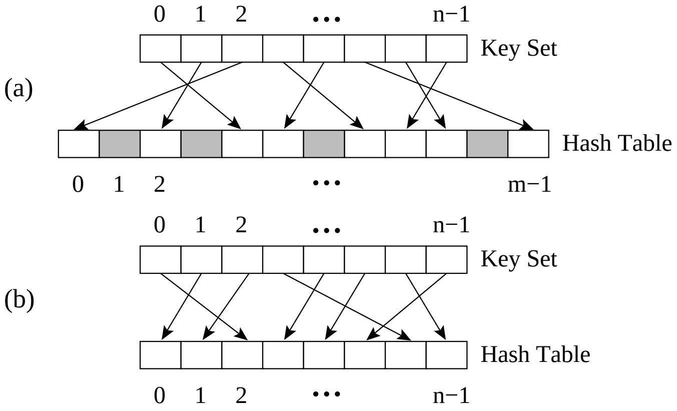


Figure 1.1: (a) Perfect hash function (b) Minimal perfect hash function.

Since PHFs and MPHFs are collision free, each key can be retrieved from the table with a single probe. MPHFs completely avoid the problem of wasted space and time. Better still, it was observed in [56] that MPHFs also avoid cache misses that arise due to collision resolution schemes like open addressing and chaining [51].

Minimal perfect hash functions are used for memory efficient storage and fast retrieval of items from static sets, such as words in natural languages, reserved words in programming languages or interactive systems, item sets in data mining techniques [21, 22], routing tables and other network applications [66], sparse spatial data [54], graph compression [7] and, to represent large web maps [27].

A PHF depends on the set S of distinct key values that occur. It is known that maintaining a PHF dynamically under insertions into S is only possible using space that is super-linear on n [28]. However, in this work we consider the case where S is fixed, and construction of a PHF can be done as part of the preprocessing of data (e.g., in a data warehouse).

To the best of our knowledge, previously perfect hashing methods have not been able to generate functions for realistic data sizes that require a constant number of bits to store the functions. All previous methods suffer from either an incomplete theoretical understanding (so there is no guarantee that they work well on a given data set) or seems impractical due to a very intricate and time-consuming evaluation procedure.

Until now, because of the limitations of current algorithms, the use of MPHFs is restricted to scenarios where the key set being hashed is relatively small. However, in many cases the demand to deal in an efficient way with very large key sets is growing. For instance, search engines are nowadays indexing tens of billions of pages and algorithms like PageRank [16], which uses the web graph to derive a measure of popularity for web pages, would benefit from an MPHF to map long URLs to smaller integer numbers that are used as identifiers to web pages, and correspond to the vertex set of the web graph.

Though there has been considerable work on how to construct good PHFs, there is a gap between theory and practice among all previous methods on perfect hashing. On one hand, there are good theoretical results without experimentally proven practicality for large key sets. On the other hand, there are the algorithms that assume unrealistic assumptions to theoretically analyze their run time and space usage.

In this thesis we present new algorithms for constructing PHFs and MPHFs that outperform the main practical algorithms available in the literature and are theoretically well-understood. Therefore we give an important step in the way of bridging the gap between theory and practice on perfect hashing. We also show that the new algorithms have made it viable to use MPHFs for applications that was not possible in the past. The algorithm we propose to construct MPHFs can easily scale to billions of entries.

In Section 1.2 we present some definitions and notation used throughout this work. In Section 1.3 we present the information theoretical lower bound to describe PHFs and MPHFs. In Section 1.4 we present two important concepts used in the analysis of hashing schemes. In Section 1.5 we disscuss some facts on random graphs used to analyze the algorithms designed in this work. In Section 1.6 we present the main results available in the literature on perfect hashing and also discuss the aforementioned gap between theory and practice on perfect hashing. In Section 1.7 we present our objectives and a technical overview of this work. In Section 1.8 we present the main contributions of this work. Finally, in Section 1.9 we present the road map of this thesis.

1.2 Definitions and Notation

The aim of this section is to establish a common vocabulary to be used throughout this work.

Definition 1 A *key* is made up by symbols from a finite and ordered alphabet Σ of size $|\Sigma|$.

Definition 2 Let Φ denote the *maximum key length*. Then $L = \Phi \log |\Sigma|$ is the maximum key length in bits[1]. Then we define a *key universe* U of size $u = 2^L$.

Throughout this thesis we consider that $L = O(1)$ and that $\log u$ fits in $O(1)$ computer words. Therefore, all algorithms we will consider are analyzed for the *Word RAM* model of computation [41]. In this model an element of the universe U fits into one machine word, and arithmetic operations and memory accesses have unit costs.

Definition 3 Let S be a subset of U containing n keys, where $n \ll u$.

Definition 4 Let $h : U \to M$ be a *hash function* that maps the keys from U to a given interval of integers $M = [0, m-1] = \{0, 1, \ldots, m-1\}$ (i.e., given a key $x \in U$, the hash function h computes an integer in $[0, m-1]$).

Definition 5 Given two keys $x, y \in U$, where $x \neq y$, and a hash function $h : U \to M$, a *collision* occurs when $h(x) = h(y)$.

Definition 6 A perfect hash function $\mathit{phf} : S \to M$ is an injection on $S \subseteq U$(i.e., for all pair s_1, $s_2 \in S$ such that $s_1 \neq s_2$, then $\mathit{phf}(s_1) \neq \mathit{phf}(s_2)$, where $m \geq n$). For being an injection, *phf* maps each key in S to a unique integer in M. As no collision occurs, if *phf* is used to index a hash table of size m with n records identified by the n keys in S, each record can be retrieved in one probe.

Definition 7 A minimal perfect hash function $\mathit{mphf} : S \to M$ is a bijection on $S \subseteq U$ (i.e., each key in S is mapped to a unique integer in M and $m = n$).

Definition 8 A perfect hash function is *order-preserving* if for any pair of keys s_i and $s_j \in S$ then $\mathit{phf}(s_i) < \mathit{phf}(s_j)$ if and only if $i < j$.

[1] Throughout this work we denote $\log_2 x$ as $\log x$.

1.3 The Information Theoretical Lower Bound to Describe PHFs and MPHFs

One of the most important metrics related to PHFs and MPHFs is the amount of space required to describe a function. The information theoretical lower bound to describe a PHF was first studied in [57]. Fredman and Komlós [40] proved a lower bound for MPHFs. A simpler proof of this was later given in [68]. The following two theorems present the information theoretical lower bound to describe a PHF and an MPHF, respectively. Here we use Stirling's approximation and so we obtained a more precise result up to an additive constant, because Stirling's approximation is tight within a constant factor. For simplicity of exposition, we consider in this thesis the case $\log u \ll n$, which allows us to ignore terms in the space usage that depend on u.

Theorem 1 Every perfect hash function $\mathit{phf} : S \to M$, where $|S| = n$ and $|M| = m$, requires at least $(1 + (m/n - 1 + 1/2n)\ln(1 - n/m))\, n \log e$ bits to be stored.

Proof. The probability of randomly mapping n elements into a range of size m without collisions (i.e., probability of getting a PHF) is:

$$\Pr_{ph}(n,m) = \frac{(m-1)(m-2)\ldots(m-n+1)}{m^n} = \frac{m!}{m^n(m-n)!}$$

By using Stirling's approximation $n! \approx n^n e^{-n}\sqrt{2\pi n}$ we obtain:

$$\Pr_{ph}(n,m) \quad \approx \quad m^{(m-n)} \cdot (m-n)^{-(m-n)} \cdot e^{-n}\sqrt{\frac{m}{m-n}}$$

Therefore, at least $1/\Pr_{ph}(n,m)$ hash functions are required to obtain a PHF. Thus, at least $\log(1/Pr_{ph}(n,m)) = (1 + (m/n - 1 + 1/2n)\ln(1 - n/m))\, n \log e$ bits are required to encode that set of hash functions. □

Theorem 2 Every minimal perfect hash function $\mathit{mphf} : S \to M$, where $|S| = n$ and $|M| = m = n$, requires at least $n \log e - O(\log n)$ bits to be stored.

Proof. The probability of finding an MPHF (where $n = m$) is:

$$\Pr_{mph}(n,n) = \frac{n!}{n^n} = \frac{n^n\sqrt{2\pi n}}{n^n e^n} = e^{-n}\sqrt{2\pi n}$$

which also uses the aforementioned Stirling's approximation. Therefore, the expected number of bits needed to describe these rare minimal perfect hash functions is at least $\log(1/\Pr_{mph}(n,n)) = n \log e - O(\log n)$. □

1.4 Uniform Hashing Versus Universal Hashing

All perfect hashing algorithms need to use hash functions chosen uniformly at random from a fixed family $\mathcal{H}$ of hash functions for constructing PHFs or MPHFs. There are two families of hash functions used in the classical analysis of hashing schemes: (i) uniform hash functions and (ii) universal hash functions. In this section we define these two families of hash functions.

1.4.1 Family of Uniform Hash Functions

The classic analysis of hashing schemes often entails the assumption that the hash functions used are uniformly chosen at random from a family of uniform hash functions, defined as follows.

Definition 9 Let $\mathcal{H}$ be the family of all m^u possible hash functions from U to $[0, m-1]$. A *uniform hash function* is a function that has independent function values and is uniformly chosen at random from $\mathcal{H}$.

The problem with uniform hash functions is the space required to describe a single function, which is $\Omega(u \log m)$ bits. This space requirement usually far exceeds the available storage and is often overlooked in the analysis of practical perfect hashing schemes available in the literature.

Lemma 1 [20] Let $\mathcal{H}$ be a family of uniform hash functions and $h : U \to M$ be a hash function taken from $\mathcal{H}$ with probability $\frac{1}{|\mathcal{H}|}$. Let $C_h(x, y) = 1$ if $x \in U$ and $y \in U$ collide by using a hash function h and 0 otherwise, where $x \neq y$. The probability of collision between two different keys $x, y \in U$ corresponds to the expected value of $C_h(x, y)$ and is given by:

$$E[C_h(x, y)] \geq \frac{1}{m} - \frac{1}{u}$$

Proof. Let $C_h(x, U)$ denote the total number of keys in U that collides with a given key $x \in U$ by using a hash function h. So, $C_h(x, U) = \sum_{y \in U, y \neq x} C_h(x, y)$. Let $C_h(U, U)$ denote the total number of collisions for all $x \in U$ by using a hash function h. So, $C_h(U, U) = \sum_{x \in U} C_h(x, U)$. Let $\mathcal{H}$ be a family or a collection of hash functions. Thus, $C_{\mathcal{H}}(U, U) = \sum_{h \in \mathcal{H}} C_h(U, U)$ denotes the total number of collisions for all $x \in U$ and for all hash functions from $\mathcal{H}$. Let us think of $M = [0, m-1]$ as a range of indexes of a hash table with m buckets and the values in M are computed by a hash function $h : U \to M$

taken with probability $\frac{1}{|\mathcal{H}|}$ from a family $\mathcal{H}$ of uniform hash functions. After mapping all keys to the range M, if a bucket $i \in M$ has three keys $\{k_1, k_2, k_3\}$, then k_1 collides with each of $\{k_2, k_3\}$, k_2 collides with each of $\{k_1, k_3\}$, and k_3 collides with each of $\{k_1, k_2\}$, so we have 6 collisions in bucket i. In the worst case, when all keys from U are mapped to the same bucket i, this corresponds to the number of ordered pairs we can form from the key universe U of size u considering a hash function $h \in \mathcal{H}$, which is given by $C_h(U, U) = u^2 - u$. Therefore, $C_{\mathcal{H}}(U, U) = |\mathcal{H}|(u^2 - u)$. As we have m buckets, then the expected number of collisions for all hash functions in $\mathcal{H}$ is:

$$E[C_{\mathcal{H}}(U, U)] = u^2 |\mathcal{H}| \left(\frac{1}{m} - \frac{1}{mu} \right)$$

Thus, by the pigeon hole principle[2] there exists $x, y \in U$ and $h \in \mathcal{H}$ such that

$$E[C_h(x, y)] = \frac{1}{m} - \frac{1}{mu} \geq \frac{1}{m} - \frac{1}{u}$$

□

1.4.2 Family of Universal Hash Functions

As mentioned in Section 1.4.1, the amount of space to represent a uniform hash function is prohibitive in practice. Fortunately in most cases heuristic hash functions behave very closely to the expected behavior of uniform hash functions, but there are cases when rigorous probabilistic guarantees are necessary [18]. For instance, various adaptive hashing schemes presume that a hash function with certain prescribed properties can be found in constant expected time. This holds if the function is chosen uniformly at random from all possible functions until a suitable one is found but not necessarily if the search is limited to a smaller set of functions. This situation has led Carter and Wegman [20] to the concept of universal hashing.

Definition 10 A family of hash functions $\mathcal{H}$ is defined as *weakly universal* or just *universal* if for any pair of distinct elements $x_1, x_2 \in U$ and h chosen uniformly at random from $\mathcal{H}$ then

$$\Pr(h(x_1) = h(x_2)) \leq \frac{1}{m}\cdot$$

[2] The pigeonhole principle states that, given two natural numbers n and m with $n > m$, if n pigeons are put into m pigeonholes, then at least one pigeonhole must contain more than one pigeon.

Definition 11 A family of hash functions $\mathcal{H}$ is defined as *strongly universal or pair-wise independent* if for any pair of distinct elements $x_1, x_2 \in U$ and arbitrary $y_1, y_2 \in M$ then

$$\Pr(h(x_1) = y_1 \text{ and } h(x_2) = y_2) = \frac{1}{m^2}.$$

It turns out that in many situations the analysis of various hashing schemes can be completed under the weaker assumption that h is chosen uniformly at random from a family of universal hash functions, rather than the assumption that h is chosen uniformly at random from all possible hash functions. In other words, limited randomness suffices in practice [70]. For instance, when we are hashing a key universe much larger than the hash function range $M = [0, m-1]$, which is the case for most hashing applications, universal hash functions behave very closely to the expected behavior of uniform hash functions. This can be seen by comparing the result of Lemma 1 with the probability of collision for universal hash functions, which is given in Definition 10. We notice that there are cases where rigorous probabilistic guarantees are necessary [18, 2]. Let us illustrate this with the following three scenarios, which have been extensively used in various settings and were reported in [2].

1. Consider that a key set $S \subseteq U$ of size n is hashed to m buckets. The question is: how many buckets m are needed to get no collisions? By using a universal hash function we need $m = O(n^2)$ to get no collisions with probability more than 1/2. By using a uniform hash function, it is well known that $o(n^2)$ is not enough to get no collisions, as exemplified by the birthday paradox[3]. Therefore, nothing is lost by using a universal hash function in this scenario.

2. Consider that the key set $S \subseteq U$ is hashed to $m = n$ buckets. The question is: what is the size of S to cover all buckets (i.e., no bucket is left empty)? By using a universal hash function, if the size of S is $2n^2$, then, all buckets are covered with probability more than 1/2. By using a uniform hash function, it is well known that a key set S of size $\theta(n \log n)$ would be enough to cover all buckets with high probability[4]. Therefore, by using a uniform hash function in this scenario, a polynomial gain is obtained by going from $O(n^2)$ to $\theta(n \log n)$.

3. Consider that the key set S of size n is hashed to $m = n$ buckets. The question is: what is the size of the largest bucket? By using a universal hash function, the largest

[3] The *birthday paradox* says that if 23 or more people are grouped together at random, the probability that at least two people have a common birthday exceeds 50%, as can be seen in Feller [36, Page 33].

[4] Throughout this thesis we write "with high probability" to mean with probability $1 - n^{-\delta}$ for $\delta > 0$.

bucket will contain $O(n^{1/2})$ keys. By using a uniform hash function, it is well known that the largest bucket will contain $\theta(\log n/\log\log n)$ keys. Therefore, by using a uniform hash function in this scenario, it is obtained an exponential gain by going from $O(n^{1/2})$ to $\theta(\log n/\log\log n)$.

1.5 Random Graphs

We now discuss some facts on random graphs that are important for analyzing our algorithms. A random graph is a graph generated by some random procedure. There are many non-equivalent ways to define random graphs and now we present two closely related models. The study of random graphs goes back to the classical work of Erdős and Rényi [33, 34, 35] (for a modern treatment, see [8, 49]).

Definition 12 Let $G = (V, E)$ be a random graph in the uniform model $\mathcal{G}(m, n)$, the model in which all the $\binom{\binom{m}{2}}{n}$ graphs on V with n edges are equiprobable. In this model, graph G starts with a fixed number of vertices $|V| = m$ and $|E| = n$ edges are randomly chosen without replacement from the set of all $\binom{m}{2}$ possible edges. A similar model, denoted by $\mathcal{G}(m, p)$, where $0 \leq p \leq 1$, is obtained by taking the same vertex set but now each edge is selected with probability p and independently of all other edges and therefore repetitions are allowed.

As presented in [48], it is often useful to regard the random graph as evolving in time by a stochastic process, starting with a vertex set without edges and then inserting edges until the complete graph is obtained. For instance, the process of adding each edge independently of the others at some random time, for example, uniformly distributed in the range $(0, 1)$, will give a random graph of type $\mathcal{G}(m, p)$ at a fixed time $p \in (0, 1)$ and a random graph of type $\mathcal{G}(m, n)$ at the random time at which the n-th edge appears.

Our best result generates a family $\mathcal{F}$ of PHFs or MPHFs based on random acyclic r-partite hypergraphs, defined as follows.

Definition 13 A hypergraph is the generalization of a standard undirected graph where each edge connects $r \geq 2$ vertices.

Definition 14 Let $G_r = (V, E)$ be a random r-partite r-uniform hypergraph for $r \geq 2$, where V is a disjoint union of the r parts $V_0, V_1, \ldots, V_{r-1}$, $|V_i| = \rho$, $|V| = m = r\rho$, and

$|E| = n$. The edges are inserted into G_r one at a time, each being picked at random from all ρ^r possible edges, allowing repetitions.

Definition 15 A hypergraph is *acyclic* if and only if some sequence of repeated deletions of edges containing vertices of degree 1 yields a hypergraph without edges [26, Page 103].

1.6 Related Work

In this section we review some of the most important theoretical, practical, and heuristic results on perfect hashing. Czech, Havas and Majewski [26] provided a more comprehensive survey until 1997.

As mentioned before, there is a gap between theory and practice among minimal perfect hashing methods. On one hand, there are good theoretical results without experimentally proven practicality for large key sets. We will argue below that these methods are indeed not practical. On the other hand, there are two categories of practical algorithms: the theoretically analyzed time and space usage algorithms that assume uniform hash functions for their methods, which is an unrealistic assumption because each uniform hash function $h : U \to [0, m-1]$ require at least $u \log m$ bits of storage space, and the heuristic algorithms that present only empirical evidences. The aim of this section is to discuss the existent gap among these three types of algorithms available in the literature.

1.6.1 Theoretical Results

In this section we review some of the most important theoretical results on minimal perfect hashing, which do not assume that uniform hash functions are available for free. Fredman and Komlós [40] proved that at least $n \log e + \log \log u - O(\log n)$ bits are required to represent an MPHF (in the worst case over all sets of size n), provided that $u \geq n^\alpha$ for some $\alpha > 2$. Mehlhorn [57] showed that the Fredman-Komlós bound is almost tight by providing an algorithm that constructs an MPHF that can be represented with at most $n \log e + \log \log u + O(\log n)$ bits. However, his algorithm is far from practice because its generation and evaluation time are exponential on n (i.e., $n^{\theta(ne^n u \log u)}$).

Schmidt and Siegel [70] proposed the first algorithm for constructing an MPHF with constant evaluation time and description size $O(n + \log \log u)$ bits. Their algorithm, as well as all other algorithms we will consider, is for the Word RAM model of computation [41] (see Section 1.2). From a practical point of view, Schmidt and Siegel's algorithm is not attractive. The scheme is complicated to implement and the constant of the space bound is large: For a set of n keys, at least $29n$ bits are used, which means a space usage similar in practice to the best schemes using $O(n \log n)$ bits. Though it seems that [70] aims to describe its algorithmic ideas in the clearest possible way, not trying to optimize the constant, it is hard to improve the space usage significantly.

More recently, Hagerup and Tholey [43] have come up with the best theoretical result we know of. The MPHF obtained can be evaluated in $O(1)$ time and stored in $n \log e + \log\log u + O(n(\log\log n)^2/\log n + \log\log\log u)$ bits. The generation time is $O(n+\log\log u)$ using $O(n)$ words of space. In spite of its theoretical importance, the Hagerup and Tholey algorithm also is not practical, as it only emphasizes asymptotic space complexity. (It is also very complicated to implement, but we will not go into that.) For $n < 2^{150}$ the scheme is not well-defined, as it relies on splitting the key set into buckets of size $\hat{n} \leq \log n/(21 \log\log n)$. If we fix this by letting the bucket size be at least 1, then buckets of size one will be used for $n < 2^{300}$, which means that the space usage will be at least $(3 \log\log n + \log 7)\, n$ bits. For a set of a billion keys, this is more than 17 bits per element. Since 2^{300} exceeds the number of atoms in the known universe, it is safe to conclude that the Hagerup-Tholey MPHF is not space efficient in practical situations. While we believe that their algorithm has been optimized for simplicity of exposition, rather than constant factors, it seems difficult to significantly reduce the space usage based on their approach.

1.6.2 Practical Results

We now describe some of the main “practical” results upon which our work is based. They are characterized by simplicity and (provably) low constant factors. In general, they are analyzed upon the unrealistic assumption that uniform hash functions are available for free.

The algorithm proposed by Czech, Havas and Majewski [25] assumes uniform hash functions to be available for free (i.e., they use universal hash functions) to construct order preserving MPHFs. The method uses two uniform hash functions $h_1 : S \to [0, cn-1]$ and $h_2 : S \to [0, cn-1]$ to generate MPHFs in the following form: $\mathit{mphf}(x) = (g[h_1(x)] + g[h_2(x)] \bmod n$ where $c > 2$. The resulting MPHFs can be evaluated in $O(1)$ time and stored in $O(n \log n)$ bits (that is optimal for an order preserving MPHF). The resulting MPHF is generated in expected $O(n)$ time.

Botelho, Kohayakawa and Ziviani [12] improved the space requirement at the expense of generating functions in the same form that are not order preserving. Their algorithm is also linear on n, but runs faster than the ones by Czech et al [25] and the resulting MPHFs are stored using half of the space because $c \in [0.93, 1.15]$. However, the resulting MPHFs still need $O(n \log n)$ bits to be stored. It was found experimentally in [12] that their generation procedure works well in practice.

Majewski et al [55] proposed an algorithm to generate a family of MPHFs based on r-uniform hypergraphs (i.e., with edges of size r). It is a generalization of the algorithm in [25]. The resulting functions can be evaluated in $O(1)$ time and stored in $O(n \log n)$ bits. Although the resulting functions are almost as compact as the ones generated by the work in [12], they still require $O(n \log n)$ bits to be stored. Botelho, Pagh and Ziviani [14] designed a family of algorithms that improves the space requirement from $O(n \log n)$ to $O(n)$ bits at the expense of generating functions that are not order preserving.

Since the space requirements for uniform hash functions makes them unsuitable for implementation, one has to settle for a more realistic setup. The first step in this direction was given by Pagh [61]. He proposed a family of randomized algorithms for constructing MPHFs of the form $mphf(x) = (f(x) + d[g(x)]) \bmod n$, where f and g are chosen from a family of universal hash functions (see Definition 10) and d is a set of displacement values to resolve collisions that are caused by the function f. Pagh identified a set of conditions concerning f and g and showed that if these conditions are satisfied, then a minimal perfect hash function can be computed in expected $O(n)$ time and stored in $(2 + \epsilon)n \log n$ bits, which is suboptimal.

Dietzfelbinger and Hagerup [29] improved [61], reducing the space usage to $(1+\epsilon)n \log n$ bits, but in their approach f and g must be chosen from a class of hash functions that meet additional requirements. Woelfel [75] has shown how to decrease the space usage further, to $O(n \log \log n)$ bits asymptotically, still with a quite simple algorithm. However, there is no empirical evidence on the practicality of this scheme.

Galli, Seybold and Simon [42] proposed an algorithm to generate MPHFs similar to the ones generated in the works [61, 29]. However, in their MPHFs the two functions f and g are defined as $f(x) = h_c(x) \bmod n$ and $g(x) = \lfloor h_c(x)/n \rfloor$, where $h_c(k) = (ck \bmod p) \bmod n^2$, $1 \leq c \leq p - 1$ and p is a prime larger than u. The resulting MPHFs are generated in linear time and stored in $O(n \log n)$ bits. The main advantage of their approach is that it can be easily adapted for dynamic key sets, but just for PHFs.

Prabhakar and Bonomi [66] designed perfect hash functions to be used for storing routing tables in routers for networking applications. They have shown that the storage requirement for the resulting functions goes to $2en$ when n goes to infinity. In their simulations the resulting functions were stored in $8.6n$ bits. The main advantage of their scheme is that it is simple enough to be implemented in hardware.

Randomized algorithms of *Las Vegas*[5] type were designed in all previous work and also in this work. Conversely, the works [4, 73] present deterministic algorithms to construct PHFs and MPHFs. The resulting functions require $O(n\log(n)+\log(\log(u)))$ bits of storage space and are evaluated in $O(\log(n)+\log(\log(u)))$. Thus, the resulting functions are not evaluated in $O(1)$ time and are within a factor of $\log n$ bits from the information theoretical lower bounds to describe PHFs and MPHFs, which are presented in Theorems 1 and 2, respectively. The average and worst case complexity of the algorithms are $O(n\log(n)\log(\log(u)))$ and $O(n^3\log(n)\log(\log(u)))$, respectively.

1.6.3 Heuristics

In this section we consider works designed for specific applications and, in general, just experimental evidences of the behavior of the algorithms are provided.

Fox et al. [39] created the first scheme with good average-case performance for large datasets, i.e., $n \approx 10^6$. They have designed two algorithms, the first one generates an MPHF that can be evaluated in $O(1)$ time and stored in $O(n\log n)$ bits. The second algorithm uses quadratic hashing and adds branching based on a table of binary values to get an MPHF that can be evaluated in $O(1)$ time and stored in $c(n+1/\log n)$ bits. They argued that c would be typically lower than 5, however, it is clear from their experimentation that c grows with n and they did not discuss this. They claimed that their algorithms would run in linear time, but, it is shown in [26, Section 6.7] that the algorithms have exponential running times in the worst case, although the worst case has small probability of occurring.

Fox, Chen and Heath [38] improved the above result to get a function that can be stored in cn bits. The method uses four uniform hash functions $h_{10} : S \rightarrow [0, n-1]$, $h_{11} : [0, p_1 - 1] \rightarrow [0, p_2 - 1]$, $h_{12} : [p_1, n-1] \rightarrow [p_2, b-1]$ and $h_{20} : S \times \{0,1\} \rightarrow [0, n-1]$ to construct an MPHF that has the following form:

$$\begin{aligned} \mathit{mphf}(x) &= (h_{20}(x,d) + g(i(x))) \bmod n \\ i(x) &= \begin{cases} h_{11} \circ h_{10}(x) & \text{if } h_{10}(x) < p_1 \\ h_{12} \circ h_{10}(x) & \text{otherwise.} \end{cases} \end{aligned}$$

[5] A random algorithm is Las Vegas if it always produces correct answers, but with a small probability of taking a long time to execute.

where $p_1 = 0.6n$ and $p_2 = 0.3n$ were experimentally determined, and $b = \lceil cn/(\log n + 1) \rceil$. Again c is only established for small values of n. It could very well be that c grows with n. So, the limitation of the three algorithms is that there is no warranty that the number of bits per key to store the function will be fixed as n increases.

The work by Lefebvre and Hoppe [54] has the same problem of not providing any warranty that the storage space of the resulting functions will be a constant number of bits per key. They have designed a PHF method to specifically represent sparse spatial data and the resulting functions require more than 3 bits per key to be stored. In the same trend, Chang, Lin and Chou [21, 22] have designed MPHFs tailored for mining association rules and traversal patterns in data mining techniques.

1.7 Technical Overview of this Work

Our primary objective was to design perfect hashing algorithms that are theoretically well-founded and can be efficiently used in practice. For that we investigate ways to bridge the existent gap between theory and practice among the minimal perfect hashing algorithms available in the literature.

In this work we used a two-step approach in order to design an algorithm that achieves our primary objective. In the first step, we partition the input key set into small buckets. This step is equivalent to the process of generating runs in an external multi-way merge sort, which is carefully engineered to make it work in linear time. In the second step, we generate a PHF or an MPHF for each bucket.

Figure 1.2 illustrates the two steps of the algorithm: the *partitioning step* and the *searching step*. The partitioning step takes a key set S of size n and uses a hash function h_0 to partition S into N_b buckets. The searching step generates an MPHF (or equivalently a PHF) for each bucket i, $0 \leq i \leq N_b - 1$, and computes the *offset* array. The evaluation of the resulting MPHF for a key x is:

$$\mathit{MPHF}(x) = \mathit{MPHF}_i(x) + \mathit{offset}[i]$$

where $i = h_0(x)$ indicates the bucket where key x is, $\mathit{MPHF}_i(x)$ is the position of x in bucket i, and $\mathit{offset}[i]$ gives the total number of entries before bucket i in the hash table.

If the key set size n fits in the internal memory available, then the first step of the algorithm is not necessary. In this situation, we just make the bucket size equal to the input size n and generate a PHF or an MPHF for this bucket. Therefore, the algorithm

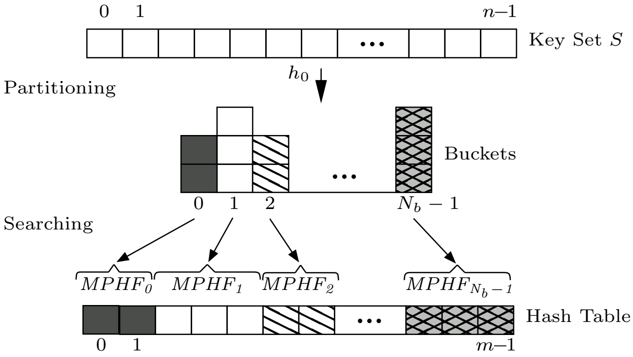


Figure 1.2: The two steps of the algorithm.

becomes an internal random access memory algorithm, referred to as *RAM algorithm* from now on. If the key set size n is larger than the size of the internal memory available, then the first step is performed to partition the input set into small buckets and the algorithm becomes an external memory algorithm, referred to as *EM algorithm* from now on. The external algorithm is also cache-aware because the buckets are small enough to fit in the CPU cache. Therefore, the EM algorithm accesses memory in a less random fashion when compared with the RAM algorithm.

We refine and combine a number of existing techniques in the design and implementation of the algorithm, as follows:

1. To generate PHFs or MPHFs for the buckets we could choose from a number of alternatives, emphasizing either space usage, construction time, or evaluation time. All methods that assume uniform hash functions can be made to work, by using the *split-and-share* technique presented in [30] to split the problem into small buckets, and simulate uniform hash functions on each bucket. In Chapter 3 we present a particular engineering of this idea, with a refinement that, without extra space usage, gives a *family* of uniform hash functions on each bucket.

2. The RAM algorithm is used to compute PHFs or MPHFs on the buckets because it generates near-space optimal functions and outperforms the main practical algorithms available in the literature, including our previous result presented in [12]. We take a PHF generation implicit in [23] as a starting point, which was also independently suggested by Belazzougui [5]. Then, we improve the analysis, refine the generation algorithm to make it succeed with high probability, extend it to generate MPHFs as well, and show how to implement everything in a near space-optimal manner. When the key set fits in the internal memory we have just one bucket of size

n, otherwise several small buckets are handled. The RAM algorithm is presented in Chapter 2.

3. External sorting (see, e.g., [74, 53]) is used to group the keys into buckets when the key set does not fit in the internal memory. Then, we handle each bucket separately. The important insight here is that we split the problem in *small* buckets and this has both theoretical and practical implications. From the theoretical point of view we showed that, by refining the split-and-share technique to simulate uniform hash functions on the small buckets, we were able to prove that the EM algorithm will work for every key set with high probability. From the practical point of view, an important feature of this is that we may make buckets that are small enough to fit in the CPU cache, resulting in a significant speedup (in processing time per element) compared to other methods. To generate the runs of the external memory sorting, we use radix sorting [24] to perform this in linear time.

   Offset tables are used to put everything together to a single PHF or MPHF. This has been done in several theoretical works (see, e.g. [70, 43]). In Chapter 4 we show how to implement this with low space overhead in practice and present the EM algorithm.

4. The EM algorithm has a high degree of parallelism because it is based on the external multi-way merge sort algorithm. In Chapter 5 we exploit this fact to design a parallel version of the EM algorithm.

5. The techniques designed in our previous work presented in [12] to generate MPHFs based on random graphs with cycles were used to optimize one version of the RAM algorithm presented in Chapter 2. This is presented in Chapter 6.

1.8 Contributions

The attractiveness of using PHFs and MPHFs depends on the following issues [43]:

1. The amount of CPU time required for generating the functions.

2. The space requirements for generating the functions.

3. The amount of CPU time required by the functions for each retrieval.

4. The space requirements of the description of the resulting functions to be used at retrieval time.

No previously known algorithm performs well for all these requirements. Usually, the space requirement for generating the functions is overlooked. That is why the algorithms in the literature cannot scale for key sets on the order of billions of keys. Also, as mentioned before, there is a gap between practical and theoretical algorithms. On one hand, practical algorithms analyze the space requirement to describe the resulting functions under the unrealistic assumption that uniform hash functions are available to be used with no extra cost of space. On the other hand, the theoretical algorithms are analyzed with no unrealistic assumption, but they emphasize asymptotic space complexity and are too complicated to implement.

The main contributions of this thesis are:

1. We present a simple, practical and highly scalable perfect hashing algorithm that takes into account the four requirements aforementioned. When the input key set fits in main memory, it becomes an internal random access memory algorithm (RAM algorithm); otherwise, it becomes an external memory algorithm (EM algorithm). Preliminary versions of the RAM and the EM algorithms were presented in [14] and in [15], respectively. We now present more details on the two algorithms.

   (a) The RAM algorithm works on random acyclic r-partite hypergraphs given by function values of uniform hash functions on the keys of S. The idea of basing perfect hashing on random acyclic hypergraphs is not new, see e.g. [55], but we proceed differently to achieve a space usage of $O(1)$ bits per key rather than $O(\log n)$ bits per key, reducing the complexity order to store the functions from $O(n \log n)$ to $O(n)$ bits. The RAM algorithm is presented in Chapter 2.

   We now comment on the four aforementioned requirements:

      i. It generates PHFs or MPHFs in linear time. The PHFs are equivalent to the ones suggested by Belazzougui [5], which were previously suggested in a more general way by Chazelle et al in [23].

      ii. It requires $O(n)$ computer words to generate PHFs or MPHFs. That is why it is appropriated for key sets that can be handled in internal memory.

      iii. It generates PHFs or MPHFs that take $O(1)$ time to be evaluated.

      iv. It generates near space-optimal PHFs and MPHFs. The space requirements of the description of the resulting functions depend on the relation between m and n. For $m = n$, the space usage is approximately $2.62n$ bits. For $m = 1.23n$, the space usage is approximately $1.95n$ bits. In all cases, this is within a small constant factor from the information theoretical minimum of approximately $1.44n$ bits for MPHFs and $0.89n$ bits for PHFs, something that has not been achieved by previous practical algorithms.

(b) The EM algorithm uses a number of techniques from the literature to allow the generation of PHFs or MPHFs for sets on the order of billions of keys. It increases one order of magnitude in the size of the greatest key set for which an MPHF was obtained in the literature [12]. This improvement comes from a combination of a novel, theoretically sound perfect hashing scheme that greatly simplifies previous methods, and the fact that it is designed to make good use of the memory hierarchy, since it is fundamentally a divide-to-conquer technique.

The EM algorithm is the first step aiming to bridge the gap between theory and practice on perfect hashing. Therefore, the EM algorithm is the first algorithm that can be used in practice, has time and space usage carefully analyzed without unrealistic assumptions, and scales for billions of keys.

We demonstrate the scalability of the EM algorithm by generating an MPHF for a set of 1.024 billion URLs from the World Wide Web of average length 64 characters in approximately 50 minutes, using a commodity PC. The EM algorithm is presented in Chapter 4.

We now comment on the four aforementioned requirements:

i. It generates PHFs or MPHFs in linear time and the dominating step in the generation algorithm consists of sorting n fingerprints of $O(\log n)$ bits.

ii. It requires $O(n^\epsilon)$ computer words to have linear time complexity, where $0 < \epsilon < 1$. This is because it only needs a heap in main memory to multi-way merge files stored on disk, and the size of the heap is the relation between the size of the input key set and the amount of the internal memory available, both in bytes. In our case, as we want to perform the merge operation in one pass, we need $\epsilon = 0.5$ (see, e.g., [1, Theorem 3.1]). This is one of the reasons that enables the EM algorithm to scale for sets on the order of billions of keys.

iii. It generate PHFs or MPHFs that take $O(1)$ time to be evaluated.

iv. It also generates near space-optimal PHFs and MPHFs, but now we do not assume that uniform hash functions are available with no additional cost of space. For that we designed in Chapter 3 a way of simulating uniform hash functions on the small buckets with only a constant factor space overhead. This enabled us to use the RAM algorithm to build the MPHFs of each bucket without unrealistic assumptions. As for the RAM algorithm, the space requirements of the description of the resulting functions also depend on the relation between m and n. For $m = n$, the space usage is approximately $3.3n$ bits. For $m = 1.23n$, the space usage is approximately $2.7n$ bits. Again, this is within a small constant factor from the information theoretical minimum for PHFs and MPHFs, something that has not been achieved by previous practical and theoretical algorithms, except asymptotically for very large n.

2. We provide a scalable parallel implementation of the EM algorithm, referred to as *Parallel External Memory* (PEM) algorithm from now on. The PEM algorithm allows to distribute the construction, description and evaluation of the resulting functions. For instance, using a 14-computer cluster the parallel EM generates an MPHF for 1.024 billion URLs in approximately 4 minutes, achieving an almost linear speedup. Also, for 14.336 billion 16-byte random integers evenly distributed among the 14 participating machines the PEM algorithm outputs an MPHF in approximately 50 minutes, resulting in a performance degradation of 20%. To the best of our knowledge there is no previous result in the perfect hashing literature that can be implemented in a parallel way to obtain better scalability and performance than the results presented by the PEM algorithm. The PEM algorithm is presented in Chapter 5. A preliminary version of the PEM algorithm was presented in [11].

3. We present techniques that allow the generation of PHFs and MPHFs based on random graphs containing cycles. A preliminary result was presented in [12]. It improved the space requirement of the algorithm by Czech, Havas and Majewski [25] at the expense of generating functions in the same form that are not order preserving. Both algorithms are linear on n, but our algorithm runs 59% faster than the one in [25], and the resulting MPHFs are stored using half of the space.

   However, the resulting MPHFs still need $O(n \log n)$ bits to be stored. As in [25], the algorithm assumes uniform hashing and needs $O(n)$ computer words of the Word RAM model to construct the functions. Recently, using ideas similar to the ones presented in [12], we have optimized the version of the RAM algorithm that works on random bipartite graphs to output the resulting functions 40% faster when cycles are allowed. These results are presented in Chapter 6.

4. We show that the PHFs and MPHFs designed in this thesis can now be used for applications in which they were not considered a good option in the past. This is a consequence of the fact that the resulting functions need $O(1)$ number of bits per key to be stored. In Chapter 7 we show that MPHFs provide the best trade-off between space usage and lookup time when compared to other hashing schemes. A preliminary version of this result was presented in [13].

5. Finally, we have created the C Minimal Perfect Hashing Library that is available at `http://cmph.sf.net` under the GNU Lesser General Public License (LGPL). The library was conceived for two reasons. First, we would like to make available our algorithms to test their applicability in practice. Second, we realized that there was a lack of similar libraries in the open source community. We have received very good feedbacks about the practicality of the library. For instance, it has received more than $2,482$ downloads (August 2008) and is incorporated by two Linux distributions: Debian[6] and Ubuntu[7].

1.9 Road Map

This text is organized as follows: Chapter 2 presents the internal random access memory algorithm (RAM algorithm), which generates a family of near space-optimal PHFs or MPHFs based on random acyclic r-partite r-uniform hypergraphs, for $r \geq 2$. Chapter 3 presents a way of simulating uniform hash functions on small key buckets. Chapter 4 presents the external memory algorithm (EM algorithm), which is the first algorithm that is theoretically well-understood and can be applied to sets on the order of billion keys. Chapter 5 presents a parallel version of the EM algorithm. Chapter 6 shows how to generate PHFs or MPHFs based on random graphs with cycles. Chapter 7 presents applications

[6] Debian is a volunteer project to develop a GNU/Linux distribution, which is available at `http://www.debian.org`. Debian was started more than a decade ago and has since grown to comprise more than 1000 members with official developer status and many more volunteers and contributors. It has expanded to encompass over 20,000 "packages" of free and open source applications and documentation.

[7] The Ubuntu project, available at `http://www.ubuntu.com`, attempts to work with Debian to address the issues that keep many users from using Debian. Ubuntu provides a system based on Debian with frequent time-based releases, corporate accountability, and a more considered desktop interface. Ubuntu provides users with a way to deploy Debian with security fixes, release critical bug fixes, a consistent desktop interface, and to never be more than six months away from the latest version of anything in the open source world.

in which the use of PHFs and MPHFs became interesting as a consequence of the results of this work. Finally, Chapter 8 presents the conclusions and some suggestions regarding future steps to be taken in this research.

Chapter 2

The Internal Perfect Hashing Algorithm

In this chapter we present a simple and efficient internal random access memory algorithm (RAM algorithm) to generate a family $\mathcal{F}$ of near space-optimal PHFs[1] or MPHFs. Its name comes from the fact that the RAM algorithm does not take into account the memory hierarchy to optimize efficiency, as the one presented in Chapter 4 does. The RAM algorithm generates a family $\mathcal{F}$ of PHFs or MPHFs based on random acyclic r-partite hypergraphs (see Section 1.5) given by function values of r uniform random hash functions on S. It is designed for key sets that induce random acyclic r-partite hypergraphs that fit in the internal random access memory. The resulting PHFs and MPHFs are stored in near optimal space (i.e., $O(n)$ bits.) Acyclic random hypergraphs has been used in previous MPHF constructions [55], but we will proceed differently to achieve a space usage of $O(n)$ bits rather than $O(n \log n)$ bits, diminishing the complexity order to store the functions from $O(n \log n)$ to $O(n)$ bits. A previous version of the RAM algorithm was presented

[1] Chazelle et al [23] present a way of constructing PHFs that is equivalent to the ones presented in this chapter. It is explained as a modification of the "Bloomier Filter" data structure, but it is not explicit that a PHF is constructed. We have independently designed an algorithm to construct a PHF that maps keys from a key set S of size n to the range $[0, (2.0 + \epsilon)n - 1]$ based on random 2-graphs, where $\epsilon > 0$. The resulting functions require $2.0 + \epsilon$ bits per key to be stored. Belazzougui [5] suggested a method to construct PHFs that map to the range $[0, (1.23 + \epsilon)n - 1]$ based on random 3-graphs. The resulting functions are stored in 2.46 bits per key and this space usage was further improved to 1.95 bits per key by using arithmetic coding. Thus, the simple construction of a PHF described must be attributed to Chazelle et al [23]. The new contribution of this chapter is to analyze and optimize the constant of the space usage considering implementation aspects as well as a way of constructing MPHFs from those PHFs.

in [14].

This chapter is organized as follows. In Section 2.1 we describe the family $\mathcal{F}$ of PHFs or MPHFs and the RAM algorithm. In Section 2.2 we present analytical results of the RAM algorithm. In Section 2.3 we present some experimental results. Finally, in Section 2.4 we conclude this chapter.

2.1 The Family of Functions

The RAM algorithm is a three-step randomized algorithm of *Las Vegas* type because it needs to generate a random acyclic r-partite hypergraph in its first step. Once the hypergraph is obtained, the two other steps are deterministic. To make the exposition as clear as possible we first present our approach for $r = 2$ and, then, generalize it for $r > 2$. Later on, we show that the two interesting cases from the family $\mathcal{F}$ of PHFs or MPHFs are based on 2-graphs and 3-graphs.

The general idea of the algorithm for $r = 2$ is as follows. For a given undirected bipartite 2-graph $G = (V, E)$, $|E| = n$, $|V| = m$ and $m > n$, build an array g such that the following function $\mathit{phf} : E \to [0, m-1]$ is a perfect hash function on E:

$$\mathit{phf}(e = \{u, v\} \in E) = \begin{cases} u, & \text{if } \ (g[u] + g[v]) \bmod 2 = 0 \\ v, & \text{if } \ (g[u] + g[v]) \bmod 2 = 1 \end{cases} \tag{2.1}$$

The problem to solve is to look for an assignment of values from $\{0, 1, r\}$ to vertices so that for each edge the sum of values associated with its endpoints taken modulo r ($r = 2$ in this case) indicates a unique value in the range $[0, m-1]$. This assignment is represented by a function $g : V \to \{0, 1, \ldots, r\}$, which is implemented as the array g in Eq. (2.1). This assignment of values to vertices can be always solved if the graph (or hypergraph) is acyclic [55]. The special value $r = 2$ is used to represent non-assigned vertices. So, we define:

Definition 16 A vertex $v \in V$ is *assigned* if $g[v] \neq r$ and *non-assigned* otherwise.

We now show how each key $x \in S$ is mapped to each edge $e \in E$. Each key $x \in S$ is assigned to edge $e = \{u, v\}$ as follows:

$$\begin{cases} u = h_0(x) \\ v = h_1(x) \end{cases}$$

where we assume $h_0 : U \to [0, m/2 - 1]$ and $h_1 : U \to [m/2, m - 1]$ as two uniform hash functions. The uniform hash assumption is discussed in Section 2.2.5. Each different pair of functions (h_0, h_1) induces a different bipartite random graph $G = G(h_0, h_1)$ and we iteratively select (h_0, h_1) until the induced graph G to be acyclic. In Section 2.2.1 we show how to obtain an acyclic bipartite random graph in an expected constant number of iterations.

To obtain an MPHF we observed that the resulting PHF presented in Eq. (2.1) associates n vertices from V to n edges of S and, by construction, all associated vertices are assigned according to Definition 16. This led us to a well-studied primitive in the succinct data structure area (see e.g. [62, 59, 69]), defined as:

Definition 17 Let $\mathit{rank} : V \to [0, n-1]$ be a function defined as:

$$\mathit{rank}(v) = |\{y \in V \mid y < v \wedge g[y] \neq r\}|. \tag{2.2}$$

Function $\mathit{rank}(v)$ counts how many vertices are assigned before a given vertex $v \in V$, which is uniquely associated with a key $x \in S$.

Therefore, our problem is reduced to computing the array g such that a function $\mathit{mphf} : E \to [0, n-1]$ is a bijection on E, i.e., an MPHF on E and, consequently, an MPHF on S since there is a one-to-one mapping between S and E by using $r = 2$ uniform hash functions:

$$\mathit{mphf}(e = \{u, v\} \in E) = \mathit{rank}(\mathit{phf}(e)) \tag{2.3}$$

The main insights that allow us to build functions that are evaluated in constant time and stored in $O(n)$ bits instead of $O(n \log n)$ bits are twofold. First, the values in the range of g are small enough to be encoded by a constant number of bits, actually $\beta = \lceil \log(r+1) \rceil$ bits. Second, It is possible to build a data structure that allows the computation of function *rank* presented in Eq. (2.2) in constant time by using $o(m)$ additional bits of space.

Figure 2.1 gives an overview of the three-step RAM algorithm for $r = 2$, on a key set $S \subseteq U$ containing the first 4 month names abbreviated to the first three letters, i.e., $S = \{\text{jan}, \text{feb}, \text{mar}, \text{apr}\}$. The mapping step in Figure 2.1(a) builds an acyclic random bipartite graph for $n = 4$ keys or, equivalently, $|E| = n = 4$, and $|V| = m = 8$. The assigning step in Figure 2.1(b) builds the array g so that each edge is uniquely associated with one of its $r = 2$ vertices. For instance, jan is mapped to 2 because $(g[2] + g[5]) \bmod 2 = 0$, feb to 6 because $(g[2]+g[6]) \bmod 2 = 1$, and so on. The ranking step builds the data structure used to compute function $\mathit{rank} : V \to [0, n-1]$ (see Definition 17) in $O(1)$ time. To illustrate, $\mathit{rank}(7) = 3$ means that there are three vertices assigned before vertex 7, which are the vertices $0, 2$ and 6. We are now ready to formally define our family $\mathcal{F}$ of PHFs or MPHFs.

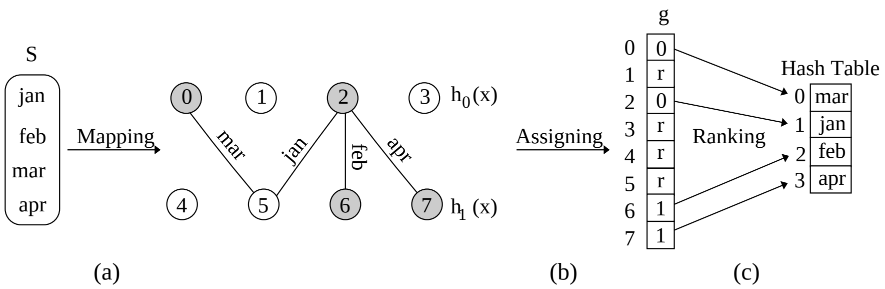


Figure 2.1: (a) The mapping step generates an acyclic bipartite random 2-graph. (b) The assigning step builds an array g so that each edge is uniquely assigned to a vertex. (c) The ranking step builds the data structure used to compute function $\mathit{rank} : V \to [0, n-1]$ in $O(1)$ time.

Definition 18 Let $\mathcal{H}$ be a family of uniform hash functions as presented in Definition 9. Let $h_i : U \to [i\frac{m}{r}, (i+1)\frac{m}{r} - 1]$, $0 \leq i < r$, be r uniform hash functions from $\mathcal{H}$. The r functions and the set S define, in a natural way, a random r-uniform r-partite hypergraph. Let $G_r = G_r(h_0, h_1 \ldots, h_{r-1})$ be such a hypergraph with vertex set $V = [0, m-1]$ and edge set $E = \{\{h_0(x), h_1(x), \ldots, h_{r-1}(x)\} \mid x \in S\}$. Let $g : V \to \{0, 1, \ldots, r\}$ be a function, which is implemented as an array g, such that for each edge the sum of values in g associated with its endpoints taken modulo r indicates a unique value in the range $[0, m-1]$. Let $\mathcal{PHF}$ be a family of PHFs from S to $[0, m-1]$ with parameters $r \geq 2$ and a class $\mathcal{H}$ of uniform hash functions, defined as:

$$\mathcal{PHF}(r, \mathcal{H}) = \left\{ \mathit{phf} \mid \mathit{phf}(x) = h_i(x), i = \left(\sum_{i=0}^{r-1} g[h_i(x)] \right) \bmod r, h_i \in \mathcal{H} \right\} \tag{2.4}$$

Let $\mathcal{MPHF}$ be a family of MPHFs from S to $[0, n-1]$ with parameter $\mathcal{PHF}$ and defined as:

$$\mathcal{MPHF}(\mathcal{PHF}) = \{ \mathit{mphf} \mid \mathit{mphf}(x) = \mathit{rank}(\mathit{phf}(x)), \mathit{phf} \in \mathcal{PHF} \} \tag{2.5}$$

Then, we define:

$$\mathcal{F}(\mathcal{PHF}, \mathcal{MPHF}) = \{h \mid h \in \mathcal{PHF} \text{ or } h \in \mathcal{MPHF}\} \tag{2.6}$$

From now on we are going to design and analyze the RAM algorithm to prove the following theorem:

Theorem 3 For a given key set S with n keys, a given $r \in D = \{x \mid x \geq 2\}$, a given class of uniform hash functions $\mathcal{H}$, and an induced random acyclic r-partite hypergraph $G_r = (V, E)$, where $|E| = n$, $|V| = m = c(r)n$ and $c : D \to \Re$, it is possible to find in expected linear time an array g that implements a function $g : V \to \{0, 1, \ldots, r\}$ and a data structure rankTable so that a function $h \in \mathcal{F}$ can be computed in $O(1)$ time and described in βm bits if h is a PHF and in $(\beta + \epsilon)m + o(m)$ bits if h is an MPHF, where $\beta = \lceil \log(r + 1) \rceil$ and $0 < \epsilon < 1$. For that $O(n)$ computer words are required.

Figure 2.2 presents a pseudo code for the RAM algorithm. If we strip off the third step we will build PHFs instead of MPHFs. The algorithm receives as input a key set S, $|S| = n$, an edge size r and a family $\mathcal{H}$ of uniform hash functions, and produces in expected $O(n)$ time the resulting functions represented by the array g and a data structure, referred to as rankTable, used to allow the computation of Eq. (2.2) in $O(1)$ time. We now describe and analyze each step in detail.

```
procedure RAM ($S$, $r$, $\mathcal{H}$, $g$, rankTable)
    Mapping ($S$, $\mathcal{H}$, $G_r$, $\mathcal{L}$);
    Assigning ($G_r$, $\mathcal{L}$, $g$);
    Ranking ($g$, rankTable);
```

Figure 2.2: The RAM algorithm.

2.1.1 Mapping Step

The mapping step takes a key set S and a family $\mathcal{H}$ of uniform hash functions as input, and creates a random acyclic r-partite hypergraph G_r and a list of edges $\mathcal{L}$. We used an edge-oriented data structure proposed in [32] to represent the hypergraphs, where each edge is explicitly represented as an array of r vertices and, for each vertex v, there is a list of edges that are incident on v. Figure 2.3 presents a pseudo code for the mapping step.

```
procedure Mapping ($S$, $\mathcal{H}$, $G_r$, $\mathcal{L}$)
repeat
    $E(G_r) = \emptyset$;
    select $h_0, h_1, \ldots, h_{r-1}$ uniformly at random from $\mathcal{H}$;
    for each $x \in S$ do
        $e = \{h_0(x), h_1(x) \ldots, h_{r-1}(x)\}$;
        addEdge ($G_r$, $e$);
    $\mathcal{L}$ = isAcyclic($G_r$);
until $E(G_r)$ is empty
```

Figure 2.3: Mapping step.

The list $\mathcal{L}$ is obtained whenever we test whether G_r is acyclic. For that we iteratively delete edges that are incident on vertices of degree one. The list $\mathcal{L}$ stores the deleted edges in the order of deletions (i.e., the first edge in $\mathcal{L}$ was the first deleted edge, the second edge in $\mathcal{L}$ was the second deleted edge, and so on.) The following algorithm can do this test:

1. Traverse G_r and store in a queue Q every edge that has at least one of its vertices with degree one.

2. Until Q is not empty, dequeue one edge from Q, remove it from G_r, store it in $\mathcal{L}$, and check if any of its vertices is now of degree one. If it is the case, enqueue the only edge that contains that vertex.

Figure 2.4 presents one possible output when applied to the random acyclic bipartite hypergraph G_2 presented in Figure 2.1. The three edges containing vertices of degree one were, first, deleted and stored in $\mathcal{L}$. Then the only edge containing vertices of degree two and three was deleted and stored in $\mathcal{L}$.

0	1	2	3	
{0,5}	{2,6}	{2,7}	{2,5}	$\mathcal{L}$

Figure 2.4: Output of the test to check whether a hypergraph has cycles.

2.1.2 Assigning Step

The assigning step takes the random acyclic r-partite hypergraph G_r and the list of edges $\mathcal{L}$ as input, and produces an assignment of values to the vertices of G_r that is represented by the array g. The assignment is created as follows. Let *Visited* be a boolean vector of size m that indicates whether a vertex has been visited. We first initialize $g[i] = r$ (i.e., each vertex is unassigned) and $\mathit{Visited}[i] = \mathit{false}$, $0 \leq i \leq m-1$. Then, for each edge $e \in \mathcal{L}$ from tail to head, we look for the first vertex u belonging to e not yet visited. Let j, $0 \leq j \leq r-1$ be the index of u in e. Then, we set $g[u] = (j - \sum_{v \in e \wedge \mathit{Visited}[v]=\mathit{true}} g[v]) \bmod r$. Whenever we pass through a vertex u from e, if it has not yet been visited, we set $\mathit{Visited}[u] = \mathit{true}$. Figure 2.5 presents a pseudo code for the assigning step.

```
procedure Assigning ($G_r$, $\mathcal{L}$, $g$)
for $u = 0$ to $m - 1$ do
   Visited[$u$] = false;
   $g[u]$ = $r$;
for i = $|\mathcal{L}| - 1$ to 0 do
   $e$ = $\mathcal{L}$[i];
   sum = 0;
   for $k = r - 1$ to 0 do
      if (not Visited[$e[k]$])
         Visited[$e[k]$] = true;
         $u$ = $e[k]$;
         $j$ = $k$;
      else sum += $g[e[k]]$;
      g[u] = ($j$ – sum) mod $r$;
```

Figure 2.5: Assigning step.

Figure 2.6 presents a step by step example for the list of edges of our example presented in Figure 2.4. The initial state is shown in Figure 2.6(a). In Figure 2.6(b), the vertices 2 and 5 of edge $\mathcal{L}[3]$ are marked as visited and $g[2] = (0 - g[5]) \bmod 2 = 0$. In Figure 2.6(c), the vertex 7 of edge $\mathcal{L}[2]$ is marked as visited and $g[7] = (1 - g[2]) \bmod 2 = 1$. In Figure 2.6(d), the vertex 6 of edge $\mathcal{L}[1]$ is marked as visited and $g[6] = (1 - g[2]) \bmod 2 = 1$. Finally, in Figure 2.6(e), the vertex 0 of edge $\mathcal{L}[0]$ is marked as visited and $g[0] = (0 - g[5]) \bmod 2 = 0$.

The reason to traverse the edges in the reverse order they were deleted is to assure that each edge will contain at least one vertex that is traversed for the first time. For example, if the deleted edges were stored in $\mathcal{L}$ in the following order: $e_1, e_2, \ldots, e_i, e_{i+1}, \ldots, e_n$ and we consider edge e_i, then we know that e_i will have at least one of its vertices of degree one by removing the edges $e_1, e_2, \ldots, e_{i-1}$. Let us refer to that vertex as v. Thus, by removing e_i, v will become of degree 0. Therefore, v is not contained in any of the edges $e_{i+1}, \ldots, e_n$. So, by traversing from e_n to e_1, at least one of the vertices in the edges will be traversed

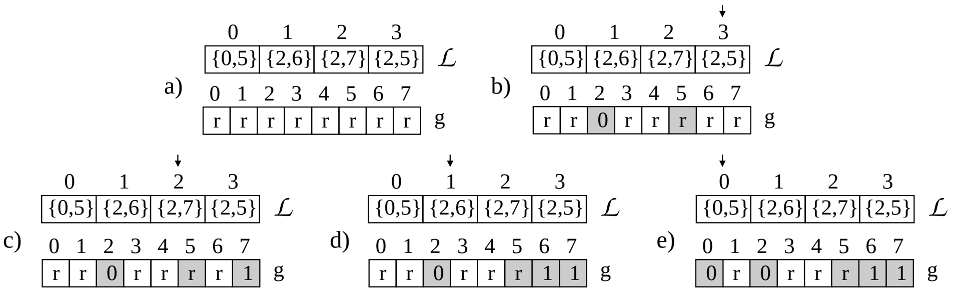

a)
0 1 2 3
{0,5} {2,6} {2,7} {2,5} ℒ
0 1 2 3 4 5 6 7
r r r r r r r r g
b)
{0,5} {2,6} {2,7} {2,5} ℒ
r r 0 r r r r r g
c)
{0,5} {2,6} {2,7} {2,5} ℒ
r r 0 r r r r 1 g
d)
{0,5} {2,6} {2,7} {2,5} ℒ
r r 0 r r r 1 1 g
e)
{0,5} {2,6} {2,7} {2,5} ℒ
0 r 0 r r r 1 1 g

Figure 2.6: Example of the assigning step.

for the first time and such a vertex can be used to uniquely represent the edge.

2.1.3 Ranking Step

The ranking step receives the array g as input and produces the data structure rankTable, which allows the computation of function *rank* presented in Eq. (2.2) in $O(1)$ time. We now present a practical variant described in [62] that uses ϵm additional bits of space, where $0 < \epsilon < 1$. We remark that it is possible to join, in a single and more succinct data structure, the array g and the data structure used to compute function *rank* in constant time (see, e.g., [59, 69]).

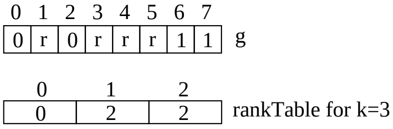
Conceptually, the scheme is very simple: store explicitly the *rank* of every kth index in a rankTable, where $k = \lfloor \log(m)/\epsilon \rfloor$. In the implementation we let the parameter k to be set by the users so that they can trade-off space for evaluation time and vice-versa. In the experiments we set k to 256 in order to spend less space to store the resulting MPHFs. This means that we store in rankTable the number of assigned vertices before every 256th entry in the array g. Figure 2.7 presents a pseudo code for the ranking step.

```
procedure Ranking ($g$, rankTable)
sum = 0;
for $i = 0$ to $|g| - 1$ do
    if($i$ mod $k$ == 0) rankTable[$i/k$] = sum;
    if($g[i] \neq r$) sum++;
```

Figure 2.7: Ranking step.

Figure 2.8 illustrates the ranking step on the array g of Figure 2.6 (e) considering $k = 3$. It means that there is no assigned vertex before $g[0]$, there are two assigned vertices before

$g[3]$, and two before $g[6]$.



0	1	2	3	4	5	6	7	
0	r	0	r	r	r	1	1	g

0	1	2	
0	2	2	rankTable for k=3

Figure 2.8: Example of the ranking step.

2.1.4 Evaluating the Resulting Functions

To compute $rank(u)$, where u is given by a perfect hash function $phf \in \mathcal{PHF}$ (see Eq. (2.4)), we look up in rankTable the rank of the largest precomputed index $v \leq u$, and count the number of assigned vertices from position v to $u-1$. To do this in time $O(1/\epsilon)$ we use a lookup table T_r that allows us to count the number of assigned vertices in $\flat = \epsilon \log m$ bits in constant time for any $0 < \epsilon < 1$. For simplicity and without loss of generality we let $\flat$ be a multiple of the number of bits β used to encode each entry of g. Then, the lookup table T_r can be generated a priori by the pseudo code presented in Figure 2.9, where LS(i',β) stands for the value of the β least significant bits of i' and >> is the right shift of bits. Note that for each $r \geq 2$ a different lookup table T_r is required.

```
procedure GenLookupTable ($\beta$, $\flat$, $T_r$)
for $i = 0$ to $2^\flat - 1$ do
    sum = 0;
    $i' = i$;
    for $j = 0$ to $\flat/\beta - 1$ do
        if(LS($i', \beta$) $\neq r$) sum++;
        $i' = i' >> \beta$;
    $T_r[i]$ = sum;
```

Figure 2.9: Generation of the lookup table T_r.

In the experiments, we have used a lookup table that allows us to count the number of assigned vertices in 8 bits in constant time. Therefore, to compute the number of assigned vertices in 256 bits we need 32 lookups. Such a lookup table fits entirely in the CPU cache because it takes 2^8 bytes of space.

We use the implementation just described because the smallest hypergraphs are obtained when $r = 3$ (see Section 2.2.1). Therefore, the most compact and efficient functions are generated when $r = 2$ and $r = 3$. That is why we have chosen these two instances of the family to be discussed in Sections 2.2.3 and 2.2.4.

Figure 2.10 presents the pseudo code for the resulting PHFs. Note that the resulting functions can be computed in $O(r)$ time. As the practical instances are for $r = 2$ and $r = 3$, then the computational cost is $O(1)$ and it is quite simple to be computed, an important characteristic at retrieval time.

```
function phf ($x$, $g$, $r$)
    $e = \{h_0(x), h_1(x), \ldots, h_{r-1}(x)\}$;
    sum = 0;
    for $i = 0$ to $r - 1$ do sum += $g[e[i]]$;
    return $e[\text{sum} \bmod r]$;
```

Figure 2.10: Pseudo code for the resulting PHFs.

Figure 2.11 presents the pseudo code for the resulting MPHFs. The variable T_r counts the number of assigned vertices in $\mathcal{E}$ entries of g or in $\flat = \beta\mathcal{E} = \epsilon \log m$ bits. We use the notation $g[i \rightarrow j]$ to represent the values stored in the entries from $g[i]$ to $g[j]$ for $i \leq j$. If $j \geq |g|$ or $(j - i + 1) < \mathcal{E}$, then the value r, which is used to represent unassigned vertices, is appended to fulfill the entries to be looked up in T_r. It is easy to see that the computational cost is $O(1/\epsilon)$.

```
function mphf ($x$, $g$, $r$, rankTable, $k$)
    $u$ = phf($x$, $g$, $r$);
    $j = u/k$;
    rank = rankTable[$j$];
    for $i = j * k$ to $u - 1$ step $\mathcal{E}$ do
        rank += $T_r[g[i \rightarrow i + \mathcal{E}]]$;
    return rank;
```

Figure 2.11: Pseudo code for the resulting MPHFs.

2.2 Analytical Results

2.2.1 The Linear Time Complexity

In this section we show that the RAM algorithm runs in expected $O(n)$ time. For that we need to show that the mapping, assigning and ranking steps run in expected $O(n)$ time.

Analysis of the Mapping Step

We start by showing how to obtain a random acyclic r-partite hypergraph $G_r = G_r(h_0, h_1, \dots, h_{r-1})$ with n edges and $m = c(r)n$ vertices with high probability, where $r \in D = \{x \mid x \geq 2\}$ and $c(r)$ is a function with real values on D. We will firstly analyze the case for $r = 2$ and, in the following, the case for $r \geq 3$.

Theorem 4 Let $G_2 = (V, E)$ be a bipartite random graph with n edges and m vertices. Then, if $m = cn$ holds for $c > 2$, the probability that G_2 is a forest (acyclic), for $n \to \infty$, is:

$$Pr_a = \sqrt{1 - \left(\frac{2}{c}\right)^2} \tag{2.7}$$

Proof. Let $G_2 = (V, E)$ be a bipartite random graph with $|V| = 2\rho = m$, and $|E| = dm/2 = n$, where $d = 2n/m$ is the average degree of G_2. To build G_2, each edge is independently taken at random with probability p from all ρ^2 possible edges. As there are $m = 2\rho$ vertices, and each edge is connected to an average of d edges, then we can conclude that $p = d/\rho = 2d/m$. Let C_{2t} be the set of cycles of length $2t$ in the complete bipartite graph K_m, for $t \geq 1$ and each m. A cycle in C_{2t} can be represented as a sequence of $2t$ distinct vertices in K_m by choosing a starting point. Therefore, the cardinality of C_{2t} is given by:

$$|\mathsf{C}_{2t}| = \frac{1}{2t}((\rho)_t)^2, \tag{2.8}$$

where $\rho = \frac{m}{2}$ and $(\rho)_t = \rho(\rho-1)\dots(\rho-t+1)$. As each edge in G_2 is selected independently of the others and with probability $p = \frac{2d}{m}$, then, each cycle in C_{2t} occurs with probability:

$$Pr_{2t}(d) = p^{2t} \tag{2.9}$$

Let $\mathcal{C}_{2t}(G_2)$ be a random variable that measures the number of cycles of length $2t$ in G_2. Let $\mathcal{C}_e(G_2)$ be a random variable that measures the number of cycles of any even length in G_2. The probability distribution of $\mathcal{C}_{2t}(G_2)$ and $\mathcal{C}_e(G_2)$ converge to a Poisson distribution with parameters λ_{2t} and λ_e, respectively. For a more detailed proof of a similar statement see [48, Page 16]. To conclude the proof we are going to show how to get λ_{2t} and λ_e, which represents the average number of cycles of length $2t$ in G_2 and the average number of cycles of even length in G_2, respectively. It is easy to see that, for $m \to \infty$:

$$\lambda_{2t} = Pr_{2t}(d) \times |\mathsf{C}_{2t}| = \left(\frac{2d}{m}\right)^{2t} \frac{1}{2t}((\rho)_t)^2 = \frac{1}{2t}d^{2t} \tag{2.10}$$

and

$$\lambda_e = \sum_{t=1}^{\infty} \lambda_{2t} = \frac{1}{2}d^2 + \frac{1}{4}d^4 + \sum_{t=3}^{\infty} \frac{1}{2t}d^{2t} = -\frac{1}{2}\ln(1-d^2). \tag{2.11}$$

As in [48], we use $\sum_{t=3}^{\infty} \frac{1}{2t}x^t = -\frac{1}{2}\ln(1-x) - \frac{1}{2}x - \frac{1}{4}x^2$, where $x = d^2$. Therefore, the probability that G_2 is a forest is given by:

$$Pr_a(\mathcal{C}_e(G_2) = 0) = e^{-\lambda_e} = \sqrt{1-d^2}. \tag{2.12}$$

Note that d is restricted to be in the range $(0,1)$. As G_2 has $m = cn$ vertices and $n = dm/2$ edges, then $d = 2/c$ and we obtain:

$$Pr_a = \sqrt{1 - \left(\frac{2}{c}\right)^2} \tag{2.13}$$

for $c > 2$. □

For example, when $c = 2.09$ we have $Pr_a = 0.29$. This is very close to 0.294 that is the value we got experimentally by generating 1,000 random bipartite 2-graphs with $n = 10^7$ keys (edges). A rigorous bound on Pr_a for $r > 2$ seems to be technically difficult to obtain. However, the heuristic argument presented in [26, Theorem 6.5], which was rigorously proved in [19], also holds for our random r-partite hypergraphs. Thereby we have the following theorem.

Theorem 5 The threshold for the appearance of a 2-core (a subgraph of minimum degree 2) in a random r-partite hypergraph for $r > 2$ is r/τ, where

$$\tau = \min_{x>0} \left\{ \frac{x}{(1-e^{-x})^{r-1}} \right\} \tag{2.14}$$

From Theorems 4 and 5 we can conclude that with Pr_a bounded by a constant ($Pr_a = \Omega(1)$) and $c(r)$ given by

$$c(r) = \begin{cases} 2+\varepsilon, \varepsilon > 0 & \text{for } r = 2 \\ r\left(\min_{x>0}\left\{\frac{x}{(1-e^{-x})^{r-1}}\right\}\right)^{-1} & \text{for } r > 2, \end{cases} \tag{2.15}$$

the random acyclic r-partite hypergraphs dominate the space of random r-partite hypergraphs. The value $c(3) \approx 1.23$ is a minimum value for Eq. (2.15), as shown in Figure 2.12, previously reported in [55]. This implies that the random acyclic r-partite hypergraphs with the smallest number of vertices happen when $r = 3$. In this case, we have got experimentally $Pr_a \approx 1$ by generating $1,000$ random 3-partite hypergraphs with $n = 10^7$ keys (hyperedges).

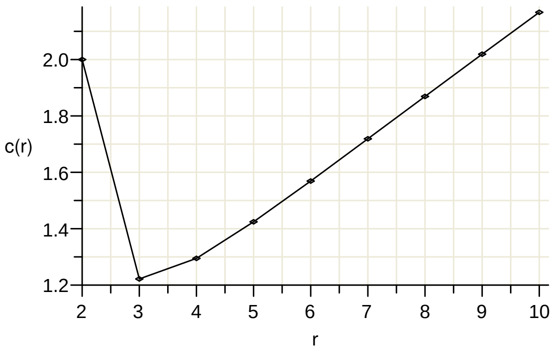


Figure 2.12: Values of $c(r)$ for $r \in \{2, 3, \ldots, 10\}$.

It is interesting to remark that the problems of generating random acyclic r-partite hypergraphs for $r = 2$ and for $r > 2$ have different natures. For $r = 2$, the probability Pr_a varies continuously with the constant c. But for $r > 2$, there is a phase transition. That is, there is a value $c(r)$ such that if $c \leq c(r)$ then Pr_a tends to 0 when n tends to ∞ and if $c > c(r)$ then Pr_a tends to 1. This phenomenon has also been reported in [55] for random r-uniform hypergraphs.

We now show that the expected number of iterations of the mapping step is bounded by a constant. When a random r-partite hypergraph with cycles occurs we abort and select randomly a new tuple of hash functions $(h_0, h_1, \ldots, h_{r-1})$ from $\mathcal{H}$. Then, we can model the number of iterations to generate a random acyclic r-partite hypergraph G_r as a random variable Z that follows a geometric distribution, since the probability Pr_a of generating a random acyclic r-partite hypergraph is $\Omega(1)$. Thus, $Pr(Z = i) = Pr_a(1 - Pr_a)^{i-1}$ and the mean of Z is $1/Pr_a$, which corresponds to the expected number of iterations to obtain G_r. Therefore, as Pr_a is $\Omega(1)$, the expected number of iterations is $O(1)$.

To conclude the analysis of the mapping step presented in Figure 2.3 we need to show that each iteration runs in $O(n)$ time. Statements 4 and 7 are the critical ones in the mapping step, once statements 2 and 3 have costs equal to $O(1)$ and $O(r)$.

It is easy to see that statement 5 in Figure 2.3 has cost $O(r)$. Statement 6 also has cost $O(r)$ because it needs to insert a given edge e in r lists of incident edges, one for each vertex in e. Thereby, statement 4 has cost $O(n)$ for $r = O(1)$. Therefore, it is safe to conclude that the mapping step takes expected $O(n)$ time because it is known (see e.g. [55, Theorem 2.2]) that the algorithm to test whether a hypergraph contains cycles performed in statement 7 also runs in $O(n)$ time.

Analysis of the Assigning Step

It is easy to see in the assigning step presented in Figure 2.5 that the loops of statement 1 and 4 have costs equal to $O(m)$ and $O(rn)$, respectively. This comes from the fact that the operations involved in all other statements have cost $O(1)$. As the number of vertices in G_r is a linear function of the number of edges, i.e., $m = c(r)n$, then, for $r = O(1)$, the assigning step runs in $O(n)$ time.

Analysis of the Ranking Step

It is also easy to see in the ranking step presented in Figure 2.7 that the ranking step runs in $O(n)$ time. This is because statement 2 just loops over the $m = c(r)n$ entries of the array g, performs operations in $O(1)$ time, and $c(r)$ is a constant fixed a priori.

In conclusion, the RAM algorithm takes expected $O(n)$ time because the mapping, assigning and ranking steps run in expected $O(n)$ time.

2.2.2 Space Requirements to Describe the Functions

In this section we present the space required to store the resulting functions disregarding the space for storing the r uniform hash functions, which is discussed in Section 2.2.5.

The description of the resulting functions is compounded by the array g, the rankTable and the lookup table T_r. The resulting array g contains values in the range $[0, r]$ and its domain size is equal to the number of vertices in G_r, i.e., $m = c(r)n$. Then, we can use $\beta = \lceil \log(r+1) \rceil$ bits to encode each value in g. Therefore, g requires βm bits of storage space. The rankTable is stored in ϵm bits because it has m/k entries of size $\log m$ bits and $k = \lfloor \log(m)/\epsilon \rfloor$ for $0 < \epsilon < 1$. The lookup table T_r is stored in $o(m)$ bits because it has m^ϵ entries of size $\log \log m$ bits. Putting all together we have that the number of bits required to store the resulting PHFs and MPHFs are βm and $(\beta + \epsilon)m + o(m)$ bits, respectively.

2.2.3 The 2-graph Instance

The use of acyclic bipartite 2-graphs allows us to generate the PHFs of Eq. (2.4) that give values in the range $[0, m-1]$, where $m = (2+\varepsilon)n$ for $\varepsilon > 0$ (see Section 2.2.1). The significant values in the range of the array g for a PHF are $\{0, 1\}$, because we do not need to represent information to calculate the function *rank* (i.e., $r = 2$). Then, we can use just one bit to represent the value assigned to each vertex, i.e., $\beta = 1$. Therefore, the resulting PHF requires m bits to be stored. For $\varepsilon = 0.09$, the resulting PHFs are stored in approximately $2.09n$ bits and map to the range $[0, 2.09n - 1]$.

To generate the MPHFs of Eq. (2.5) we need to include the ranking information. Thus, we must use the value $r = 2$ to represent unassigned vertices and now two bits are required to encode each value assigned to the vertices, i.e., $\beta = 2$. Then, the resulting MPHFs require $(2+\epsilon)m+o(m)$ bits to be stored (remember that the ranking information requires ϵm bits and the lookup table T_2 requires $o(m)$ bits), which corresponds to $(2+\epsilon)(2+\varepsilon)n+o(n)$ bits for any $\epsilon > 0$ and $\varepsilon > 0$. In the experiments, for $\epsilon = 0.125$ and $\varepsilon = 0.09$ the resulting functions are stored in approximately $4.44n$ bits. We now present two packing schemes that give more compact MPHFs and can be done in $O(n)$ time.

Packing the Resulting MPHFs for $r = 2$ with Arithmetic Coding

The range of significant values assigned to the vertices is clearly [0,2]. Hence, we need log(3) bits to encode the value assigned to each vertex. Theoretically we use arithmetic coding as block of values. Therefore, we can compress the resulting MPHF to use $(\log(3) + \epsilon)(2+\varepsilon)n + o(n)$ bits of storage space by using a simple packing technique. In practice, we can pack the values assigned to every group of 5 vertices into one byte because each assigned value comes from a range of size 3 and $3^5 = 243 < 256$. At generation time we should use a small lookup table of size 5 containing: *pow3_table*$[5] = \{1, 3, 9, 27, 81\}$. To assign a value $x \in [0, 2]$ to a vertex $u \in V$ we use:

```
byte   = g[u/5];
byte  += x * pow3_table[u mod 5];
g[u/5] = byte;
```

At retrieval time we should use a lookup table T_{lookup} of size 5*256=1280 bytes to speed up the recovery of the value x assigned to a given vertex u, as shown below.

```
byte = g[u/5];
```

```
$x = T_{lookup}[u \bmod 5][byte]$;
```

Each entry of the lookup table T_{lookup} is computed by:

$$T_{lookup}[i][j] = (j/pow3\_table[i]) \bmod 3, \tag{2.16}$$

where $0 \leq i < 5$ and $0 \leq j < 256$. In the experiments, for $\epsilon = 0.125$ and $\varepsilon = 0.09$, the resulting functions are stored in approximately $3.6n$ bits.

A More Effective Packing Scheme for $r = 2$

We now present a more effective packing scheme that allows us to compress the resulting MPHFs to use $(3+\epsilon)n$ bits, for $\epsilon > 0$. The basic idea is to put the information to compute the array g and the information to compute the function *rank* in different data structures. Therefore, the range of the values in the array g is narrowed to $[0, 1]$ instead of $[0, 2]$. Then, it is now possible to spend just $\beta = 1$ bit for each one of the m values of g. This implies that the array g is used to represent a $phf \in \mathcal{PHF}$.

Let $V_a = \{phf(x) \mid x \in S \ \wedge \ phf \in \mathcal{PHF}\}$ be the set of assigned vertices in V. To compute function *rank* somehow we need to represent V_a. Let $\mathcal{R}$ be a bit vector of size $|V| = m$ used to represent V_a. That is, $\mathcal{R}[v] = 1$ if $v \in V_a$ and $\mathcal{R}[v] = 0$ otherwise. Thereby we can redefine the function *rank* as follows.

Definition 19 Let $rank : V \rightarrow [0, n-1]$ be a function defined as:

$$rank(v) = |\{y \in V \ \mid \ y < v \wedge \mathcal{R}(y) = 1\}|. \tag{2.17}$$

In this case it would be required to store the array g and the vector $\mathcal{R}$, both with m one-bit entries, plus $o(m)$ bits required to compute function *rank* in $O(1)$ time. However, we can create a compressed representation that uses just over 3 bits per key by noticing that there exist exactly n assigned vertices in V, i.e., $|V_a| = n$, and the value of g for all non-assigned vertices $V_{na} = V - V_a$ is equal to 0. Thus, the contents of g and $\mathcal{R}$ are not independent. For instance, there can be a non-zero bit in $g[v]$ only if $\mathcal{R}[v] = 1$. Therefore, it is possible to create a compressed representation g' that uses only n bits and enables us to compute any bit of g in constant time. First of all, if $\mathcal{R}[v] = 0$ we can conclude that $g[v] = 0$. We want to initialize g' such that $g[v] = g'[rank(v)]$ whenever $\mathcal{R}[v] = 1$, i.e., $v \in V_a$. This is possible since $rank(v)$ is 1-1 on elements in V_a. In conclusion, we can replace g by g' and reduce the space usage to $n + m + o(m)$ bits. By using $m = (2 + \epsilon/2)n$

for $\epsilon > 0$ and $(\epsilon/2)n$ bits of extra space to support rank operations efficiently, the total space is $(3 + \epsilon)n$ bits.

2.2.4 The 3-graph Instance

The use of 3-graphs allows us to generate more compact PHFs and MPHFs at the expense of one more hash function h_2. An acyclic 3-partite random 3-graph is generated with probability $\Omega(1)$ for $m \geq c(3)n$, where $c(3) \approx 1.23$ is the minimum value for $c(r)$ (see Section 2.2.1). Therefore, we will be able to generate the PHFs of Eq. (2.4) so that they will produce values in the range $[0, (1.23 + \varepsilon)n - 1]$ for any $\varepsilon \geq 0$. The values assigned to the vertices are drawn from $\{0, 1, 2, 3\}$ and, consequently, each value requires $\beta = 2$ bits to be represented. Thus, based on the fact that for PHFs no ranking information is needed (i.e., $\epsilon = 0$), the resulting PHFs require $2(1.23 + \varepsilon)n$ bits to be stored, which corresponds to $2.46n$ bits for $\varepsilon = 0$.

We can generate the MPHFs of Eq. (2.5) from the PHFs that take into account the special value $r = 3$. The resulting MPHFs require $(2 + \epsilon)(1.23 + \varepsilon)n + o(n)$ bits to be stored for any $\epsilon > 0$ and $\varepsilon \geq 0$, once the ranking information must be included. In the experiments, for $\epsilon = 0.125$ and $\varepsilon = 0$, we have got MPHFs that are stored in approximately $2.62n$ bits.

Packing the Resulting PHFs for $r = 3$ with Arithmetic Coding

For PHFs that map to the range $[0, (1.23+\varepsilon)n-1]$ we can get still more compact functions. This comes from the fact that the only significant values assigned to the vertices that are used to compute Eq. (2.4) are $\{0, 1, 2\}$. Then, we can apply the arithmetic coding technique aforementioned to get PHFs that require $\log(3)(1.23 + \varepsilon)n$ bits to be stored, which is approximately $1.95n$ bits for $\varepsilon = 0$. For this we must replace the special value $r = 3$ to 0.

2.2.5 The Use of Universal Hashing

The uniform hashing assumption is not feasible because each hash function $h_i : U \to [i\frac{m}{r}, (i + 1)\frac{m}{r} - 1]$ for $0 \leq i < r$ would require at least $n \log \frac{m}{r}$ bits to be stored plus the space for the PHFs. As mentioned in Chapter 1 (Section 1.4) limited randomness represented by universal hash functions is often as good as total randomness when the key universe U is much larger than the functions range.

For our experiments we choose h_i from a family of heuristic hash functions proposed in [50] with very good performance in practice but with no theoretical foundation. These functions do not impose any upper bound for the key sizes and their description requires just the storage of an integer that is used as a seed for a pseudo random number generator. The function just loops over the key doing bitwise operations on blocks of 12 bytes and, at the end, a 12 byte long integer is generated.

From a theoretical perspective, the uniform hashing assumption is not too harmful, as we can use the *split-and-share* approach [30] to simulate a uniform hash function by using $o(n)$ bits of extra space. We use this in the design of the EM algorithm presented in Chapter 4. In Chapter 3 we show how to use this idea to create uniform hash functions for the small buckets generated in the EM algorithm.

2.2.6 The Space Requirements to Generate the Functions

In this section we show that the RAM algorithm presented in Figure 2.2 needs $O(n)$ computer words of main memory to generate functions of $\mathcal{F}$. For that we will assume that the key set S is kept in external memory and just the data structures involved in the generation process are kept in internal memory. We need to maintain the following data structures in internal memory.

1. The r uniform hash functions $h_0, h_1, \ldots, h_{r-1}$. Each function can be described in $o(n)$ bits by using the *split-and-share* technique.

2. The random acyclic r-partite hypergraph G_r. As $m = c(r)n$, it is possible to store G_r in $O(rn)$ computer words by using the data structure proposed in [32].

3. The list $\mathcal{L}$ of deleted edges obtained when we test whether G_r is a forest. It is also stored in $O(rn)$ computer words.

4. The description of a resulting function $h \in \mathcal{F}$. This corresponds to βm bits if $h \in \mathcal{PHF}$ and $(\beta + \epsilon)m + o(m)$ bits if $h \in \mathcal{MPHF}$.

Therefore, for $r = O(1)$, we need $O(n)$ computer words to generate the functions of $\mathcal{F}$. This ends the proof of Theorem 3.

2.3 Experimental Results

The purpose of this section is to evaluate the performance of the RAM algorithm and to compare it with the main practical perfect hashing algorithms we found in the literature. In Section 2.3.1 we consider key sets that can be handled in internal memory by the RAM algorithm. The experimental results for the RAM algorithm match the analytical results presented in Section 2.2. In Section 2.3.2 we compare the RAM algorithm with the main ones found in the literature.

The algorithms were implemented in the C language and are available under the GNU Lesser General Public License (LGPL) at `http://cmph.sf.net`. The experiments were carried out on a computer running the Linux operating system, version 2.6, with a 1.86 gigahertz Intel Core 2 processor with a 4 megabyte L2 cache and 1 gigabyte of main memory. For the experiments we used two collections: (i) a set of 150 million randomly generated 4 byte long IP addresses, and (ii) a set of $1,024$ million 64 byte long (on average) URLs collected from the Web.

To compare the algorithms we used the following metrics: (i) The amount of time to generate PHFs or MPHFs, referred to as Generation Time. (ii) The space requirement for the description of the resulting PHFs or MPHFs to be used at retrieval time, referred to as Storage Space. (iii) The amount of time required by a PHF or an MPHF for each retrieval, referred to as Evaluation Time.

2.3.1 Performance of the RAM Algorithm

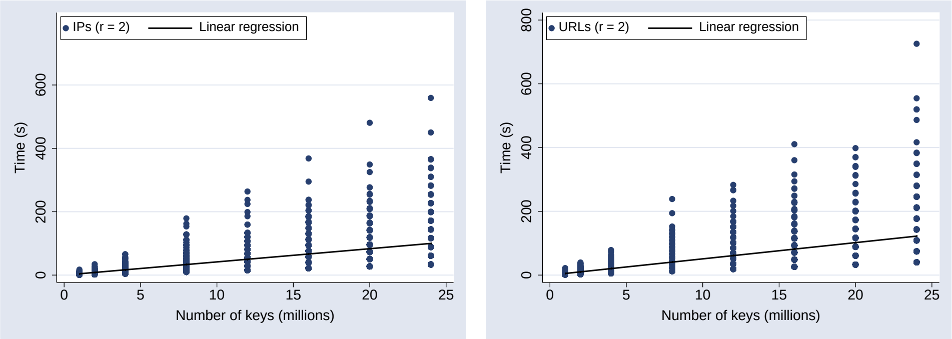
In this section we evaluate the performance of the RAM algorithm considering generation time and storage space as metrics. We will consider two versions of the RAM algorithm: (i) the version that works on random graphs with no cycles when $r = 2$ and, (ii) the version that works on random hypergraphs with no cycles when $r = 3$.

Let us start with the version of the RAM algorithm that works on random graphs (i.e., $r = 2$) with no cycles. We can consider the runtime of the algorithm to have the form $\alpha n Z$ for an input of n keys, where α is some machine dependent constant that further depends on the length of the keys and Z is a random variable with geometric distribution with mean $1/Pr_a$, where $Pr_a = \sqrt{1-(2/c)^2}$ (see Theorem 4). All results in our experiments for this version were obtained taking $c = 2.09$; the larger is c the faster is the algorithm because Pr_a increases continuously with c.

The values chosen for n were $1, 2, 4, 8, 12, 16, 20$ and 24 million keys. Although we have 150 million random IPs and $1,024$ million URLs, on a PC with 1 gigabyte of main memory, the RAM algorithm is able to handle an input with at most 24 million keys. This is mainly because the sparse random hypergraph required to generate the functions is memory demanding.

In order to estimate the number of trials for each value of n we used a statistical method for determining a suitable sample size (see, e.g., [47, Chapter 13]). As we obtained different values for each n, we used the maximal value obtained, namely, 300 trials in order to have a confidence level of 95%.

Figure 2.13 presents the runtime for each trial. In addition, the solid line corresponds to a linear regression model obtained from the experimental measurements. As we can see, the runtime for a given n has a considerable fluctuation, which gives a coefficient of determination $R^2 = 66\%$. However, the fluctuation also grows linearly with n, as explained in the following.



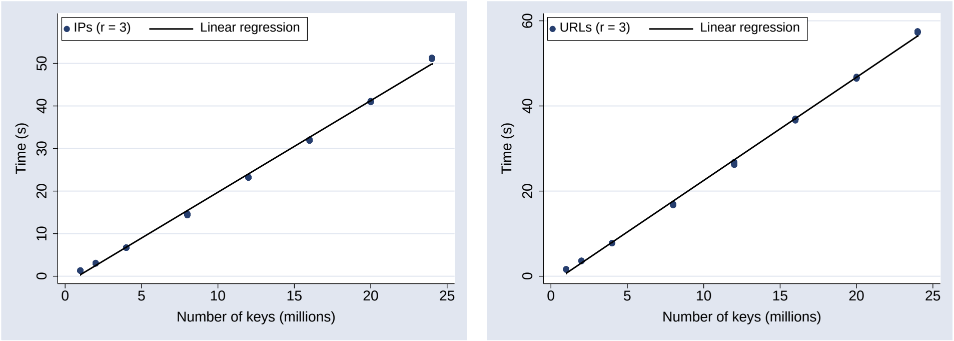
(a) IPs collection

(b) URLs collection

Figure 2.13: Number of keys in S versus generation time for the RAM algorithm that works on acyclic random graphs with $r = 2$. The solid line corresponds to a linear regression model for the generation time ($R^2 = 66\%$).

The observed fluctuation in the runtimes is as expected; recall that this runtime has the form $\alpha n Z$ with Z a geometric random variable with mean $1/Pr_a = 1/0.29$ for $c = 2.09$. Thus, the runtime has mean $\alpha n/Pr_a = 3.45\alpha n$ and standard deviation $\alpha n\sqrt{(1 - Pr_a)/(Pr_a)^2} = 2.91\alpha n$. Therefore, the standard deviation also grows linearly with n, as experimentally verified in Figure 2.13.

The version of the RAM algorithm that works on hypergraphs with no cycles, where $r = 3$, is the fastest version. This is a consequence of Theorem 5, because the probability of obtaining a hypergraph with no cycles for $r > 2$ tends to 1 for $c > c(r)$, where $c(r)$ is given in Eq. (2.15). For $r = 3$, $c(3) \in (1.22, 1.23)$ and therefore we use c $= 1.23$ in our experiments. We again use the statistical method for determining a suitable sample size to estimate the number of trials to be run for each value of n. We got that just one trial for each n would be enough with a confidence level of 95%. However, we made 25 trials. This number of trials seems rather small, but, as shown in Figure 2.14, the behavior of this version of the RAM algorithm is very stable and its runtime is almost deterministic (i.e., the standard deviation is very small), which gives a coefficient of determination $R^2 = 99\%$ for the linear regression model obtained.



(a) IPs collection (b) URLs collection

Figure 2.14: Number of keys in S versus generation time for the RAM algorithm that works on acyclic random hypergraphs with $r = 3$. The solid line corresponds to a linear regression model for the generation time ($R^2 = 99\%$).

To conclude this section we now compare the two versions of the RAM algorithm by taking $n = 1, 12$ and 24 million keys in the two collections and by considering generation time and storage space as metrics. Table 2.1 presents the respective confidence intervals for the generation time, which is given by the average time ± the distance from average time considering a confidence level of 95%, and the respective values for the storage space. It is possible to see that the generation time, as expected, is influenced by the key length (IPs are 4 bytes long and URLs are 64 bytes long on average), and the storage space is not. It is also possible to see that the fastest algorithm, for $r = 3$, also generates the most compact functions.

2.3.2 Comparison with the Main Practical Results in the Literature

The main practical perfect hashing algorithms we found in the literature to compare the RAM algorithm with are: Botelho, Kohayakawa and Ziviani [12] (referred to as BKZ), Fox, Chen and Heath [38] (referred to as FCH), Majewski, Wormald, Havas and Czech [55]

n	RAM algorithm	Generation Time (sec)		Storage Space	
		IPs	URLs	Bits/Key	Size (MB)
1×10^6	$r = 2$	3.09 ± 0.28	4.00 ± 0.34	3.60	0.43
	$r = 3$	1.32 ± 0.01	1.61 ± 0.01	2.62	0.31
12×10^6	$r = 2$	48.30 ± 4.42	59.04 ± 5.47	3.60	5.15
	$r = 3$	23.2 ± 0.02	26.31 ± 0.06	2.62	3.75
24×10^6	$r = 2$	101.59 ± 9.13	125.65 ± 11.35	3.60	10.30
	$r = 3$	51.19 ± 0.03	57.39 ± 0.04	2.62	7.50

Table 2.1: Comparison of the two versions of the RAM algorithm considering generation time and storage space, and using $n = 1, 12,$ and 24 million keys for the two collections.

(referred to as MWHC), and Pagh [61] (referred to as PAGH). For the MWHC algorithm we used the version based on random hypergraphs with $r = 3$. We did not consider the one that uses random graphs because it is shown in [12] that the BKZ algorithm outperforms it. The BKZ algorithm is presented in Chapter 6.

We used the hash function presented in [50] for all the algorithms. For all the experiments we used $n = 3,541,615$ keys for the two collections. The reason to choose a small value for n is because the FCH algorithm has exponential time on n for the generation phase, and the times explode even when a number of keys are a little over.

We first compare the RAM algorithm for constructing MPHFs with the other algorithms, considering generation time and storage space. Table 2.2 shows that the RAM algorithm for $r = 3$, and the MWHC algorithm are faster than the others in generating MPHFs. This is because they work on random acyclic hypergraphs with $r = 3$ and the probability of obtaining such a hypergraph tends to 1 as n tends to infinity. Therefore, they scan the whole key set stored in external memory once with high probability, whereas all the other algorithms scan the whole key set everytime a failure occurs and they have a higher probability of failure.

It is also important to note that the resulting functions of the RAM algorithm are the most compact functions. The storage space requirements in bits per key for the two versions of the RAM algorithm are 3.6 when $r = 2$, and 2.62 when $r = 3$. For the BKZ, MWHC and PAGH algorithms they are $\log n$, $1.23 \log n$ and $2.03 \log n$ bits per key, respectively. Therefore, the RAM algorithm is the best choice for sets that can be handled in main memory.

We now compare the algorithms considering evaluation time. Table 2.3 shows the

Algorithms		Generation Time (sec)		Storage Space	
		IPs	URLs	Bits/Key	Size (MB)
RAM	$r = 2$	11.39 ± 1.33	16.73 ± 1.89	3.60	1.52
	$r = 3$	5.46 ± 0.01	6.74 ± 0.01	2.62	1.11
BKZ		9.22 ± 0.63	11.33 ± 0.70	21.76	9.19
FCH		$2,052.7 \pm 530.96$	$2,400.1 \pm 711.60$	4.22	1.78
MWHC		5.98 ± 0.01	7.18 ± 0.01	26.76	11.30
PAGH		39.18 ± 2.36	42.84 ± 2.42	44.16	18.65

Table 2.2: Comparison of the algorithms for constructing MPHFs considering generation time and storage space, and using $n = 3,541,615$ for the two collections.

evaluation time for a random permutation of the n keys. Although the number of memory probes at retrieval time of the MPHF generated by the PAGH algorithm is optimal [61] (it performs only 1 memory probe), it is important to note in this experiment that the evaluation time is smaller for the FCH and the RAM algorithms because the generated functions fit entirely in the machine's L2 cache (see the storage space size for the RAM algorithm and the FCH algorithm in Table 2.2). Therefore, the more compact an MPHF is, the more efficient it is if its description fits in the cache. For example, for sets of size up to 13 million keys of any type the resulting functions generated by the RAM algorithm with $r = 3$ will entirely fit in a 4 megabyte L2 cache.

Algorithms		RAM		BKZ	FCH	MWHC	PAGH
		$r = 2$	$r = 3$				
Evaluation Time (sec)	IPs	1.19	1.16	1.33	0.75	1.53	1.30
	URLs	2.12	2.11	2.24	1.61	2.46	2.20

Table 2.3: Comparison of the algorithms considering evaluation time and using the collections IPs and URLs with $n = 3,541,615$.

In a converse situation, where the functions do not fit in the cache, the MPHFs generated by the PAGH algorithm are the most efficient, as shown in Table 2.4.

Finally, we compare the PHFs and MPHFs generated by the different versions of the RAM algorithm. Table 2.5 shows that the generation times for PHFs and MPHFs are almost the same, with the algorithms for $r = 3$ being the fastest because the probability of obtaining an acyclic 3-graph for $c = 1.23$ tends to one, whereas the probability for a 2-graph where $c = 2.09$ tends to 0.29 (see Section 2.2.1). For PHFs with $m = 1.23n$, the storage requirement drops from 2.62 to 1.95 bits per key when $r = 3$. The PHFs with

Algorithms		RAM		BKZ	FCH	MWHC	PAGH
		$r = 2$	$r = 3$				
Evaluation Time (sec)	IPs	7.11	8.02	4.86	–	6.29	4.60
	URLs	10.17	11.49	9.29	–	9.61	9.25

Table 2.4: Comparison of the algorithms considering evaluation time and using the collections IPs and URLs with $n = 15,000,000$.

$m = 2.09n$, and $m = 1.23n$ are the fastest at evaluation time because no ranking or packing information needs to be computed.

RAM		m	Generation Time (sec)		Eval. Time (sec)		Sto. Space
r	Packed		IPs	URLs	IPs	URLs	Bits/Key
2	no	$2.09n$	10.50 ± 1.24	14.79 ± 1.58	0.68	1.63	2.09
	yes	n	11.39 ± 1.33	16.73 ± 1.89	1.19	2.12	3.60
3	no	$1.23n$	5.51 ± 0.01	6.76 ± 0.01	0.79	1.68	2.46
	yes	$1.23n$	5.54 ± 0.01	6.78 ± 0.02	0.79	1.71	1.95
	no	n	5.46 ± 0.01	6.74 ± 0.01	1.16	2.11	2.62

Table 2.5: Comparison of the PHFs and MPHFs generated by the RAM algorithm, considering generation time, evaluation time and storage space metrics using $n = 3,541,615$ for the two collections. For packed schemes see Sections 2.2.3 and 2.2.4.

2.4 Conclusions

We have presented an efficient algorithm to generate a family of near-space optimal PHFs or MPHFs for key sets that can be handled in the internal memory. The algorithm accesses memory in a random fashion and then was called internal random access algorithm (RAM).

The space necessary to describe the functions takes a constant number of bits per key. The space usage depends on the relation between the size m of the hash table and the size n of the input. For $m = n$, the space usage is in the range $2.62n$ to $4.44n$ bits, depending on the constants involved in the construction and in the evaluation phases. For $m = 1.23n$ the space usage is in the range $1.95n$ to $2.46n$ bits. In all cases, this is within a small constant factor from the information theoretical minimum of approximately $1.44n$ bits for MPHFs and $0.89n$ bits for PHFs, something that has not been achieved by previous algorithms, except asymptotically for very large n (i.e, $n > 2^{300}$).

The resulting functions are evaluated for a given element of a key set in constant time. Moreover, as the generated function is space economical, its evaluation is likely to be performed in the CPU cache, which is very efficient in time. However, the resulting MPHFs still assume uniform hashing. In Chapter 3 we present a particular engineering of the *split-and-share* technique [30] to simulate a uniform hash function on small key buckets. This result is used by the EM algorithm presented in Chapter 4 to generate simple and near space-optimal PHFs and MPHFs without assuming uniform hashing.

Chapter 3

Using Split-and-Share to Simulate Uniform Hash Functions

In this chapter we show how to use the *split-and-share* approach presented in [30] to simulate uniform hash functions on each of the small key buckets generated by the EM algorithm (the EM algorithm is described in detail in Chapter 4). Our implementation has two advantages compared to the implementation described in [30]. First, it generates a *family* of hash functions for each bucket, with only a constant factor of space overhead. This is necessary because the RAM algorithm needs to be able to choose new hash functions when it fails because the random graph induced by the current hash functions contains cycles. Second, the hash function is well suited for string data, and the buckets obtained are provably very small – a fact that we exploit in the implementation.

We describe how to implement a single hash function in the family. To get the r hash functions needed for the RAM algorithm we conceptually just keep r representations ($r = 3$). In the implementation we exploit the fact that the number of random accesses can be kept down by merging the hash function representations, as explained in Section 3.2.2. A previous version of this result was presented in [15].

This chapter is organized as follows. In Section 3.1 we show how to split the key set into small buckets. In Section 3.2 we show how to simulate uniform hash functions on the small buckets. Finally, in Section 3.3 we conclude this chapter.

3.1 Splitting

The first ingredient we need is a hash function that maps the keys of S to $N_b = 2^b$ buckets, such that all buckets are of approximately the same size. If a uniform hash function is used and $N_b < n/\log n$, it is well known that the largest bucket will contain $O(n/N_b)$ keys with high probability. Most explicitly defined hash functions (e.g., universal or polynomial hash functions) have much weaker guarantees. However, in [2] it is shown that if we fix a concrete family of universal hash functions, it is possible to considerably diminish the loss by using universal hash functions. We want to apply the following result presented in [2], where N_b is assumed to be a power of 2:

Theorem 6 [2] Let $\mathcal{H}$ be the family of all linear transformations over Galois field 2, or simply GF(2), the field of two elements, mapping $\{0,1\}^L$ to $\{0,1\}^b$. Let $N_b = 2^b$ and suppose that $N_b \leq n/\log n$. Let $S \subseteq \{0,1\}^L$ be a set of size n, and pick $h_0 \in \mathcal{H}$ uniformly at random. Then the expected size of the largest bucket when hashing S using h_0 is $O(n \log\log(n)/N_b)$.

To apply the above result we need to identify strings with bit vectors in $\{0,1\}^L$. Since we are dealing with zero-terminated strings, this is simple: Just pad with extra zeros at the end to get a string of L bits. As we will see shortly, the time to hash a string will be proportional to its length, not proportional to L.

The theorem says that the expected size of the largest bucket is within a factor $O(\log\log n)$ of the average bucket size. This means that with a function from $\mathcal{H}$ we can split the set into $O(n\log\log(n)/\ell)$ buckets of maximum size ℓ. Thus, for a given constant $\kappa > 0$ we have:

$$\begin{aligned} 2^b &\leq \frac{\kappa n \log\log(n)}{\ell} \\ b &\leq \log n + \log\log\log n - \log \ell + \log \kappa \qquad (3.1) \end{aligned}$$

For the EM algorithm to generate functions with space complexity $O(n)$ bits we have to impose the following restriction on ℓ:

$$\begin{aligned} N_b &\leq \frac{n}{\log n} \\ \ell &\geq \kappa \log n \log\log n \\ \ell &= \Omega(\log n \log\log n) \qquad (3.2) \end{aligned}$$

We will not analyze the constant in the number of buckets – instead, our algorithm simply chooses the smallest number of buckets possible with a given hash function. Technically this means that the space usage of the EM algorithm, as it is implemented, will be $O(n)$ bits only in expectation. However, given an upper bound on the constant of Theorem 6 we may turn this into a worst-case space bound by picking a new function h_0 until the maximum bucket size is close to the expectation. Our implementation is engineered to work with maximum bucket size $\ell = 256$. For extremely large sets (hundreds of billions of keys) a larger maximum bucket size ℓ is needed to keep the space at $O(n)$ bits. This happens because ℓ increases asymptotically with n, as shown in Eq. (3.2).

Let $h_0 : \{0,1\}^L \to \{0,1\}^b$ be a function from the family $\mathcal{H}$ of Theorem 6 with the following form: $h_0 = Ax$, where A is a $b \times L$ matrix with entries in GF(2). To represent h_0 we need to store the bL bits of the matrix A. A matrix-vector product Ax can be implemented by adding the columns corresponding to 1s in x. Note that addition of vectors over GF(2) corresponds to bit-wise exclusive-or. For example, let us consider $L = 3$ bits, $b = 3$ bits, $x = 110$ and

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}.$$

Then

$$h_0(x) = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

The evaluation time for this is $O(L)$, assuming that a column vector can be stored in one machine word. To obtain faster evaluation we use a tabulation idea from [3] that gives evaluation time $O(L/\log\sigma)$ by using space $O(\sigma L/\log\sigma \log n)$ for $\sigma > 0$. Note that if x is short, e.g. 8 bits, we can simply tabulate all the function values and compute $h_0(x)$ by looking up the value in an array. To make the same thing work for longer keys, split the matrix A into parts of 8 columns each: $A = A_1|A_2|\ldots|A_{\lceil L/8 \rceil}$, and create a lookup table $h_{0,i}$ for each submatrix. Similarly split x into parts of 8 bits, $x = x_1 x_2 \ldots x_{\lceil L/8 \rceil}$. Now $h_0(x)$ is the bit-wise exclusive-or of $h_{0,i}[x_i]$, for $i = 1, \ldots, \lceil L/8 \rceil$. Therefore, we have set σ to 256 so that keys of size L can be processed in chunks of $\log\sigma = 8$ bits. Observe that all zero characters in a string can simply be skipped, since their contribution to the matrix-vector product will be zero. This means that the evaluation time is proportional to the number of characters in the input string.

3.2 Simulating Uniform Hash Functions

The second ingredient of split-and-share is a single hash function ρ that, when applied to the keys of a single bucket, behaves like a fully random hash function with high probability. This function can then be shared among all buckets. As stated earlier, we will in fact present a way of generating a family of hash functions such that for any bucket, each function behaves like a fully random function with high probability. Technically, this is done by making ρ a function of two parameters (see Eq. (3.3)), where the second parameter s describes which function in the family is used. To make the analysis go through we need the fact that for any bucket, the functions in the family are pairwise independent.

3.2.1 The Shared Function

Let $y_1, \ldots, y_k$ be independently chosen functions from a pairwise independent family of functions from $\{0,1\}^L$ to $\{0,1\}^\delta$, where $2^\delta \gg \ell$ is a parameter to be chosen later. Also, let p be a prime number, and k a positive integer. We will use a variation of a family due to [31] that achieves full independence with high probability on small sets:

$$\rho(x,s) = \left(\sum_{j=1}^{k} t_j[y_j(x)] + s \times t'_j[y_j(x)] \right) \bmod p. \tag{3.3}$$

The tables $t_1, \ldots, t_k$ and $t'_1, \ldots, t'_k$ contain 2^δ random values from $\{0, \ldots, p-1\}$. We prove the following lemma to obtain the independence property we need:

Lemma 2 For any $s_i, s'_i \in \{1, \ldots, p-1\}$, $s_i \neq s'_i$, $B_i \subseteq S$ of size $|B_i|$, where B_i is the set of keys in bucket i, the following holds: With probability at least $1 - |B_i| \left(|B_i|/2^\delta\right)^k$ over the choice of $y_1, \ldots, y_k$ the function values $\rho(x,s)$, $x \in B_i$, $s \in \{s_i, s'_i\}$ are independent and uniformly distributed in $\{0, \ldots, p-1\}$.

PROOF. Consider arbitrary values $v_{x,s} \in \{0, \ldots, p-1\}$, for $x \in B_i$, $s \in \{s_i, s'_i\}$. Independence means that the probability that $\rho(x,s) = v_{x,s}$ for all $x \in B_i$, $s \in \{s_i, s'_i\}$ is $p^{-2|B_i|}$. To arrive at a sufficient condition for independence, consider how the entries of $t_1, \ldots, t_k$ and $t'_1, \ldots, t'_k$ are accessed when computing $\rho(x,s)$ for $x \in B_i$ and $s \in \{s_i, s'_i\}$. Assume that a key $x \in B_i$ has an associated *unique entry* $y_{j_x}(x)$ in t_{j_x} and t'_{j_x}, that is not read when evaluating ρ on keys in $B_i - \{x\}$. Then for any choice of values in other entries, the values $\rho(x, s_i)$ and $\rho(x, s'_i)$ are independent and uniformly distributed in $\{0, \ldots, p-1\}$. This is because there is exactly one choice of $t_{j_x}[y_{j_x}(x)]$ and $t'_{j_x}[y_{j_x}(x)]$ for each value of v_{x,s_i}, v_{x,s'_i}

(two independent linear equations with two variables in GF(p)). In conclusion, a sufficient condition for independence is that we can assign a unique entry to each $x \in B_i$.

Since $y_1, \ldots, y_k$ are chosen from a pairwise independent family we know that for any $x \in B_i$ the probability that x does *not* have a unique entry is at most $(|B_i|/2^\delta)^k$. By the union bound, the probability that *some* key in B_i does not have a unique entry is at most $|B_i|\,(|B_i|/2^\delta)^k$. □

3.2.2 Using the Shared Function

We want to use the shared function to implement the RAM algorithm on the buckets. In fact, we will use three independent shared functions ρ_0, ρ_1, ρ_2, one for each hash function needed by RAM. However, for reasons explained below all three functions will use the same functions $y_1, \ldots, y_k$.

Definition 20 Let $|B_i|$ denote the number of keys mapped by h_0 to bucket B_i and $m_i = \frac{c|B_i|}{3}$, for $c \geq 1.23$, then

$$\begin{aligned} h_{i0}(x) &= \rho_0(x, s_i) \bmod m_i \\ h_{i1}(x) &= \rho_1(x, s_i) \bmod m_i + m_i \\ h_{i2}(x) &= \rho_2(x, s_i) \bmod m_i + 2m_i \end{aligned}$$

The variable s_i is specific for bucket i. The algorithm randomly selects s_i from $\{1, \ldots, p-1\}$ until the functions h_{i0}, h_{i1}, and h_{i2} work with the RAM algorithm of Section 2.2.4, which is used to generate a PHF or an MPHF for each bucket. We will prove in Section 3.2.3 that, with high probability, a constant fraction of the set of all choices of s_i works.

In the implementation we have focused on ways to make the memory access pattern more local when computing h_{i0}, h_{i1}, h_{i2}. This is to make better use of the CPU cache. The idea is that the tables used for storing the function descriptions are merged, such that all 6 values looked up using $y_1(x)$ (two in each function ρ_j, where $0 \leq j \leq 2$) are stored in consecutive memory locations, and so on for $y_2(x), \ldots, y_k(x)$.

3.2.3 Analysis of The Shared Function

By Lemma 2 the probability that we fail to get a family of fully random hash functions for all buckets is at most $\sum_i |B_i|\,(|B_i|/2^\delta)^k \leq n(\ell/2^\delta)^k$. If we choose, for example, $\delta =$

$\lceil \log(\sqrt[3]{n}\ell) \rceil$ and $k \geq 4$, we have that this probability is $o(1/n)$. Then, it will succeed with high probability, i.e., $1 - o(1/n)$.

Finally, we need to show that it is possible to obtain, with high probability, a value of s_i such that the functions h_{i0}, h_{i1}, and h_{i2} make the RAM algorithm work for B_i. There are two issues. First, the functions h_{i0}, h_{i1}, and h_{i2} do not produce values that are exactly uniformly distributed in $\{0, \ldots, |B_i| - 1\}$, because $|B_i|$ does not divide p. However, it is not hard to see that the probability of a particular set of hash function values (or, in the analysis of RAM, of a particular graph) is close to the probability in the uniformly distributed case. More specifically, the probability is at most a factor $e^{m_i|B_i|/p}$ higher, because the probability of getting a given set of hash values is upper bounded by $\lceil p/m_i \rceil^{|B_i|}/p^{|B_i|} \leq (1 + m_i/p)^{|B_i|} m_i^{-|B_i|} \leq e^{m_i|B_i|/p} m_i^{-|B_i|}$. Since $p \gg \ell \geq |B_i|$ this means that the failure probability will be very close to the uniform case.

The second issue is to show that even though any single choice of s_i makes RAM fail with constant probability, $\varepsilon_{\text{err}} < 1$, then with high probability there are many values of s_i that will make RAM work. We may assume that the choice of $y_1, \ldots, y_k$ was successful, i.e., that all functions in Definition 20 are fully random on all buckets. The expected number X of choices of s_i that makes the hash functions fail is $\varepsilon_{\text{err}} p$, since there are p possible values for s_i. Lemma 2 tells us that the events that the hash functions fail, for any two different values of s_i, are independent. This means that $\text{Var}(X)$ is bounded by the expectation, and consequently $\text{Var}(X) \leq \varepsilon_{\text{err}} p$. Chebyshev's inequality (see e.g. [58]) then says that the probability that more than $p(1 + \varepsilon_{\text{err}})/2$ choices of s_i make the hash functions fail is bounded by $(1 - \varepsilon_{\text{err}})^2/p$.

3.2.4 Implementation Details

The family of linear hash functions over GF(2) enables us to compute the functions h_0, $y_1, y_2, y_3, \ldots, y_k$ in parallel. The idea is to take a linear function $h' : \{0,1\}^L \rightarrow \{0,1\}^\gamma$ from the family of linear hash functions analyzed in [2] that produces a γ-bit fingerprint for each key $x \in S \subseteq \{0,1\}^L$ with sufficiently many bits, and chop the hash function values into (disjoint) parts. Clearly, these functions will be independent.

The keys in S are mapped to a γ-bit fingerprint set F. The value of γ must be encoded by at least $b + k\delta$ bits so that a single fingerprint will be able to represent the values of functions h_0, $y_1, y_2, y_3, \ldots, y_k$. As the keys in S are assumed to be all distinct, then all fingerprints in F should be distinct as well. As the function h' comes from a family of universal hash functions [2], the probability that there exist two keys that have the same values under all functions is at most $\binom{n}{2}/2^{b+k\delta}$. This probability can be made negligible by choosing k and δ appropriately.

In the implementation we used a function that produces $\gamma = 96$ bits. The 32 most significant bits are used to compute h_0, i.e., $h_0(x) = h'(x)[65, 96] >> (32 - b)$, where $x \in S$ and the symbol $>>$ denotes the right shift of bits. The other 64 bits correspond then to the values of $y_1(x), y_2(x), \ldots y_k(x)$, for $k = 4$, leading to $\delta = 16$. However, to save space for storing the tables used for computing h_{i0}, h_{i1}, and h_{i2}, we hard coded the linear hash function to make the most significant bit of each chunk of 16 bits equal to zero. Therefore, $\delta = 15$.

The last parameter related to the hash functions we need to talk about is the prime number p. It should be chosen as large as possible, and in all cases $p \gg \ell$. In the implementation we set it to the largest 32-bit integer that is prime, i.e, $p = 4294967291$.

Although it is always possible to set up a configuration in which the EM algorithm will work with high probability, the implementation is engineered for $\ell = 256$. We have two reasons for choosing $\ell = 256$. The first one is to keep the bucket size small enough to be represented by 8-bit integers. The second is to allow the memory accesses during the generation time and the resulting function evaluation to be done in the CPU cache most of the time.

In experiments we noticed that the constant κ presented in Eq. (3.1) and in Eq. (3.2) is in the range $0 < \kappa < 1$. For instance, taking $n = 1,024$ billion keys we got $b = 23$ and therefore $\kappa \approx 0.42$. This holds for smaller values of n, see Section 4.3.1. Therefore, based on those experimental results, it is possible to estimate the largest problem we can solve in 32-bit and 64-bit architectures. The largest problem we can solve in a 32-bit architecture is a key set with 500 billion keys. The problem here is that for larger sets more than 32 bits would be required to address a single bucket, i.e., $b > 32$. But in 64-bit architecture we can deal with sets of sizes up to $1,8 \times 10^{21}$ keys with high probability. For larger sets b would require more than 64 bits. We remark that these estimates are based on the constant $\kappa \approx 0.42$ obtained experimentally and this can change for n asymptotically large.

3.3 Conclusions

We have presented a particular engineering of the *split-and-share* technique [30] to simulate a uniform hash function on the small buckets generated by the EM algorithm presented in Chapter 4. The main contribution is that we are able to generate a *family* of uniform hash functions for each bucket with only a constant factor of space overhead. This is necessary because the RAM algorithm needs to be able to choose new hash functions when it fails due to the occurrence of cycles in the random graph induced by the current hash functions.

Chapter 4

The External Cache-Aware Perfect Hashing Algorithm

In this chapter we use a number of techniques from the literature to obtain a novel external memory perfect hashing algorithm, referred to as *EM algorithm*, which is cache-aware. The EM algorithm is for key sets that do not fit in the internal memory. The main novelties are: (i) it uses external memory to allow the generation of PHFs or MPHFs for sets on the order of a billion keys; (ii) it generates the resulting functions without assuming that uniform hash functions are available for free; and (iii) it partitions the input into buckets small enough to fit in the CPU cache.

The EM algorithm produces MPHFs that requires approximately 3.3 bits per key of storage space. For PHFs with range $\{0, \ldots, 1.23n - 1\}$ the space usage drops to approximately 2.7 bits per key. The main insight supporting the EM algorithm is that it splits the incoming key set S into small buckets containing at most $\ell = 256$ keys. Then, a PHF or an MPHF is generated for each bucket and using an *offset* array we obtain a PHF or an MPHF for the whole set S. Therefore, the EM algorithm works on subsets of size lower than 256 and this increases the probability of cache hits. That is why the EM algorithm generates the functions as fast as the algorithms that operate only on data structures stored in internal memory.

The EM algorithm increases one order of magnitude in the size of the greatest key set for which an MPHF was obtained in the literature [12]. This improvement comes from a combination of a novel perfect hashing scheme that greatly simplifies previous methods, and the fact that the EM algorithm is designed to make good use of memory hierarchy. Also, the algorithm is theoretically sound because we have completely analyzed its time and space usage without unrealistic assumptions. This is a accomplished because the RAM algorithm used to generate an MPHF for each bucket uses the hash functions designed in Chapter 3, which simulate uniform hash functions on small buckets.

We demonstrate the scalability of the EM algorithm by considering a set of 1.024 billion strings (URLs from the world wide web of average length 64), for which we construct a MPHF on a commodity PC in approximately 50 minutes. If we use the range $\{0, \ldots, 1.23n - 1\}$, the space for the PHF is less than 324 MB, and we still get hash values that can be represented in a 32 bit word. Certainly, the EM algorithm will be useful for a number of current and practical data management problems that were not possible before. A previous version of the EM algorithm was presented in [15].

This chapter is organized as follows. In Section 4.1 we present the EM algorithm. In Section 4.2 we analyze the EM algorithm. In Section 4.3 we evaluate the EM algorithm experimentally. Finally, in Section 4.4 we conclude this chapter.

4.1 Design of the EM Algorithm

The EM algorithm is also a two-step randomized algorithm of Las Vegas type because it uses the Las Vegas type RAM algorithm in its second step, as illustrated in Figure 1.2. The first step, referred to as *partitioning step*, takes a key set $S \subseteq \{0,1\}^L$ and uses a hash function $h_0 : S \to \{0,1\}^b$ to partition S into $N_b = 2^b$ buckets for some integer b. The second step, referred to as *searching step*, generates a PHF or an MPHF for each bucket i, $0 \leq i < N_b$, and computes the *offset* array. The PHFs or MPHFs for the buckets are generated with the version of the RAM algorithm described in Section 2.2.4.

The EM algorithm generates a family $\mathcal{J}$ of PHFs or MPHFs, defined as follows:

Definition 21 Let $B_i = \{x \in S \mid h_0(x) = i\}$ denote the ith bucket. Let $f_i \in \mathcal{F}(\mathcal{PHF}, \mathcal{MPHF})$ denote a PHF if $f_i \in \mathcal{PHF}$ or an MPHF if $f_i \in \mathcal{MPHF}$ on B_i. Let M_i be the maximum value of f_i on B_i plus one, and $\mathit{offset}[i] = \sum_{j=0}^{i-1} M_i$. Note that, if $f_i \in \mathcal{MPHF}$, then $M_i = |B_i|$. Let $\mathcal{H}$ be the family of linear hash functions presented in Section 3.1. Therefore,

$$\mathcal{J}(\mathcal{F}, \mathcal{H}) = \{f \mid f(x) = f_i(x) + \mathit{offset}[i], i = h_0(x), f_i \in \mathcal{F}, h_0 \in \mathcal{H}\} \tag{4.1}$$

is a family of PHFs or MPHFs for the whole set S. Thus, the problem is reduced to computing and storing the function f_i for each bucket and the *offset* array.

Now we are going to design and analyze the EM algorithm to prove the following theorem:

Theorem 7 For a given key set $S \subseteq \{0,1\}^L$ with n keys, where $L = O(1)$ is the maximum key length in bits, the family $\mathcal{H}$ of all linear transformations over GF(2), a function $h_0 : S \to \{0,1\}^b$ taken uniformly at random from $\mathcal{H}$, an induced set of buckets $\xi = \{B_i \mid B_i = \{x \in S \mid h_0(x) = i\}\}$, where $|\xi| = N_b = 2^b$, $\max |B_i| = \ell$, $b \leq \log n + \log\log\log n - \log \ell + \log \kappa$, $\ell \geq \kappa \log n \log\log n$, for $\kappa > 0$, it is possible to find in expected linear time all functions $f_i \in \mathcal{F}$, $0 \leq i < N_b$, and the *offset* array so that any function $f \in \mathcal{J}$ can be computed in $O(1)$ time and described in $\log(3)cn + o(n) + O(\log n)$ bits if f is a PHF, and in $(2+\epsilon)cn + o(n) + O(\log n)$ bits if f is an MPHF, where $c \geq 1.23$ and $\epsilon > 0$. For that $O(N_f)$ computer words are required, where $N_f = \Omega(n^\tau)$ and $0 < \tau < 1$.

We consider the situation in which the set of all keys may not fit in the internal memory and so the first step of the algorithm is necessary to deal with the keys stored on disk to form the buckets. The EM algorithm first scans the list of keys and computes the hash function values that will be needed afterwards in the algorithm. These values will (with high probability) distinguish all keys, so we can discard the original keys. It is well known that hash values of at least $2 \log n$ bits are required to make this work. Thus, for sets of a billion keys or more we cannot expect the list of hash values to fit in the internal memory of a standard PC.

To form the buckets we sort the hash values of the keys according to the value of h_0. Since we are interested in scalability for large key sets, this is done using an implementation of an external memory mergesort [53] with some nuances to make it work in linear time. The total work on disk consists of reading the keys, plus writing and reading the hash function values once. Since the h_0 hash values are relatively small (less than 15 decimal digits) we can use radix sort to do the internal memory sorting.

We have designed two versions of the EM algorithm. The first one uses the hash functions described in Section 3.2, which guarantee that the EM algorithm can be made to work for every key set with high probability. The second one uses faster and more compact pseudo random hash functions proposed in [50], referred to as *heuristic EM algorithm* from now on, because it is not guaranteed that it can be made to work for every key set. However, empirical studies show that limited randomness properties are often as good as total randomness in practice [2], and the heuristic EM has worked for all key sets we have applied it to so far.

Figure 4.1 presents a pseudo code for the EM algorithm. The detailed description of the partitioning and searching steps are presented in Sections 4.1.1 and 4.1.2, respectively. The internal algorithm presented in Section 2.2.4 uses three hash functions h_{i0}, h_{i1}, and h_{i2} to compute a function $f_i \in \mathcal{F}$. These hash functions, as well as the hash function h_0 used in the partitioning step of the algorithm, were described in Section 3.2.

```
function EM ($S$, $\mathcal{H}$, $\{f_0, f_1, \dots f_{N_b-1}\}$, $offset$)
    Partitioning ($S$, $\mathcal{H}$, $Files$)
    Searching ($Files$, $\{f_0, f_1, \dots f_{N_b-1}\}$, $offset$)
```

Figure 4.1: The EM algorithm.

4.1.1 Partitioning Step

The partitioning step performs two important tasks. First, the variable-length keys are mapped to γ-bit fingerprints by using a linear hash function $h' : S \to \{0,1\}^\gamma$ taken uniformly at random from the family $\mathcal{H}$ of linear hash functions presented in Section 3.1. That is, the variable-length key set $S \subseteq \{0,1\}^L$ is mapped to a fixed-length key set F of fingerprints. Second, the set S of n keys is partitioned into N_b buckets, where b is a suitable parameter chosen to guarantee that each bucket has at most $\ell = \Omega(\log n \log\log n)$ keys with high probability (see Eq. (3.2)). It outputs a set of *Files* containing the buckets, which are merged in the searching step when the buckets are read from disk. Figure 4.2 presents the partitioning step.

The critical point in Figure 4.2 that allows the partitioning step to work in linear time is the internal sorting algorithm. We have two reasons to choose radix sort. First, it sorts each key block $\mathcal{B}_j$ in linear time, since keys are short integer numbers (less than 15 decimal digits). Second, it just needs $O(|\mathcal{B}_j|)$ words of extra memory so that we can control the memory usage independently of the number of keys in S.

At this point one could ask: why not to use the well known replacement selection algorithm to build files larger than the internal memory area size? The reason is that the radix sort algorithm sorts a block $\mathcal{B}_j$ in time $O(|\mathcal{B}_j|)$ while the replacement selection algorithm requires $O(|\mathcal{B}_j| \log |\mathcal{B}_j|)$. We have tried out both versions and the one using the radix sort algorithm outperforms the other. A worthwhile optimization we have used is the last run optimization proposed in [53], where the last block is kept in memory instead of dumping it to disk to be read again in the second step of the algorithm.

```
function Partitioning ($S$, $\mathcal{H}$, $Files$)
    ▶ Let $\beta$ be the size in bytes of the fixed-length key set $F$
    ▶ Let $\mu$ be the size in bytes of an a priori reserved internal memory area
    ▶ Let $N_f = \lceil \beta/\mu \rceil$ be the number of key blocks that will be read from disk
      into an internal memory area

select $h'$ uniformly at random from $\mathcal{H}$
for $j = 1$ to $N_f$ do
    DiskReader ($S_j$) {read a key block $S_j$ from disk}
    Hashing ($S_j$, $\mathcal{B}_j$) {store $h'(x)$, for each $x \in S_j$, into $\mathcal{B}_j$, where $|\mathcal{B}_j| = \mu$}
    BlockSorter ($\mathcal{B}_j$) {cluster $\mathcal{B}_j$ into $N_b$ buckets using an indirect radix sort algorithm that
                                   takes $h_0(x)$ for $x \in S_j$ as sorting key (i.e, the $b$ most significant bits
                                   of $h'(x)$) and if any bucket $B_i$ has more than $\ell$ keys restart in the
                                   partitioning step}
    BlockDumper ($\mathcal{B}_j$, $Files[j]$) {dump $\mathcal{B}_j$ to disk into $Files[j]$}
```

Figure 4.2: Partitioning step.

Figure 4.3(a) shows a *logical* view of the N_b buckets generated in the partitioning step. In reality, the γ-bit fingerprints belonging to each bucket are distributed among many files, as depicted in Figure 4.3(b). In the example of Figure 4.3(b), the γ-bit fingerprints in bucket 0 appear in files 1 and N_f, the γ-bit fingerprints in bucket 1 appear in files 1, 2 and N_f, and so on.

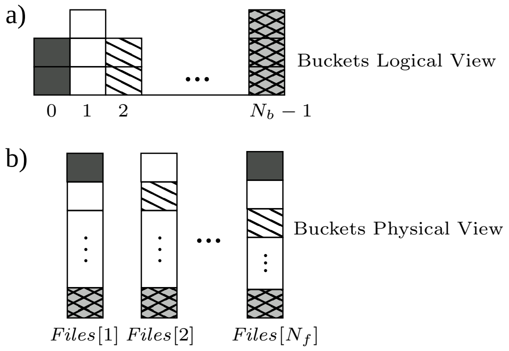


Figure 4.3: Situation of the buckets at the end of the partitioning step: (a) Logical view (b) Physical view.

This scattering of the γ-bit fingerprints in the buckets could generate a performance problem because of the potential number of seeks needed to read the γ-bit fingerprints in each bucket from the N_f files on disk during the second step. But, as we show afterwards in Section 4.2.1, the number of seeks can be kept small by using buffering techniques.

4.1.2 Searching Step

Figure 4.4 presents the searching step. The searching step is responsible for generating a function $f_i \in \mathcal{F}$ for each bucket (using the RAM algorithm presented in Section 2.2.4) and for computing the *offset* array. Statement 1 of Figure 4.4 constructs a heap H of size N_f, which is well known to be linear on N_f. The order relation in H is given by the bucket address i (i.e., the b most significant bits of $x \in F$). Statement 4 has four steps. In statement 5, a bucket is read from disk, as described below. In statement 6, a function f_i is generated for each bucket B_i using the internal random access memory algorithm presented in Section 2.2.4. In statement 7, the next entry of the *offset* array is computed. Finally, statement 8 writes the description of f_i and *offset*$[i]$ to disk. Note that to compute *offset*$[i+1]$ we just need M_i (i.e., the maximum value of f_i in bucket B_i) and *offset*$[i]$. So, we just need to keep two entries of the *offset* array in memory all the time.

```
function Searching ($Files$, $\{f_0, f_1, \dots f_{N_b-1}\}$, $offset$)
    ▶ Let $H$ be a minimum heap of size $N_f$
    ▶ Let the order relation in $H$ be given by
      $i = x[\gamma - b + 1, \gamma]$ for $x \in F$

1. for $j = 1$ to $N_f$ do { Heap construction }
2.     Read the first $\gamma$-bit fingerprint $x$ from $Files[j]$ on disk
3.     Insert $(i, j, x)$ in $H$
4. for $i = 0$ to $N_b - 1$ do
5.     BucketReader ($Files$, $H$, $B_i$) {Read bucket $B_i$ from disk driven by heap $H$}
6.     if MPHFGen ($B_i$, $f_i$) fails then
           Restart the partitioning step
7.     $offset[i + 1] = offset[i] + |M_i|$
8.     MPHFDumper ($f_i$, $offset[i]$) {Write the description of $f_i$ and $offset[i]$ to the disk}
```

Figure 4.4: Searching step.

The algorithm to read bucket B_i from disk is presented in Figure 4.5. Bucket B_i is distributed among many files and the heap H is used to drive a multiway merge operation. Statement 2 extracts and removes triple (i, j, x) from H, where i is a minimum value in H. Statement 3 inserts x in bucket B_i. Statement 4 performs a seek operation in *Files*$[j]$ on disk for the first read operation and reads sequentially all γ-bit fingerprints $x \in F$ that have the same index i and inserts them all in bucket B_i. Finally, statement 5 inserts in H the triple (i', j, x'), where $x' \in F$ is the first γ-bit fingerprint read from *Files*$[j]$ (in statement 4) that does not have the same bucket address as the previous keys.

```
function BucketReader ($Files$, $H$, $B_i$)
1. while bucket $B_i$ is not full do
2.     Remove $(i, j, x)$ from $H$
3.     Insert $x$ into bucket $B_i$
4.     Read sequentially all $\gamma$-bit fingerprints from $Files[j]$ that have the same $i$
       and insert them into $B_i$
5.     Insert the triple $(i', j, x')$ in $H$, where $x'$ is the first $\gamma$-bit fingerprint read
       from $Files[j]$ that does not have the same bucket index $i$
```

Figure 4.5: Reading a bucket.

4.2 Analytical Results

4.2.1 The Linear Time Complexity

In this section we show that the EM algorithm runs in expected $O(n)$ time. For that end we need to show that the partitioning and searching steps run in expected $O(n)$ time.

Analysis of the Partitioning Step

The partitioning step presented in Figure 4.2 runs in expected $O(n)$ time. As in statement 1 we need to select a function h' from the family $\mathcal{H}$ of linear hash functions presented in Section 3.1 and each function h' is described in $O(L \log n)$ bits, this statement has cost $O(L)$ in the *Word RAM* model of computation [41] with a word size equal to $\log n$ bits (recall that L is the maximum key length in bits). Each iteration of the loop **for** in statement 2 runs in $O(|\mathcal{B}_j|)$ time, $1 \leq j \leq N_f$, where $|\mathcal{B}_j|$ is the number of γ-bit fingerprints that fit in block $\mathcal{B}_j$ of size μ. This is because statement 3 just read $|\mathcal{B}_j|$ keys from disk, statement 4 compute the related fingerprints and stores them all into the internal memory area of size μ, statement 5 runs a radix sort algorithm that is well known to be linear on the number of keys it sorts (i.e., $|\mathcal{B}_j|$ γ-bit fingerprints), and statement 6 just dumps $|\mathcal{B}_j|$ γ-bit fingerprints to the disk into *File*$[j]$. Thus, the loop **for** runs in $\sum_{j=1}^{N_f} O(|\mathcal{B}_j|)$ time. As $\sum_{j=1}^{N_f} |\mathcal{B}_j| = n$, then the partitioning step runs in expected $O(n)$ time. It is expected because the partitioning step can fail in statement 5 whenever a bucket with more than ℓ keys is generated. However, it will succeed with high probability, as showed in Section 3.2.3 and, in turn, the number of iterations is $O(1)$.

Analysis of the Searching Step

The searching step presented in Figure 4.4 also runs in expected $O(n)$ time. It is expected because the RAM algorithm used for the buckets is a randomized algorithm that can fail with small probability for a given bucket, when we cannot find appropriated hash functions h_{i0}, h_{i1} and h_{i2}. When it fails, we restart in the partitioning step. By using the hash functions designed in Section 3.2, it is possible to make the searching step work with high probability, then the number of iterations will be bounded by a constant.

Let us, first, analyze the number of heap operations performed in statement 5, which reads $|B_i|$ γ-bit fingerprints of bucket B_i and is detailed in Figure 4.5. It is well known that the heap construction of statement 1 runs in $O(N_f)$ time. Each iteration of statement 4 performs two heap operations in statement 5 (see statements 2 and 5 in Figure 4.5) and each one costs $O(\log N_f)$. So, the total cost of statement 4 in terms of heap operations is $2 \times N_b \times O(\log N_f)$. Considering that: (i) $N_b < \frac{n}{\log n}$ and (ii) $N_f \ll n$, we can conclude that the number of heap operations is $O(n)$.

However, in the worst case the γ-bit fingerprints of bucket i are spread in at most ℓ files on disk (recall that ℓ is the maximum number of keys found in any bucket). Therefore, we need to take into account that the critical step in reading a bucket is in statement 4 of Figure 4.5, where a seek operation in $\mathit{Files}[j]$ may be performed by the first read operation.

In order to amortize the number of seeks performed we use a buffering technique [51]. We create a buffer j of size $q = \mu/N_f$ for each file j, where $1 \leq j \leq N_f$ (recall that μ is the size in bytes of an a priori reserved internal memory area). Every time a read operation is requested to file j and the data is not found in the jth buffer, q bytes are read from file j to buffer j. Hence, the number of seeks performed in the worst case is given by β/q (remember that β is the size in bytes of the fixed-length key set F). For that we have made the pessimistic assumption that one seek happens every time buffer j is filled in. Thus, the number of seeks performed in the worst case is $\gamma n/8q$, since after the partitioning step we are dealing with γ-bit fingerprints instead of 64-byte URLs, on average. Therefore, the number of seeks is linear on n and amortized by q.

It is important to emphasize two things. First, the operating system uses techniques to diminish the number of seeks and the average seek time. This makes the amortization factor to be greater than q in practice. Second, almost all main memory is available to be used as file buffers because just the γ-bit fingerprints of the bucket being processed and $O(N_f)$ words for the heap must be kept in main memory during the searching step.

To conclude the searching step analysis we need to show that statements 6 and 8 perform a number of operations proportional to $|B_i|$. If this is true, then the rest of statement 4 runs in $\phi \sum_{i=0}^{N_b-1} |B_i|$ time, where ϕ is a machine-dependent constant.

Statement 6 runs the algorithm used to generate the function f_i of each bucket. That algorithm is linear on the number of keys it is applied to, as we have shown in Section 2.2.1. As it is applied to buckets with $|B_i|$ keys, then statement 6 performs a number of operations proportional to $|B_i|$.

Statement 8 has time complexity proportional to $|B_i|$ because it writes to disk the description of each generated function f_i and each description is stored in $O(|B_i|)$ bits (see Section 2.2.2 for details). As $\sum_{i=0}^{N_b-1} |B_i| = n$, then statement 4 runs in $O(n)$ time. In conclusion, the EM algorithm takes expected $O(n)$ time because both the partitioning and the searching steps run in expected $O(n)$ time.

4.2.2 The Space Requirements to Describe the Functions

The description of the resulting functions is compounded by the function h_0, the *offset* array, and the functions $f_i \in \mathcal{F}(\mathcal{PHF}, \mathcal{MPHF})$, $0 \leq i < N_b$. Remember that b is given by Eq. (3.1) and $N_b < n/\log n$. The function h_0 comes from the family $\mathcal{H}$ of linear hash functions over GF(2) and therefore requires $O(L \log n)$ bits to be stored. By assuming that $L = O(1)$, then h_0 takes $O(\log n)$ bits of space. The *offset* array has N_b entries of $\log n$ bits and, then, requires $o(n)$ bits since $N_b < n/\log n$.

To store each function f_i, if $f_i \in \mathcal{PHF}$ then it requires $|f_i| = \log(3)c|B_i|$ bits of space, for $c \geq 1.23$. If $f_i \in \mathcal{MPHF}$ then it requires $|f_i| = (2+\epsilon)c|B_i| + o(|B_i|)$ bits of space, for $c \geq 1.23$ and $\epsilon > 0$. Therefore, $\sum_{i=0}^{N_b-1} |f_i| = \log(3)cn$ bits if f_i is a PHF, and $\sum_{i=0}^{N_b-1} |f_i| = (2+\epsilon)cn + o(n)$ bits if f_i is an MPHF.

Additionally, we need to store the hash functions h_{i0}, h_{i1}, and h_{i2} (see Definition 20). For this we need to store $6k$ tables with 2^δ entries of $\log p$ bits, where p is a large prime number, and the seed numbers s_i of $\log p$ bits, where $0 \leq i < N_b$. We can assume with no loss of generality that $\log p = O(\log n)$. Therefore, as $\delta = \lceil \log(\sqrt[3]{n}\ell) \rceil$ and $k = 4$ are values chosen to make the EM algorithm to work with high probability and $N_b < n/\log n$, then, h_{i0}, h_{i1}, and h_{i2} are stored in $o(n) + o(n) = o(n)$ bits.

Putting this all together, we have that the number of bits required to store a resulting function $f \in \mathcal{J}$ is $\log(3)cn + o(n) + O(\log n)$ bits if f is a PHF and $(2+\epsilon)cn + o(n) + O(\log n)$ bits for $\epsilon > 0$ if f is an MPHF.

4.2.3 The Space Requirements to Generate the Functions

In this section we show that the EM algorithm presented in Figure 4.1 needs $O(N_f)$ computer words of main memory to generate the functions of $\mathcal{J}$. We need to maintain the following data structures in internal memory.

1. The internal memory area of size μ bytes to be used in the partitioning step and in the searching step. The size μ is fixed a priori and depends only on the amount of internal memory available to run the algorithm (i.e., it does not depend on the size n of the problem).

2. The main memory required to run the indirect radix sort algorithm. It just needs $O(|\mathcal{B}_j|)$ words of extra memory so that we can control the memory usage independently of the size of the problem and can be fixed a priori.

3. The additional space required is $O(N_f)$ computer words that corresponds to the size of the heap H used to drive a N_f-way merge operation in the searching step, which allows the merge operation to be performed in one pass through each file.

Therefore, as the memory usage in the partitioning step does not depend on the number of keys in S and, in the searching step, the internal algorithm is applied to problems of size up to ℓ keys, we can conclude that the EM algorithm requires $O(N_f)$ computer words to generate a function $f \in \mathcal{J}$. As shown in [1, Theorem 3.1], to get a linear time complexity we need $N_f = \Omega(n^\tau)$ computer words for $0 < \tau < 1$ and to allow the merge operation to be performed in one pass we need $\tau = 0.5$. This ends the proof of Theorem 7.

4.3 Experimental Results

The purpose of this section is to evaluate the performance of the EM algorithm and to compare it with both the RAM algorithm presented in Chapter 2 and the algorithm by Fox, Chen and Heath [38] (referred to as FCH). We do not consider the other practical perfect hashing algorithms compared with the RAM algorithm in Section 2.3.2 because the RAM algorithm outperforms them in the same experimental setup. In Section 4.3.1 we consider key sets that cannot be handled in internal memory. In this case, the partitioning in small buckets and the use of external memory are needed by the EM algorithm. The experimental results for the EM algorithm match the analytical results presented in Section 4.2. In Section 4.3.2 we carry out the comparison.

The algorithms were implemented in the C language and are available under the GNU Lesser General Public License (LGPL) at `http://cmph.sf.net`. The experiments were carried out on a computer running the Linux operating system, version 2.6, with a 1.86 gigahertz Intel Core 2 processor with a 4 megabyte L2 cache and 1 gigabyte of main memory. For the experiments we used the same two collections considered in Chapter 2: (i) a set of 150 million randomly generated 4 byte long IP addresses, and (ii) a set of $1,024$ million 64 byte long (on average) URLs collected from the Web.

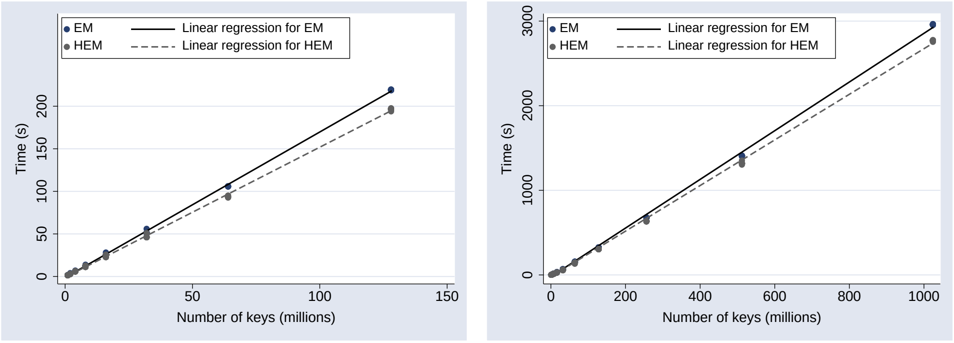
To compare the algorithms we used the following metrics: (i) The amount of time to generate PHFs or MPHFs, referred to as Generation Time. (ii) The space requirement for the description of the resulting PHFs or MPHFs to be used at retrieval time, referred to as Storage Space. (iii) The amount of time required by a PHF or an MPHF for each retrieval, referred to as Evaluation Time.

4.3.1 Performance of the EM Algorithm

In this section we evaluate the performance of the EM algorithm considering generation time and storage space as metrics. First, we are interested in verifying the claim that the EM algorithm runs in linear time. Therefore, we run both versions of the algorithm for several numbers n of keys in the two collections. The values chosen for n were 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 and $1,024$ million keys. The size μ of the a priori reserved internal memory area was set to 250 megabytes. Subsequently, we show how μ affects the runtime of the algorithm. The parameter b (see Eq. (3.1)) was set to the minimum value that gives us a maximum bucket size lower than $\ell = 256$. For each value chosen for n, the respective values for b are $13, 14, 15, 16, 17, 18, 19, 20, 21, 22$ and 23 bits.

In order to estimate the number of trials for each value of n we used a statistical method for determining a suitable sample size [47, Chapter 13]. We got that just one trial for each n would be enough with a confidence level of 95%. However, we conducted 25 trials. This number of trials seems rather small but, as shown below, the behavior of the EM algorithm is very stable and its runtime is almost deterministic (i.e., the standard deviation is very small) because it is a random variable that follows a (highly concentrated) normal distribution.

Figure 4.6 presents the runtime for each trial in the two collections. In addition, the solid and dashed lines correspond to a linear regression model obtained from the experimental measurements for both: (i) the EM algorithm and (ii) the heuristic EM algorithm (HEM). For both versions of the EM algorithm the coefficient of determination R^2 is 99%. As we were expecting, the runtime for a given n has almost no variation and the heuristic EM algorithm is faster than the EM algorithm because it uses a faster pseudo random hash function, as explained later in this section.



(a) IPs collection

(b) URLs collection

Figure 4.6: Number of keys in S versus generation time for the EM algorithm and the heuristic EM algorithm. The solid and dashed lines correspond to a linear regression model for the generation time ($R^2 = 99\%$).

An intriguing observation is that the runtime of both versions of the EM algorithm is almost deterministic. A given bucket i, $0 \leq i < N_b$ (recall that $N_b = 2^b$), is a small key set (at most 256 keys) and, the runtime of the building block algorithm is a random variable X_i that follows a geometric distribution with mean $1/Pr_a \approx 1$, because $Pr_a \to 1$ as $n \to \infty$ for the RAM algorithm where $r = 3$. Let $Y = \sum_{0 \leq i < N_b} X_i$ denote the runtime of the searching step of the EM algorithm. Under the hypothesis that the X_i are independent and bounded, the *law of large numbers* (see, e.g., [47]) implies that the random variable Y/N_b converges to a constant as $n \to \infty$. This and the fact that the partitioning step was never restarted because the parameter b is chosen so that the maximum bucket size ℓ is lower than or equal to 256 with high probability explains why the runtime is almost deterministic.

The next important metric on PHFs and MPHFs is the space required to store the functions. Table 4.1 shows that the EM algorithm can be used for constructing PHFs and MPHFs that require on average 2.6 and 3.21 bits per key to be stored, respectively. It also shows that the heuristic EM algorithm outputs PHFs and MPHFs that require on average 2.51 and 3.1 bits per key to be stored, respectively.

The lookup tables used by the hash functions of the EM algorithm require a fixed storage cost of $3,345,409$ bytes and this cost is not considered in Table 4.1. To avoid the space needed for lookup tables we have implemented the heuristic EM algorithm. It uses

n	b	EM (Bits/key)		Heuristic EM (Bits/key)	
		PHF	MPHF	PHF	MPHF
10^5	9	2.41	3.00	2.32	3.04
10^6	13	2.67	3.29	2.54	3.12
10^7	16	2.53	3.13	2.42	2.97
10^8	20	2.74	3.34	2.70	3.21
10^9	23	2.67	3.29	2.55	3.12

Table 4.1: Space usage to respectively store the resulting PHFs and MPHFs of the EM algorithm and the Heuristic EM algorithm.

the pseudo random hash function proposed in [50] to replace the hash functions described in Chapter 3. The Jenkins function just loops around the key, doing bitwise operations over chunks of 12 bytes. Then, it returns the last chunk. Thus, in the mapping step, the key set S is mapped to F, which contains 12-byte long fingerprints (recall that $\gamma = 96$ bits).

The Jenkins function needs just one random seed of 32 bits to be stored instead of quite long lookup tables, a major improvement over the $3,345,409$ bytes necessary to implement truly random hash functions on the buckets. Therefore, there is no fixed cost to store the resulting MPHFs, but three random seeds of 32 bits are required to describe the functions h_{i0}, h_{i1} and h_{i2} of each bucket. As a consequence, the MPHFs generation is faster (see Figure 4.6). The reason is that there are no large lookup tables to cause *cache misses*. For example, the generation time for a set of $1,024$ million URLs has dropped from 49.3 down to 46.2 minutes in the same setup. The disadvantage of using the Jenkins function is that there is no formal proof that it works for every key set. That is why the hash functions we have designed in Chapter 3 are required, even being slower. In the implementation available, the user can choose the hash functions to be used.

Controlling Disk Accesses

In order to lower the number of seek operations on disk we benefit from the fact that both versions of the EM algorithm leave almost all main memory available to be used as disk I/O buffer. In this section we evaluate how much the parameter μ affects the runtime of both versions of the EM algorithm. For that we fixed n in $1,024$ million URLs and used μ equal to 100, 200, 300, 400, 500, and 600 megabytes.

Table 4.2 presents the number of files N_f, the buffer size q used for all files, the number of seeks β/q in the worst case, considering the pessimistic assumption mentioned in Section 4.2.1, and the time to generate a PHF or an MPHF for $1,024$ million URLs as a function of the amount of internal memory available. Remember that β is the size in bytes of the fixed-length key set F, and $\beta = 12n$ for both the EM algorithm and the heuristic EM algorithm. Observing Table 4.2 we noticed that the time spent in the generation decreases as the value of μ increases. However, for $\mu > 400$, the time variation is not as significant as for $\mu \leq 400$. This can be explained by the fact that the kernel 2.6 I/O scheduler of Linux has smart policies for avoiding seeks and diminishing the average seek time (see `http://www.linuxjournal.com/article/6931`).

μ (MB)	EM				Heuristic EM			
	N_f	q (KB)	β/q	time (min)	N_f	q (KB)	β/q	time (min)
100	301	340	$47,059$	59.8	226	453	$26,491$	56.0
200	119	$1,721$	$9,297$	50.0	89	$2,301$	$5,216$	46.4
300	74	$4,151$	$3,855$	48.5	56	$5,485$	$2,188$	45.3
400	54	$7,585$	$2,110$	47.2	41	$9,990$	$1,202$	44.4
500	43	$11,906$	$1,344$	47.0	32	$16,000$	750	44.0
600	35	$17,554$	912	47.0	26	$23,630$	508	44.0

Table 4.2: Influence of the internal memory area size (μ) in the runtime of both versions of the EM algorithm to construct PHFs or MPHFs for 1.024 billion URLs (time in minutes).

4.3.2 Comparison with RAM and FCH Algorithms

We used the hash function presented in [50] for all the algorithms, except for the EM algorithm, where we used the one designed in Chapter 3. For all the experiments we used $n = 3,541,615$ keys for the two collections. The reason to choose a small value for n is because the FCH algorithm has exponential time on n for the generation phase, and the times explode even when a number of keys are a little over.

We first compare the EM algorithm for constructing MPHFs with both the RAM and FCH algorithms, considering generation time and storage space. Table 4.3 shows that the RAM algorithm for $r = 3$, the EM and heuristic EM algorithms are faster than the FCH algorithm in generating MPHFs. The performance of both versions of the EM algorithm is quite surprising once they use external memory at generation time and the other algorithms do not. The reason is twofold. First, both versions of the EM algorithm simply scan the whole key set once and maps it to a set of fixed length fingerprints. Second, as the whole key set is broken into buckets with at most 256 keys, the memory is accessed in a less random fashion which implies fewer cache misses.

Algorithms		Generation Time (sec)		Storage Space	
		IPs	URLs	Bits/Key	Size (MB)
RAM	$r = 2$	11.39 ± 1.33	16.73 ± 1.89	3.60	1.52
	$r = 3$	5.46 ± 0.01	6.74 ± 0.01	2.62	1.11
EM		5.86 ± 0.17	7.68 ± 0.22	3.31	1.40
Heuristic EM		5.56 ± 0.16	6.27 ± 0.11	3.17	1.34
FCH		$2,052.7 \pm 530.96$	$2,400.1 \pm 711.60$	4.22	1.78

Table 4.3: Comparison of the algorithms for constructing MPHFs considering generation time and storage space, and using $n = 3,541,615$ for the two collections.

It is also important to note that the resulting functions of the RAM and EM algorithms are the most compact functions. The storage space requirements in bits per key for the two versions of the RAM algorithm are 3.6 when $r = 2$, and 2.62 when $r = 3$. For the EM and heuristic EM algorithms the storage space requirements are 3.21 and 3.17 bits per key, respectively. Therefore, the RAM algorithm is the best choice for sets that can be handled in main memory and the EM algorithm is the first one that can be efficiently applied to sets that do not fit in main memory. We remark that the EM algorithm can also be applied to key sets that can be handled in internal memory and the RAM algorithm fails when applied to them, because the RAM algorithm assumes uniform hashing for free and the EM algorithm does not.

We now compare the algorithms considering evaluation time. Table 4.4 shows the evaluation time for a random permutation of the n keys. In this experiment the only resulting MPHF that does not fit entirely in the machine's L2 cache is the one generated by the EM algorithm. This is because the size of the lookup tables used to compute the functions. That is why they are the slowest functions. The MPHFs generated by the FCH algorithm are the fastest ones because they are optimal in terms of memory probes, as the ones by Pagh [61]. That is, just one memory probe is performed in their computation (see the form of those MPHFs in Section 1.6.3.) Thus, the more compact an MPHF is, the more efficient it is if its description fits in the cache. However, functions that carry out less memory probes are preferred. The main problem with the FCH algorithm is the time to generate a MPHF, which is exponential on n.

Algorithms		RAM		EM	Heuristic EM	FCH
		$r = 2$	$r = 3$			
Evaluation Time (sec)	IPs	1.19	1.16	2.72	1.75	0.75
	URLs	2.12	2.11	4.36	2.73	1.61

Table 4.4: Comparison of the algorithms considering evaluation time and using the collections IPs and URLs with $n = 3,541,615$.

Finally, we compare the PHFs and MPHFs generated by the different versions of the RAM and EM algorithms. Table 4.5 shows that the generation times for PHFs and MPHFs are almost the same, with the algorithms for $r = 3$ being the fastest because the probability of obtaining an acyclic 3-graph for $c = 1.23$ tends to one, whereas the probability for a 2-graph where $c = 2.09$ tends to 0.29 (see Sections 2.2.1). For PHFs with $m = 1.23n$, the storage requirement drops from 2.62 to 1.95 bits per key when $r = 3$. The PHFs with $m = 2.09n$, and $m = 1.23n$ are the fastest at evaluation time because no ranking or packing information needs to be computed. The slowest MPHFs are generated by the EM algorithm. Nevertheless, the difference is not so significant (each key can be evaluated in few microseconds) and the EM algorithm is the first efficient option for sets that do not fit in main memory.

RAM		m	Generation Time (sec)		Eval. Time (sec)		Sto. Space
r	Packed		IPs	URLs	IPs	URLs	Bits/Key
2	no	$2.09n$	10.50 ± 1.24	14.79 ± 1.58	0.68	1.63	2.09
	yes	n	11.39 ± 1.33	16.73 ± 1.89	1.19	2.12	3.60
3	no	$1.23n$	5.51 ± 0.01	6.76 ± 0.01	0.79	1.68	2.46
	yes	$1.23n$	5.54 ± 0.01	6.78 ± 0.02	0.79	1.71	1.95
	no	n	5.46 ± 0.01	6.74 ± 0.01	1.16	2.11	2.62
EM		$1.23n$	5.82 ± 0.17	7.34 ± 0.05	2.27	3.97	2.76
		n	5.86 ± 0.17	7.68 ± 0.22	2.72	4.36	3.31
Heuristic EM		$1.23n$	5.47 ± 0.16	5.97 ± 0.09	1.44	2.43	2.62
		n	5.56 ± 0.16	6.27 ± 0.11	1.75	2.73	3.17

Table 4.5: Comparison of the PHFs and MPHFs generated by our algorithms, considering generation time, evaluation time and storage space metrics using $n = 3,541,615$ for the two collections. For packed schemes see Sections 2.2.3 and 2.2.4.

It is important to emphasize that the RAM and FCH algorithms, as well as the other ones considered in Section 2.3.2 were analyzed under uniform hashing assumption. Therefore, the EM algorithm is the first one that has experimentally proven practicality for large key sets and with both space usage for representing the resulting functions and the generation time having been carefully proven. Additionally, it constructs the functions efficiently and the resulting functions are much simpler than the ones generated by previous theoretically well-founded schemes so that they can be used in practice. Furthermore, it considerably improves the first step taken by Pagh with his hash and displace method [61] in the way it joins theory and practice for perfect hashing.

4.4 Conclusions

In this chapter we presented a time efficient, highly scalable and near space-optimal perfect hashing algorithm. The basic idea to obtain scalability is to partition the input key set into small buckets. It is an external memory algorithm suitable for key sets larger than the size of the internal memory available. In this case, it partitions the input key set into small buckets such that each bucket fits in the CPU cache and then was called cache-aware external memory algorithm (EM).

We perform an external sorting to partition the input key set into small buckets. Then, we handle each bucket separately. Splitting the problem into small buckets has both theoretical and practical implications. From the theoretical point of view we show how to simulate fully random hash functions on the small buckets, being able to prove that the EM algorithm will work for every key set with high probability. From the practical point of view we show how to make buckets that are small enough to fit in the CPU cache, resulting in a significant speedup in processing time per element compared to other methods known in the literature.

The dominating phase in the construction of the functions consists of external sorting n fingerprints of $O(\log n)$ bits in $O(n)$ time. The construction algorithm is highly scalable because it uses a little amount of internal memory to work, basically the space necessary to accommodate a heap that drives a multiway merge operation, which is $O(n^\epsilon)$ computer words to have linear time complexity, where $0 < \epsilon < 1$. In our case, as we want to perform the merge operation in one pass, we need $\epsilon = 0.5$ (see, e.g., [1, Theorem 3.1]).

The space necessary to describe the functions takes a constant number of bits per key. The space usage depends on the relation between the size m of the hash table and the size n of the input. For $m = n$, the space usage is in the range $3.1n$ to $3.3n$ bits, depending on which version of the algorithm is used (i.e., EM or Heuristic EM). For $m = 1.23n$ the space usage is in the range $2.5n$ to $2.7n$ bits. In all cases, this is within a small constant factor from the information theoretical minimum of approximately $1.44n$ bits for MPHFs and $0.89n$ bits for PHFs, something that has not been achieved by previous algorithms, except asymptotically for very large n. The resulting functions are evaluated for a given element of a key set in constant time.

The algorithm is theoretically well understood and is the first one with theoretical properties that scale for billions of keys and can be used in practice. We have illustrated the scalability of our algorithm by constructing an MPHF for a set of 1.024 billion URLs from the World Wide Web of average length 64 characters in approximately 50 minutes, using a commodity PC.

Finally, the algorithm is suitable for a distributed and parallel implementation. For instance, in the next chapter we present one implementation that is able to generate an MPHF for a set of 14.336 billion 16-byte integer keys in 50 minutes using 14 commodity PCs, achieving an almost linear speedup.

Chapter 5

A Highly Scalable and Parallel Perfect Hashing Algorithm

In this chapter we present a parallel version of the *External Memory* (EM) algorithm presented in Chapter 4. The EM algorithm allows the generation of PHFs or MPHFs for sets in the order of billions of keys. For instance, if we consider an MPHF that requires 3.3 bits per key to be stored, for 1 billion URLs it would take approximately 400 megabytes. Considering now the time to generate an MPHF, taking the same set of 1.024 billion URLs as input, the algorithm outputs an MPHF in approximately 50 minutes using a commodity PC. It is well known that big search engines are nowadays indexing more than 20 billion URLs. Then, we are talking about approximately 8 gigabytes to store a single MPHF and approximately 1,000 minutes to construct an MPHF. Thus, two problems arise when the input key set size increases: (i) the amount of time to generate an MPHF becomes large for a single machine, and (ii) the storage space to describe an MPHF might be unsuitable for a single machine.

This motivated us to design parallel implementation of the EM algorithm, referred to as *Parallel External Memory* (PEM) algorithm from now on. The algorithm was designed for the *PRAM model* [67]. This model consists of a control unit, global memory, and an unbounded set of processors, each with its own private memory and executing identical instructions. In our implementation the network was considered the global memory and the processors share information by exchanging messages.

The PEM algorithm distributes both the construction and the description of the resulting functions. For instance, by using a 14-computer cluster the PEM algorithm generates a PHF or an MPHF for 1.024 billion URLs in approximately 4 minutes, achieving an almost linear speedup. Also, for 14.336 billion 16-byte random integers evenly distributed among the 14 participating machines the PEM algorithm outputs a PHF or an MPHF in approximately 50 minutes, resulting in a performance degradation of 20%. To the best of our knowledge there is no previous result in the perfect hashing literature that can be implemented in a parallel way to obtain better scalability and performance than the results presented hereinafter. A previous version of the PEM algorithm was presented in [11].

This chapter is organized as follows. In Section 5.1 we present the metrics used for evaluating the PEM algorithm. In Section 5.2 we describe the PEM algorithm in detail. In Section 5.3 we evaluate the PEM algorithm experimentally. Finally, we conclude in Section 5.4.

5.1 Metrics Used to Evaluate The PEM Algorithm

To evaluate the performance of the PEM algorithm we use two metrics: *speedup* and *scale-up*. By fixing the problem size, the speedup refers to how much a parallel algorithm is faster than a corresponding sequential algorithm, and is defined as:

Definition 22 The *speedup* $\mathcal{S}_p$ of a parallel algorithm using p processors is:

$$\mathcal{S}_p = \frac{T_1}{T_p}, \tag{5.1}$$

where T_1 is the execution time of the sequential algorithm and T_p is the execution time of the parallel algorithm with p processors.

Definition 23 The *efficiency* $\mathcal{E}_p$ of a parallel algorithm using p processors is:

$$\mathcal{E}_p = \frac{\mathcal{S}_p}{\mathcal{S}_{max}}, \tag{5.2}$$

where

$$\mathcal{S}_{max} = \frac{p}{1 + f \times (p-1)} \tag{5.3}$$

is the maximum speedup a parallel algorithm can achieve and $0 < f < 1$ corresponds to the sequential portion of the parallel algorithm (i.e., the fraction that cannot be improved using parallelism). This comes from the Amdahl's law [67].

By increasing the problem size proportionally to the number of processors p, the scale-up refers to the ability of solving a problem p times larger in the same amount of time the corresponding sequential algorithm would solve a problem $1/p$ times lower and is defined as:

Definition 24 The *scale-up* U_p of a parallel algorithm using p processors is:

$$U_p = \frac{T_p}{T_1}, \tag{5.4}$$

where T_1 is the execution time of the sequential algorithm to solve a problem of size X and T_p is the execution time of the parallel algorithm with p processors to solve a problem of size pX.

5.2 Parallel Algorithm

In this section we describe the *Parallel External Memory* (PEM) algorithm. As mentioned before, the main motivation for implementing a parallel version of the EM algorithm is scalability in terms of the size of the key set that has to be processed. In this case, we must assume that the keys to be processed will be distributed among several machines. Further, both the buckets and the construction of the hash functions for each bucket are also distributed among the participating machines. In this scenario, the partitioning and the searching steps present different requirements when compared to the sequential version, as we discuss next.

In Section 5.2.1 we discuss how to speedup the construction of a PHF or an MPHF by distributing the buckets (during the partitioning phase) and the construction of the functions f_i (remember that f_i is either a PHF or an MPHF) for each bucket (during the searching phase) among the participating machines. In Section 5.2.2 we present a version of the PEM algorithm where both the description and the evaluation of the resulting function is *centralized* in one machine, from now on referred to as *PEM-CE*. In Section 5.2.3 we present another version of the PEM algorithm where both the description and the evaluation of the resulting function are *distributed* among the participating machines, from now on referred to as *PEM-DE*.

5.2.1 Parallel Construction

In this section we present the steps that are common to both PEM-CE and PEM-DE algorithms. We employed two types of processes: manager and worker. This scheme is shown in Figure 5.1.

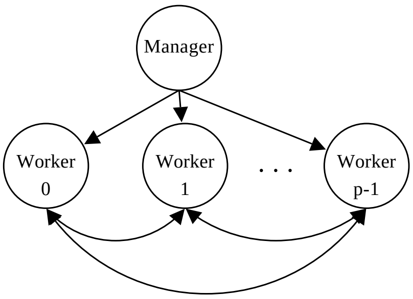


Figure 5.1: The manager/worker scheme.

The manager acts like the control unit of the PRAM model [67] and is responsible for assigning tasks to the workers, determining global values during the execution, and dumping the resulting PHFs or MPHFs received from the workers to disk. This last task is different for the PEM-CE and PEM-DE algorithms, as we will show later on.

The worker stores a partition of the key set, its buckets and the related PHF or MPHF of each bucket. Each worker sends and receives data from other workers whenever necessary. The workers are implemented as thread-based processes, where each thread is responsible for a task, allowing larger overlap between computation and communication (disk and network) in both steps of the algorithm.

Our major challenge in producing such a parallel version is that we do not know in advance which keys will be clustered together in the same bucket. Our strategy in this case is to migrate data whenever necessary. On the other hand, once we have the buckets, we are able to generate the functions.

The manager starts the processing by sending the overall assignment of buckets to workers before each worker starts processing its portion of the keys, so that each worker becomes aware of the worker to which keys (actually, fingerprints) must be sent. For that verification, the manager sends the following information: (i) the function $h' \in \mathcal{H}$ used to compute the fingerprints; (ii) the worker identifier i, where $0 \leq i < p$ and p is the number of workers; and (iii) the number of buckets per worker, which is given by $B_{pw} = \lceil N_b/p \rceil$ (recall that N_b is the number of buckets). Therefore, each worker i is responsible for the buckets in the range $[iB_{pw}, (i+1)B_{pw} - 1]$.

Each worker then starts reading a key $k \in S$, applies the received hash function h' and verifies whether it belongs to another worker. For that each worker i computes $w = h_0(k)/B_{pw}$ and checks if $w \neq i$ (recall that $h_0(k)$ corresponds to the b most significant bits of $h'(k)$.) If it is the case, it sends the corresponding fingerprint to the worker w, otherwise, it stores the fingerprint locally for further processing.

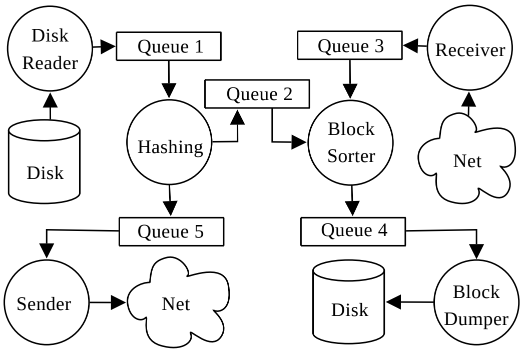


Figure 5.2: The partitioning step in the worker.

Figure 5.2 illustrates the partitioning step in each worker. The partitioning step of the sequential algorithm presented in Figure 4.2 is divided into four major tasks: data reading (line 3), hashing (line 4), block sorting (line 5), and block dumping (line 6).

As depicted in Figure 5.2, the worker is divided into the following six threads:

1. *Disk Reader*: it reads the keys from the worker's portion of the set S and puts them in Queue 1. When there are no more keys to be read, then an end of file marker is put in Queue 1.

2. *Hashing*: it gets the keys from Queue 1 and generates the fingerprints for the keys, as mentioned in Section 4.1.1. This thread then checks whether the key being currently analyzed is assigned to another worker. If it is, its fingerprint is passed to the Sender thread through Queue 5, otherwise its fingerprint is placed in Queue 2. When there are no more keys to be processed in Queue 1, then an end of file marker is put in both Queue 2 and 5.

3. *Sender*: it sends a fingerprint taken from Queue 5 to the worker that is responsible for it. When there are no more fingerprints in Queue 5, then an end of file marker is sent to all other workers.

4. *Receiver*: it receives fingerprints sent from other workers through the net, and puts them in Queue 3. It finishes its work when an end of file marker is received from all other workers.

5. *Block Sorter*: it takes fingerprints from Queues 2 and 3 until a buffer of size $\mu/2$ bytes is completely full (recall that μ is the amount of internal memory available),

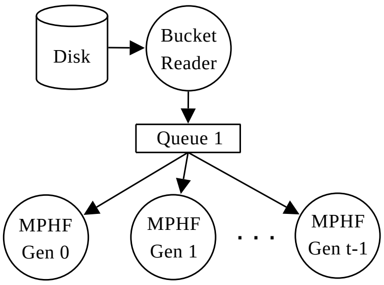


Figure 5.3: The searching step in the worker.

organizes them into buckets, and puts them in Queue 4. The process is repeated until an end of file marker is obtained from both Queues 2 and 3. In this case, it also places an end of file marker in Queue 4.

6. *Block Dumper*: it takes the buckets from Queue 4 and writes them to disk, for further processing by the searching step. It finishes when an end of file marker is taken from Queue 4.

After each worker finishes the partitioning step, it sends the size of each bucket to the manager, which then calculates the offset array. This does not depend on the searching step, so the manager may compute the offset array whereas the workers are performing the searching step.

Figure 5.3 illustrates the searching step in each worker. It consists of generating the functions f_i for each bucket i (remember that f_i is either a PHF or an MPHF.) The searching step of the sequential algorithm of Figure 4.4 is divided into three tasks: bucket reading (line 5), PHF or MPHF construction (lines 6 and 7), and PHF or MPHF dumping (line 8). Notice that, in this step, there is no need for communication between workers, since the generation of function f_i for each bucket does not depend on keys that are in other buckets.

Again, the worker is divided into threads of execution, each thread being responsible for a task. Following Figure 5.3, the worker is divided into the following two threads:

1. *Bucket Reader*: it reads the buckets from disk, and puts them in Queue 1. When there are no more buckets to be read, then an end of file marker is put in Queue 1.

2. *MPHF Gen*: it gets buckets from Queue 1 and generates the functions for them until no more bucket remains. It can be instantiated t times, where t can be thought of as the number of processors of the machine.

5.2.2 Centralized Evaluation of the Resulting Functions

In this section we present the PEM-CE algorithm, where both the description and the evaluation of the resulting PHF or MPHF is centralized in a single machine (the one running the manager process).

After each worker finishes the partitioning step, it sends the size of each bucket to the manager, which then calculates the offset array. This does not depend on the searching step, so the manager may compute the offset array whereas the workers are performing the searching step. After each worker finishes the construction of the PHFs or MPHFs of their buckets, it sends them to the manager, that will then write sequentially the final PHF or MPHF to disk, and the algorithm resumes.

The task of writing the final PHF or MPHF to disk corresponds to the sequential part of the algorithm and represents approximately 0.5% of the execution time. Thus, there is a fraction of 99.5% of the execution time from which we can exploit parallelism. That is why the PEM-CE algorithm can be considered an embarrassingly parallel algorithm.

The evaluation of the resulting functions is done in the same way as it is done in the sequential algorithm presented in Section 4.1 (see Definition 21).

5.2.3 Parallel Evaluation of the Resulting Functions

In this section we present the PEM-DE algorithm, where both the description and the evaluation of the resulting function are distributed and stored locally in each worker. The PEM-DE algorithm calculates a *localoffset* array in each worker, in the same way as it is done in the searching step of the sequential algorithm shown in Figure 4.4 (see line 7). At the end of the partitioning step, each worker sends the number of keys assigned to it to the manager, which calculates a *globaloffset*, whereas the workers are performing the searching step.

To evaluate a key k using the resulting PHF or MPHF function f, the manager first discovers the worker w that generated the PHF or MPHF for the bucket in which k is (recall that this is done by calculating $w = h_0(k)/B_{pw}$). Then, the key k (actually, its fingerprint) is sent to the worker w, which calculates locally a partial result

$$f_{partial}(k) = f_i(k) + \mathit{localoffset}[i],$$

where $i = h_0(k) \bmod B_{pw}$ is the local bucket address where k belongs and $\mathit{localoffset}[i]$ gives the total number of keys before bucket i. Once this partial result is calculated, it is sent back to the manager, which calculates the final result

$$f(k) = f_{partial}(k) + \mathit{globaloffset}[w],$$

where $\mathit{globaloffset}[w]$ has p entries and gives the total number of keys handled by the workers before worker w.

The downside of this is that the evaluation of a single key is harmed, due to the communication overhead between the manager and the workers. However, if the system is being fed by a key stream, the average performance will improve because p keys can be evaluated in parallel by p workers. This will indeed happen because the keys are uniformly placed in the buckets by using a hash function, which will balance the key stream among the p workers. The experimental results in Section 5.3 confirm this fact.

Other advantage of the PEM-DE algorithm is that the workers do not need to send the PHFs or MPHFs generated locally for the buckets they are responsible for to the manager. Instead, they are written in parallel by the workers. Therefore, in this case, the fraction of parallelism we can potentially exploit corresponds to 100% of the execution time.

Therefore, as shown in Section 5.3, the PEM-DE algorithm provides a slightly better construction time than the PEM-CE algorithm. But the main advantage of the PEM-DE algorithm is that it distributes the resulting PHF or MPHF among several machines. When the number n of keys in the key set S grows, the size of the resulting PHF or MPHF also grows linearly with n. For very large n, it may not be possible to represent the resulting function in just one machine, whereas the PEM-DE algorithm addresses this by distributing uniformly the resulting function.

5.2.4 Implementation Decisions

In this section we present and discuss some implementation decisions that aim to reduce the overhead of the parallel algorithms we just described.

A very first decision is to exploit multiprogramming in the worker, motivated not only by the characteristics of the execution platform, but also by the complementary profiles of the steps, which are either CPU or I/O-intensive. As a result, we are able to maximize the overlap between computation and communication, represented by disk and network traffic.

Further, in order to reduce the overhead due to context changes we grouped steps (described in Section 5.2.1) into fewer threads, as detailed next. This strategy speeds up the execution time, even on a single core machine, which is our case.

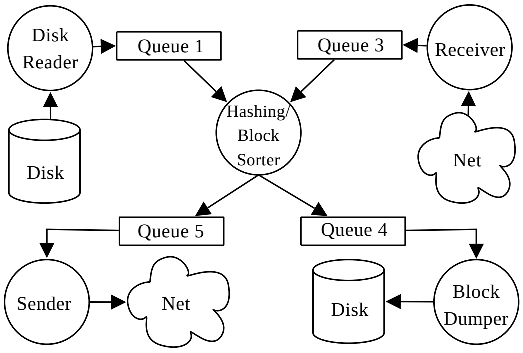


Figure 5.4: The actual partitioning step used in the experiments.

In the partitioning step, the *Hashing* and *Block Sorter* threads were grouped together into a single thread, as shown in Figure 5.4. Notice that these two steps are the most CPU-intensive and the merge would prevent them to contend for the CPU. As a result, one thread is almost always keeping the CPU busy, while the remaining threads are usually waiting for system calls to resume (*Disk Reader* reading data from disk, *Net Reader* receiving messages from the net, and *Block Dumper* writing buckets back to disk whenever necessary).

In the searching step, the structure replicates the step-based division presented, but instantiating just one MPHF Gen thread (i.e., $t = 1$), as shown in Figure 5.5.

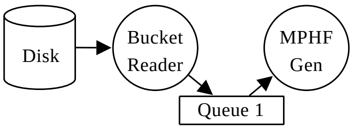


Figure 5.5: The actual searching step used in the experiments.

We also coalesced messages for both reducing the number of system calls associated with exchange messages and better exploiting the available bandwidth. That is, we group the fingerprints that were going to be sent from one to another worker in buffers of a fixed size.

5.3 Experimental Results

The purpose of this section is to evaluate the performance of both the PEM-CE and PEM-DE algorithms in terms of speedup and scale-up (see Definitions 22 and 24), considering the impact of the key size in both metrics. We also verify whether the load is balanced among the workers. To compute the metrics we use the time to construct a PHF or an MPHF in the parallel algorithms. We remark that we simplified a lot our experimental evaluation. For instance, we did not analyze the influence of some factors (e.g., message coalescing) in the speedup and scale-up. Our aim in this section is to illustrate that the two versions of the PEM algorithm are embarrassingly parallel but a more thorough experimental evaluation is left to be done as a future work.

The experiments were run in a cluster with 14 equal single core machines, each one with 2.13 gigahertz, 64-bit architecture, running the Linux operating system version 2.6, and 2 gigabytes of main memory.

For the experiments we used three collections: (i) a set of URLs collected from the web, (ii) a set of randomly generated 16-byte integers, and (iii) a set of randomly generated 8-byte integers. The collections are presented in Table 5.1. The main reason to choose these three different collections is to evaluate the impact of the key size on the results.

Collection	Average key size	n (billions)
URLs	64	1.024
Random Integers	16	1.024
	8	1.024

Table 5.1: Collections used for the experiments.

In Section 5.3.1 we discuss the impact of key size on speedup and scale-up. In Section 5.3.2 we study the communication overhead. In Section 5.3.3 we discuss the load balance among workers. In Section 5.3.4 we discuss the parallel evaluation of an MPHF when the function is being fed by a key stream. The same results were obtained for a PHF and therefore were not presented.

5.3.1 Key Size Impact

In this section we evaluate the impact of the key size and how it changes as we increase the number of processors. We use both speedup and scale-up as metrics for performing such evaluation.

In order to compute the speedup we need the execution time of the sequential EM algorithm. Table 5.2 shows how much time the EM algorithm requires to build an MPHF for 1.024 billion keys taken from each collection shown in Table 5.1.

n (billion)	Collection	time (min.)
1.024	64-byte URLs	50.02
	16-byte integers	39.35
	8-byte integers	34.58

Table 5.2: Time in minutes of the sequential algorithm (EM) to construct an MPHF for 1.024 billion keys.

We start by evaluating the speedup of the parallel algorithm and perform three sets of experiments, using the three collections presented in Table 5.1 and varying the number of machines from 1 to 14.

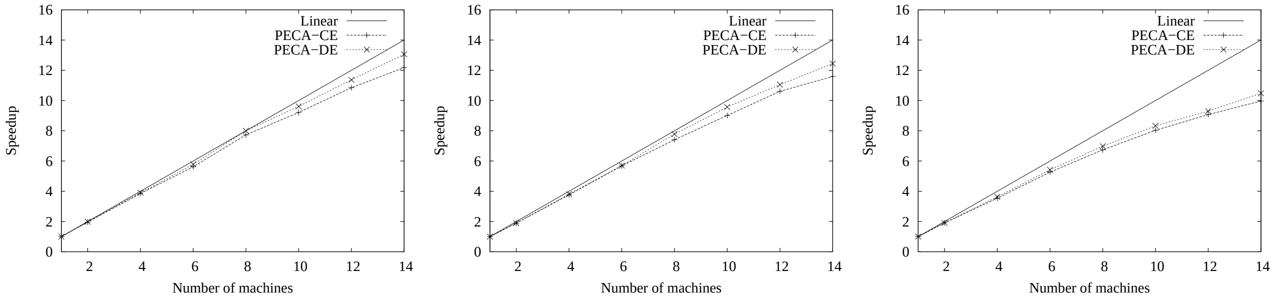
Table 5.3 presents the maximum speedup ($\mathcal{S}_{max}$), the speedup $\mathcal{S}_p$ and the efficiency $\mathcal{E}_p$ for both the PEM-CE and PEM-DE algorithms for each collection. In almost all cases, the speedup was very good, achieving an efficiency of up to 93% using 14 machines, confirming the expectations of that not only there is a parallelism opportunity to be exploited, but also it is significative enough that allows good efficiencies even for relatively large configurations. The comparison between PEM-CE and PEM-DE also shows that the strategy employed in PEM-DE was effective.

It is remarkable that the key size impacts the observed speedups, since the efficiency for the 64-byte URLs is greater than 90% for all configurations evaluated, but for 16-byte and 8-byte random integers it is greater than or equal to 90% only for $p \geq 12$ and $p \geq 6$, respectively. This happens because when we decrease the key size, the amount of computation decreases proportionally in the partitioning step, but the amount of communication remains constant since the γ-bit fingerprints will continue with the same size $\gamma = 96$ bits (or 12 bytes.) The size γ of a fingerprint depends on the number of keys n, but does not depend on the key size [15]. Therefore, the smaller is the key size, the smaller is the value of p to fully exploit the available parallelism, resulting in eventual performance degradation. A graphical view of the speedups can also be seen in Figure 5.6.

We performed similar sets of experiments for evaluating the scale-up and the results are presented in Table 5.5 and Figure 5.7, where we may confirm the good scalability of the algorithm, which allows just 17% of degradation when using 14 machines to solve a problem 14 times larger. These results show that not only the algorithm proposed is

p	$\mathcal{S}_{max}$		64-byte URLs				16-byte random integers				8-byte random integers			
			PEM-CE		PEM-DE		PEM-CE		PEM-DE		PEM-CE		PEM-DE	
	PEM-CE	PEM-DE	$\mathcal{S}_p$	$\mathcal{E}_p$	$\mathcal{S}_p$	$\mathcal{E}_p$	$\mathcal{S}_p$	$\mathcal{E}_p$	$\mathcal{S}_p$	$\mathcal{E}_p$	$\mathcal{S}_p$	$\mathcal{E}_p$	$\mathcal{S}_p$	$\mathcal{E}_p$
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	1.99	2.00	1.96	0.98	1.99	1.00	1.89	0.95	1.90	0.95	1.91	0.96	1.91	0.96
4	3.94	4.00	3.85	0.98	3.90	0.98	3.76	0.95	3.81	0.95	3.54	0.90	3.63	0.91
6	5.85	6.00	5.62	0.96	5.78	0.96	5.68	0.97	5.70	0.95	5.27	0.90	5.42	0.90
8	7.73	8.00	7.73	1.00	8.00	1.00	7.41	0.96	7.78	0.97	6.74	0.87	6.98	0.87
10	9.57	10.00	9.21	0.96	9.61	0.96	9.01	0.94	9.57	0.96	8.03	0.84	8.33	0.83
12	11.37	12.00	10.85	0.95	11.37	0.95	10.61	0.93	11.05	0.92	9.07	0.80	9.30	0.78
14	13.15	14.00	12.18	0.93	13.06	0.93	11.59	0.88	12.44	0.89	9.97	0.76	10.48	0.75

Table 5.3: Speedup obtained with a confidence level of 95% for both the PEM-CE and PEM-DE algorithms considering 1.024 billion keys (73,142,857 keys in each machine).



(a) 64-byte URLs (b) 16-byte random integers (c) 8-byte random integers

Figure 5.6: Speedup obtained with a confidence level of 95% for both the PEM-CE and PEM-DE algorithms considering 1.024 billion keys (73,142,857 keys in each machine).

efficient, but also is very effective dealing with larger datasets. For instance, in Table 5.4 it is shown that the performance degradation is up to 20% even for 14.336 billion keys evenly distributed among 14 machines. Again, the key size has a definite impact on the performance.

5.3.2 Communication Overhead

We now analyze the communication overhead. There is a significant overhead associated with message traffic among workers in the net. Since the hash function h_0 is a linear hash function [15] that behaves closely to a fully random hash function, the chance of a given key in the key set S belonging to a given bucket is close to $\frac{1}{N_b}$. Since each worker has $\frac{N_b}{p}$ buckets, the chance that a key it reads belongs to another worker is close to $\frac{p-1}{p}$. Since

n	Random integer	Construction time (min)		
(billions)	collections	EM	PEM-DE	U_p
14.336	16-byte	41.17	49.5	1.20
	8-byte	34.58	58.00	1.68

Table 5.4: Scale-up obtained with a confidence level of 95% for the PEM-DE algorithm considering 14.336 billion keys (1.024 billion keys in each machine).

p	64-byte URLs				16-byte random integers				8-byte random integers			
	PEM-CE		PEM-DE		PEM-CE		PEM-DE		PEM-CE		PEM-DE	
	t (min)	U_p	t (min)	U_p	t (min)	U_p	t (min)	U_p	t (min)	U_p	t (min)	U_p
1	3.71	1.00	3.68	1.00	2.68	1.00	2.70	1.00	2.00	1.00	2.00	1.00
2	3.76	1.01	3.71	1.01	2.74	1.02	2.69	1.00	2.16	1.08	2.11	1.06
4	3.84	1.03	3.77	1.03	2.77	1.03	2.71	1.00	2.44	1.22	2.35	1.17
6	3.91	1.05	3.81	1.04	2.82	1.05	2.73	1.01	2.68	1.34	2.58	1.29
8	3.96	1.07	3.82	1.04	2.94	1.10	2.76	1.02	3.04	1.52	2.82	1.41
10	4.02	1.08	3.83	1.04	3.10	1.15	2.86	1.06	3.25	1.62	3.10	1.55
12	4.02	1.08	3.84	1.05	3.23	1.20	3.02	1.12	3.48	1.74	3.29	1.64
14	4.11	1.11	3.85	1.05	3.40	1.27	3.16	1.17	3.47	1.73	3.30	1.65

Table 5.5: Scale-up obtained with a confidence level of 95% for both the PEM-CE and PEM-DE algorithms considering 1.024 billion keys (73,142,857 keys in each machine).

each worker has to read $\frac{n}{p}$ keys from disk, it will send through the net approximately

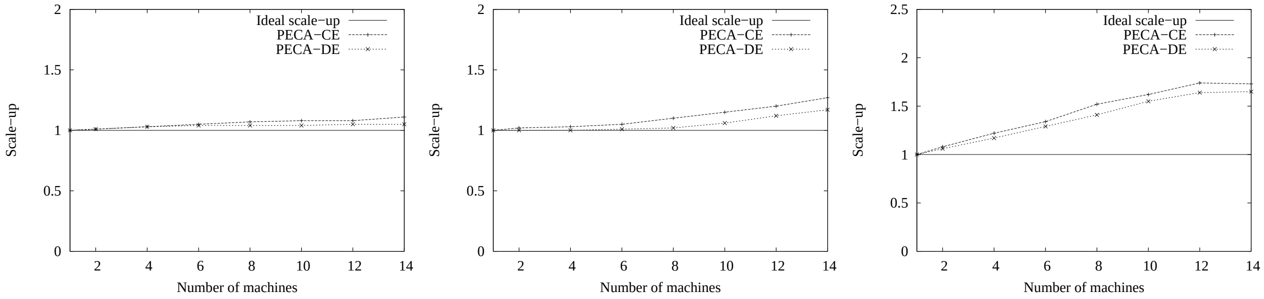
$$\frac{n(p-1)}{p^2}.$$

Thus, the total traffic τ of fingerprints through the net is approximately

$$\tau \approx \frac{n(p-1)}{p}. \tag{5.5}$$

Table 5.6 shows the minimum and maximum amount of keys sent to the net by a worker. It also shows the expected amount computed by using Eq. (5.5). As it shows, the empirical measurements are really close to the expected value.

That results in a relevant overhead due to communication among the workers, and as the number of workers increases, the speedup can be penalized if the network bandwidth is not enough for the traffic. In our 1 gigabit ethernet network this was not a problem for at most 14 workers.



(a) 64-byte URLs (b) 16-byte random integers (c) 8-byte random integers

Figure 5.7: Scale-up obtained with a confidence level of 95% for both the PEM-CE and PEM-DE algorithms considering 1.024 billion keys (73,142,857 keys in each machine).

p	Keys sent by a worker to the net		
	Max (%)	Min (%)	τ (%)
2	50.005	49.996	50.000
4	75.008	74.994	75.000
6	83.339	83.327	83.333
8	87.506	87.492	87.500
10	90.009	89.991	90.000
12	91.673	91.657	91.667
14	92.864	92.849	92.857

Table 5.6: Worst, best and expected percentage of keys sent by a worker to the net.

5.3.3 Load Balancing

In this section we quantify the load imbalance and correlate it with the results. An important issue is how much the load is balanced among the workers. The load depends on the following parameters: (i) the number of keys each worker reads from disk in the partitioning step; (ii) the number of buckets each worker is responsible for; (iii) the number of keys in each bucket.

The first two parameters are fixed by construction and are evenly distributed among the workers. The only parameter that could present some variation in each execution is the last one. However, as we use the hash function h_0 to split the key set into buckets, it was shown in [15] that each key goes to a given bucket with probability close to $1/N_b$ and therefore the distribution of the bucket sizes follows a binomial distribution with average n_p/B_{pw}, where $n_p = \sum_{i=0}^{B_{pw}-1} |B_i|$ is the number of keys each worker has stored in the buckets it is responsible for and B_{pw} is the number of buckets per worker.

It is also shown in [15] that the largest bucket is within a factor $O(\log \log n_p)$ of the average bucket size. Therefore, n_p has a very small variation from worker to worker, which makes the load balanced among the p machines. Table 5.7 presents experimental results confirming this, as the difference between the execution time of the fastest worker (t_{fw}) and the slowest worker (t_{sw}) was less than or equal to 0.1 minutes.

p	PEM-CE			PEM-DE		
	t_{fw}	t_{sw}	$t_{sw} - t_{fw}$	t_{fw}	t_{sw}	$t_{sw} - t_{wb}$
2	25.27	25.37	0.10	25.01	25.06	0.05
4	12.94	13.04	0.10	12.78	12.86	0.09
6	8.58	8.69	0.11	8.53	8.65	0.11
8	6.19	6.26	0.07	6.18	6.25	0.07
10	5.14	5.22	0.08	5.11	5.20	0.09
12	4.31	4.39	0.08	4.32	4.40	0.07
14	3.81	3.88	0.07	3.76	3.84	0.08

Table 5.7: Fastest worker time (t_{fw}), slowest worker time (t_{sw}), and difference between t_{sw} and t_{fw} to show the load balancing among the workers for 1.024 billion 64-byte URLs distributed in p machines. The times are in minutes.

5.3.4 Parallel Evaluation

In this section we show that the parallel evaluation of an MPHF is worth when compared to the ones generated by both the sequential and PEM-CE algorithms. These results assume that the parallel function is being fed by a key stream, instead of one key at a time.

Table 5.8 shows the times that both the EM algorithm and PEM-DE algorithm needs to evaluate one billion keys taken at random. As expected, the parallel evaluation was faster because p keys of the key stream can be evaluated in parallel by p participating machines. Here we also used the message coalescing technique. We remark that a more thorough evaluation must be done to identify the impact of the message coalescing technique in the parallel evaluation time.

Collection	Evaluation time (min)	
	EM	PEM-DE
64-byte URLs	33.11	21.68
16-byte random integers	24.54	11.47
8-byte random integers	18.2	10.1

Table 5.8: Evaluation time in minutes for both the sequential algorithm EM and the parallel algorithm PEM-DE algorithm, considering 1 billion keys.

5.4 Conclusions

In this chapter we have presented a parallel implementation of the External Memory (EM) perfect hashing algorithm presented in Chapter 4. We have designed two versions. The PEM-CE algorithm distributes the construction of the resulting PHFs or MPHFs among p machines and centralize the evaluation and description of the resulting functions in a single machine, as in the sequential case. Then the goal in this version is to speedup the construction of the PHFs or MPHFs by exploiting the high degree of parallelism of the EM algorithm. The PEM-DE algorithm distributes both the construction and the evaluation of the resulting functions. In this version the goal is to allow the descriptions of the resulting functions be uniformly distributed among the participating machines.

We have evaluated both the PEM-CE and PEM-DE algorithms using speedup and scale-up as metrics. Both versions presented an almost linear speedup, achieving an efficiency larger than 90% by using 14-computer cluster and keys of average size larger than or equal to 16 bytes. For smaller keys, e.g. 8-byte integers, we have shown that the existent parallelism between computation and communication is captured with 90% of efficiency by using a smaller number of machines (e.g, $p = 6$). This was as expected, because the smaller is the key the smaller is the amount of computation, but the amount of communication remains constant for a given number n of keys, penalizing the speedup.

We have also shown that both the PEM-CE and PEM-DE algorithms scale really well for larger keys. Smaller keys also impose restrictions on the scalability due to the smaller degree of overlap between computation and communication aforementioned. To illustrate the scalability, the time to generate an MPHF for 14.336 billion 16-byte random integers using a 14-computer cluster with 1.024 billion 16-byte random integers in each machine is just a factor of 1.2 more than the time spent by the sequential algorithm when applied to 1.024 billion keys.

Chapter 6

MPHFs and Random Graphs With Cycles

In this chapter we describe two algorithms for constructing minimal perfect hash functions based on random graphs with cycles. A previous version of the first algorithm was presented by Botelho, Kohayakawa and Ziviani in [12]. For this reason it will be referred to as BKZ algorithm, which is an acronym for its author names. The second algorithm uses the same techniques used in the BKZ algorithm to speedup the execution time of the RAM algorithm that works on random acyclic bipartite graphs, which is presented in Chapter 2.

The reason to use random graphs with cycles comes from the fact that the functions are generated faster and are more compact than the ones generated based on acyclic random graphs. This is because both the generation time and the space usage of the resulting functions depend on the number of vertices in the random graphs and the acyclic ones are more sparse. That is, the ratio between the number of vertices and number of edges must be larger than two.

This chapter is organized as follows. In Section 6.1 we present the BKZ algorithm and compare the BKZ algorithm with an algorithm that was used as departure point in its design. In Section 6.2 we show how to speedup the RAM algorithm with the techniques used in the design of the BKZ algorithm and compare the optimized version of the RAM algorithm with the version of the RAM algorithm presented in Chapter 2. Finally, in Section 6.3 we conclude this chapter.

6.1 The BKZ Algorithm

The BKZ algorithm shares several features with the one due to Czech, Havas and Majewski [25], from now on referred to as CHM algorithm. In particular, the BKZ algorithm is also based on the generation of random graphs $G = (V, E)$, where E is in one-to-one correspondence with the key set S for which we want to generate the hash function. The two main differences between the BKZ algorithm and the CHM algorithm are as follows: (i) the BKZ algorithm generates random graphs $G = (V, E)$ with $|V| = cn$ and $|E| = |S| = n$, where $c = 1.15$ ($|V| = 1.15n$), and hence G contains cycles with high probability, whereas the CHM algorithm generates *acyclic* random graphs $G = (V, E)$ with $|V| = cn$ and $|E| = |S| = n$, with a greater number of vertices: $c = 2.09$ ($|V| = 2.09n$); (ii) The CHM algorithm generates order preserving minimal perfect hash functions whereas the BKZ algorithm does not preserve order. Thus, the BKZ algorithm improves the space requirement at the expense of generating functions that are not order preserving.

As the CHM algorithm, the BKZ algorithm produces an MPHF in $O(n)$ expected time for a set of n keys. The MPHF description requires $1.15n$ computer words, and evaluating it requires two accesses to an array of $1.15n$ integers. We further derive a heuristic that improves the space requirement from $1.15n$ words down to $0.93n$ words. The BKZ algorithm is very practical. To generate a minimal perfect hash function for a collection of 100 million universe resource locations (URLs), each 63 bytes long on average, the BKZ algorithm running on a commodity PC takes 811 seconds on average. In Section 6.1.1 we present the CHM algorithm. In Section 6.1.2 we present the design of the BKZ algorithm. In Section 6.1.3 we compare the BKZ algorithm with the CHM algorithm experimentally.

6.1.1 The CHM algorithm

In this section we briefly present the CHM algorithm. Consider a problem known as the *perfect assignment problem*: For a given undirected graph $G = (V, E)$, where $|V| = cn$ and $|E| = n$, find a function $g{:}V \to \{0, 1, \ldots, |V| - 1\}$ such that the function $\mathit{mphf} : E \to \{0, 1, \ldots, n - 1\}$, defined as

$$\mathit{mphf}(e) = (g(a) + g(b)) \bmod n \tag{6.1}$$

is a bijection, where $e = \{a, b\}$. This means that we are looking for an assignment of values to vertices so that for each edge the sum of values associated with endpoints taken modulo the number of edges is a unique integer in the range $[0, n - 1]$.

Many methods for generating MPHFs use a *mapping*, *ordering* and *searching* (MOS) approach, a description coined by Fox, Chen and Heath [38]. In the MOS approach, the construction of a minimal perfect hash function is accomplished in three steps. First, the mapping step transforms the key set from the original universe to a new universe. Second, the ordering step places the keys in a sequential order that determines the order in which hash values are assigned to keys. Third, the searching step attempts to assign hash values to the keys. The CHM algorithm uses the MOS approach as well as our algorithm presented in Section 6.1.

The ordering and searching steps of the MOS approach are a very simple way of solving the perfect assignment problem. Czech, Havas and Majewski [25] showed that the perfect assignment problem can be solved in optimal time if G is acyclic. To generate an acyclic random graph, the method assumes that two uniform hash functions h_1 and h_2 are available for free. The functions h_1 and h_2 are constructed as follows. We impose some upper bound L on the lengths of the keys in S. To define h_j (j = 1,2), we generate an $L \times |\Sigma|$ table of random integers table$_j$. For a key $x \in S$ of length $|x| \leq L$ and $j \in \{1, 2\}$, we let

$$h_j(x) = \left(\sum_{i=0}^{|x|-1} \text{table}_j[i, x[i]] \right) \bmod m. \tag{6.2}$$

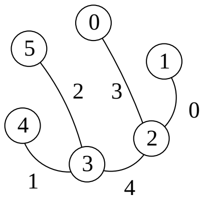
Thus, set S has a corresponding graph $G = G(h_1, h_2)$, with $V = \{0, 1, \ldots, m-1\}$, where $|V| = m$, and $E = \{\{h_1(x), h_2(x)\} : x \in S\}$. In order to guarantee acyclicity the algorithm repeatedly selects h_1 and h_2 until the corresponding graph is acyclic. For the solution to be useful we must have $|S| = n$ and $m = cn$, for some constant c, such that acyclic graphs dominate the space of all random graphs. Havas et al. [44] proved that if $m = cn$ holds with $c > 2$ the probability that G is acyclic is

$$Pr_a = e^{1/c} \sqrt{\frac{c-2}{c}}. \tag{6.3}$$

For $c = 2.09$ the probability of a random graph being acyclic is $Pr_a > \frac{1}{3}$. Consequently, for such c, the expected number of iterations to obtain an acyclic graph is lower than 3 and the g function needs $2.09n$ integer numbers to be stored, since its domain is the set V of size $m = cn$.

Given an acyclic graph G, for the ordering step we associate with each edge an unique number $mphf(e) \in [0, n-1]$ in the order of the keys of S to obtain an order preserving function. Figure 6.1 illustrates the perfect assignment problem for an acyclic graph with six vertices and with the five function values assigned to the edges.

The searching step starts from the weighted graph G obtained in the ordering step. For each connected component of G choose a vertex v and set $g(v)$ to 0. For example, suppose that vertex 0 in Figure 6.1 is chosen and the assignment $g(0) = 0$ is made. Traverse the graph using a depth-first or a breadth-first search algorithm, beginning with vertex v. If vertex b is reached from vertex a and the value associated with the edge $e = \{a, b\}$ is $mphf(e)$, set $g(b)$ to $(mphf(e) - g(a)) \bmod n$. In Figure 6.1, following the adjacent list of vertex 0, $g(2)$ is set to 3. Next, following the adjacent list of vertex 2, $g(1)$ is set to 2 and $g(3)$ is set to 1, and so on.



v	$g(v)$
0	0
1	2
2	3
3	1
4	0
5	1

Figure 6.1: Perfect assignment problem for a graph with six vertices and five edges.

Now we show why G must be acyclic. If the graph G was not acyclic, the assignment process might trace around a cycle and insist on reassigning some already-processed vertex with a different g value than the one that has already been assigned to it. For example, let us suppose that in Figure 6.1 the edge $\{3, 4\}$ has been replaced by the edge $\{0, 1\}$. In this case, two different values are set to $g(0)$. Following the adjacent list of vertex 1, $g(0)$ is set to 4. But $g(0)$ was set to 0 before.

6.1.2 Design of The BKZ Algorithm

We now present how our MPHF, which has the same form of the one generated by the CHM algorithm, will be constructed. We make use of two uniform hash functions h_1 and $h_2 : U \to V$, where $V = [0, m-1]$ for some suitably chosen integer $m = cn$, where $n = |S|$ (see Eq. (6.2)). We build a random graph $G = G(h_1, h_2)$ on S, whose edge set is $\big\{\{h_1(x), h_2(x)\} : x \in S\big\}$. There is an edge in G for each key in the key set S. Note that in our case the random graph G may have cycles.

In what follows, we shall be interested in the *2-core* of the random graph G, that is, the maximal subgraph of G with minimal degree at least 2 (see, e.g., [8, 49]). Because of its importance in our context, we call the 2-core the *critical* subgraph of G and denote it by G_{crit}. The vertices and edges in G_{crit} are said to be *critical*. We let $V_{\text{crit}} = V(G_{\text{crit}})$ and $E_{\text{crit}} = E(G_{\text{crit}})$. Moreover, we let $V_{\text{ncrit}} = V - V_{\text{crit}}$ be the set of *non-critical* vertices in G. We also let $V_{\text{scrit}} \subseteq V_{\text{crit}}$ be the set of all critical vertices that have at least one non-critical vertex as a neighbor. Let $E_{\text{ncrit}} = E(G) - E_{\text{crit}}$ be the set of *non-critical* edges in G. Finally, we let $G_{\text{ncrit}} = (V_{\text{ncrit}} \cup V_{\text{scrit}}, E_{\text{ncrit}})$ be the *non-critical* subgraph of G. The non-critical subgraph G_{ncrit} corresponds to the "acyclic part" of G. We have $G = G_{\text{crit}} \cup G_{\text{ncrit}}$.

We then construct a suitable labelling $g : V \to \mathbb{Z}$ of the vertices of G: we choose $g(v)$ for each $v \in V(G)$ in such a way that $mphf(x) = g(h_1(x)) + g(h_2(x))$ $(x \in S)$ is an MPHF for S. We will see later on that this labelling g can be found in linear time if the number of edges in G_{crit} is at most $\frac{1}{2}|E(G)|$.

Figure 6.2 presents a pseudo code for the algorithm. The procedure GenerateMPHF (S, g) receives as input the key set S and produces the labelling g. The method uses a mapping, ordering and searching approach. We now describe each step.

```
procedure GenerateMPHF ($S$, $g$)
    Mapping ($S$, $G$);
    Ordering ($G$, $G_{\text{crit}}$, $G_{\text{ncrit}}$);
    Searching ($G$, $G_{\text{crit}}$, $G_{\text{ncrit}}$, $g$);
```

Figure 6.2: Main steps of the algorithm for constructing a minimal perfect hash function.

Mapping Step

The procedure Mapping (S, G) receives as input the key set S and generates the random graph $G = G(h_1, h_2)$, by generating two auxiliary functions h_1, $h_2 : U \to [0, m-1]$ (see Eq. (6.2)). This is done by filling each table$_j$ for $j \in \{1, 2\}$ with random integer numbers.

The random graph $G = G(h_1, h_2)$ has vertex set $V = [0, m-1]$ and edge set $\big\{\{h_1(x), h_2(x)\} : x \in S\big\}$. We need G to be simple, i.e., G should have neither loops nor multiple edges. A loop occurs when $h_1(x) = h_2(x)$ for some $x \in S$. We solve this in an ad hoc manner: we simply let $h_2(x) = (2h_1(x) + 1) \bmod m$ in this case. If we still find a loop after this, we generate another pair (h_1, h_2). When a multiple edge occurs we abort and generate a new pair (h_1, h_2).

Analysis of the Mapping Step

We start by discussing some facts on random graphs. Let $G = (V, E)$ with $|V| = m$ and $|E| = n$ be a random graph in the uniform model $\mathcal{G}(m, n)$, the model in which all the $\binom{\binom{m}{2}}{n}$ graphs on V with n edges are equiprobable. The study of $\mathcal{G}(m, n)$ goes back to the classical work of Erdős and Rényi [33, 34, 35] (for a modern treatment, see [8, 49]). Let $d = 2n/m$ be the average degree of G. It is well known that, if $d > 1$, or, equivalently, if $c < 2$ (recall that we have $m = cn$), then, almost every G contains[1] a "giant" component of order $(1 + o(1))bm$, where $b = 1 - T/d$, and $0 < T < 1$ is the unique solution to the equation $Te^{-T} = de^{-d}$. Moreover, all the other components of G have $O(\log m)$ vertices. Also, the number of vertices in the 2-core of G (the maximal subgraph of G with minimal degree at least 2) that do not belong to the giant component is $o(m)$ almost surely.

Pittel and Wormald [65] present detailed results for the 2-core of the giant component of the random graph G. Since table$_j$ ($j \in \{1, 2\}$) are random, $G = G(h_1, h_2)$ is a random graph. In what follows, we work under the hypothesis that $G = G(h_1, h_2)$ is drawn from $\mathcal{G}(m, n)$. Thus, following [65], the number of vertices of G_{crit} is

$$|V(G_{\text{crit}})| = (1 + o(1))(1 - T)bm \tag{6.4}$$

almost surely. Moreover, the number of edges in this 2-core is

$$|E(G_{\text{crit}})| = (1 + o(1))\Big((1 - T)b + b(d + T - 2)/2\Big)m \tag{6.5}$$

almost surely. Let $d_{\text{crit}} = 2|E(G_{\text{crit}})|/|V(G_{\text{crit}})|$ be the average degree of G_{crit}. We are interested in the case in which d_{crit} is a constant.

As mentioned before, for us to find the labelling $g : V \to \mathbb{Z}$ of the vertices of $G = G(h_1, h_2)$ in linear time, we require that $|E(G_{\text{crit}})| \leq \frac{1}{2}|E(G)| = \frac{1}{2}|S| = n/2$. The crucial step now is to determine the value of c (in $m = cn$) to obtain a random graph $G = G_{\text{crit}} \cup G_{\text{ncrit}}$ with $|E(G_{\text{crit}})| \leq \frac{1}{2}|E(G)|$.

Table 6.1 gives some values for $|V(G_{\text{crit}})|$ and $|E(G_{\text{crit}})|$ using Eqs (6.4) and (6.5). The theoretical value for c is around 1.152, which is remarkably close to the empirical results presented in Table 6.2. In this table, generated from real data, the probability $P_{|E(G_{\text{crit}})|}$ that $|E(G_{\text{crit}})| \leq \frac{1}{2}|E(G)|$ tends to 0 when $c < 1.15$ and it tends to 1 when $c \geq 1.15$ and n increases. We found this match between the empirical and the theoretical results

[1] As is usual in the theory of random graphs, we use the terms 'almost every' and 'almost surely' to mean 'with probability tending to 1 as $m \to \infty$'.

most pleasant, and this is why we consider that this a random graph, conditioned on being simple, strongly resembles the random graph from the uniform model $\mathcal{G}(m, n)$.

d	T	b	$\lvert V(G_{\text{crit}})\rvert$	$\lvert E(G_{\text{crit}})\rvert$	c
1.734	0.510	0.706	$0.399n$	$0.498n$	1.153
1.736	0.509	0.707	$0.400n$	$0.500n$	1.152
1.738	0.508	0.708	$0.401n$	$0.501n$	1.151
1.739	0.508	0.708	$0.401n$	$0.501n$	1.150
1.740	0.507	0.709	$0.401n$	$0.502n$	1.149

Table 6.1: Determining the c value theoretically.

We now briefly argue that the expected number of iterations to obtain a simple graph $G = G(h_1, h_2)$ is constant for $m = cn$ and $c = 1.15$. Let p be the probability of generating a random graph G without loops and without multiple edges. If p is bounded from below by some positive constant, then we are done, because the expected number of iterations to obtain such a graph is then $1/p = O(1)$.

Let X be a random variable counting the number of iterations to generate G. Variable X follows a geometric distribution with $P(X = i) = p(1-p)^{i-1}$. So, the expected number of iterations to generate G is $N_i(X) = \sum_{j=1}^{\infty} jP(X = j) = 1/p$ and its variance is $V(X) = (1-p)/p^2$.

Let ξ be the space of edges in G that may be generated by h_1 and h_2. The graphs generated in this step are undirected and the number of possible edges in ξ is given by $|\xi| = \binom{m}{2}$. The number of possible edges that might become a multiple edge when the jth

c	URLs (n)						
	10^3	10^4	10^5	10^6	2×10^6	3×10^6	4×10^6
1.13	0.22	0.02	0.00	0.00	0.00	0.00	0.00
1.14	0.35	0.15	0.00	0.00	0.00	0.00	0.00
1.15	0.46	0.55	0.65	0.87	0.95	0.97	1.00
1.16	0.67	0.90	1.00	1.00	1.00	1.00	1.00
1.17	0.82	0.99	1.00	1.00	1.00	1.00	1.00

Table 6.2: Probability $P_{|E_{\text{crit}}|}$ that $|E(G_{\text{crit}})| \leq n/2$ for different c values and different number of keys for a collections of URLs.

edge is added to G is $j-1$, and the incremental construction of G implies that $p(m)$ is:

$$p(m) = \prod_{j=1}^{n} \frac{\binom{m}{2} - (j-1)}{\binom{m}{2}} = \prod_{j=0}^{n-1} \frac{\binom{m}{2} - j}{\binom{m}{2}}.$$

As $m = cn$ we can rewrite the probability $p(n)$ as:

$$p(n) = \prod_{j=0}^{n-1} 1 - \left(\frac{2j}{c^2n^2 - cn} \right).$$

Using an asymptotic estimate from Palmer [64] that states that the inequality $f_1(x) \leq f_2(x)$ is true $\forall\, x \in \Re$ for two functions $f_1 : \Re \rightarrow \Re$ and $f_2 : \Re \rightarrow \Re$ defined as $f_1(x) = 1 - x$ and $f_2(x) = e^{-x}$, we have

$$p(n) \leq \prod_{j=0}^{n-1} e^{-\left(\frac{2j}{c^2n^2 - cn}\right)} = e^{-\left(\frac{n-1}{c^2n - c}\right)}.$$

for $x = \frac{2j}{c^2n^2 - cn}$. Thus,

$$\lim_{n \to \infty} p(n) \simeq e^{-\frac{1}{c^2}}. \tag{6.6}$$

As $N_i(X) = 1/p$ then $N_i(X) \simeq e^{\frac{1}{c^2}} = 2.13$ (recall $c = 1.15$). Therefore, as the expected number of iterations is $O(1)$, the mapping step takes $O(n)$ time.

Ordering Step

The procedure Ordering (G, G_{crit}, G_{ncrit}) receives as input the graph G and partitions G into the two subgraphs G_{crit} and G_{ncrit}, so that $G = G_{\text{crit}} \cup G_{\text{ncrit}}$. For that, the procedure iteratively remove all vertices of degree 1 until it is done.

Figure 6.3(a) presents a sample graph with 9 vertices and 8 edges, where the degree of a vertex is shown besides each vertex. Applying the ordering step in this graph, the 5-vertex graph showed in Figure 6.3(b) is obtained. All vertices with degree 0 are non-critical vertices and the others are critical vertices. In order to determine the vertices in V_{scrit} we collect all vertices $v \in V(G_{\text{crit}})$ with at least one vertex u that is in $\text{Adj}(v)$ and in $V(G_{\text{ncrit}})$, as the vertex 8 in Figure 6.3(b).

Analysis of the Ordering Step

The time complexity of the ordering step is $O(|V(G)|)$ (see [26]). As $|V(G)| = m = cn$, the ordering step takes $O(n)$ time.

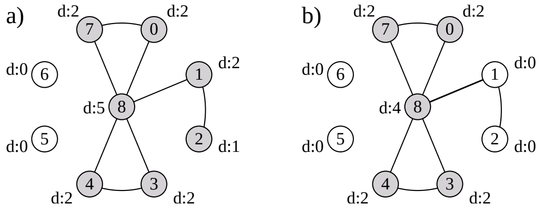


Figure 6.3: Ordering step for a graph with 9 vertices and 8 edges.

Searching Step

In the searching step, the key part is the *perfect assignment problem*: find $g : V(G) \to \mathbb{Z}$ such that the function $mphf : E(G) \to \mathbb{Z}$ defined by

$$mphf(e) = g(a) + g(b) \qquad (e = \{a, b\}) \tag{6.7}$$

is a bijection from $E(G)$ to $[0, n-1]$ (recall $n = |S| = |E(G)|$). We are interested in a labelling $g : V \to \mathbb{Z}$ of the vertices of the graph $G = G(h_1, h_2)$ with the property that if x and y are keys in S, then $g(h_1(x)) + g(h_2(x)) \neq g(h_1(y)) + g(h_2(y))$; that is, if we associate to each edge the sum of the labels on its endpoints, then these values should be all distinct. Moreover, we require that all the sums $g(h_1(x)) + g(h_2(x))$ $(x \in S)$ fall between 0 and $|E(G)| - 1 = n - 1$, so that we have a bijection between S and $[0, n-1]$.

The procedure Searching $(G, G_{\text{crit}}, G_{\text{ncrit}}, g)$ receives as input G, G_{crit}, G_{ncrit} and finds a suitable $\lfloor \log |V(G)| \rfloor + 1$ bit value for each vertex $v \in V(G)$, stored in the array g. This step is first performed for the vertices in the critical subgraph G_{crit} of G (the 2-core of G) and then it is performed for the vertices in G_{ncrit} (the non-critical subgraph of G that contains the "acyclic part" of G). The reason the assignment of the g values is first performed on the vertices in G_{crit} is to resolve reassignments as early as possible (such reassignments are consequences of the cycles in G_{crit} and are depicted hereinafter).

Assignment of Values to Critical Vertices

The labels $g(v)$ $(v \in V(G_{\text{crit}}))$ are assigned in increasing order following a greedy strategy where the critical vertices v are considered one at a time, according to a breadth-first search on G_{crit}. If a candidate value x for $g(v)$ is forbidden because setting $g(v) = x$ would create two edges with the same sum, we try $x + 1$ for $g(v)$. This fact is referred to as a *reassignment*.

Let A_E be the set of addresses assigned to edges in $E(G_{\text{crit}})$. Initially $A_E = \emptyset$. Let x be a candidate value for $g(v)$. Initially $x = 0$. Considering the subgraph G_{crit} in Figure 6.3(b), a step by step example of the assignment of values to vertices in G_{crit} is presented in Figure 6.4. Initially, a vertex v is chosen, the assignment $g(v) = x$ is made and x is set to $x + 1$. For example, suppose that vertex 8 in Figure 6.4(a) is chosen, the assignment $g(8) = 0$ is made and x is set to 1.

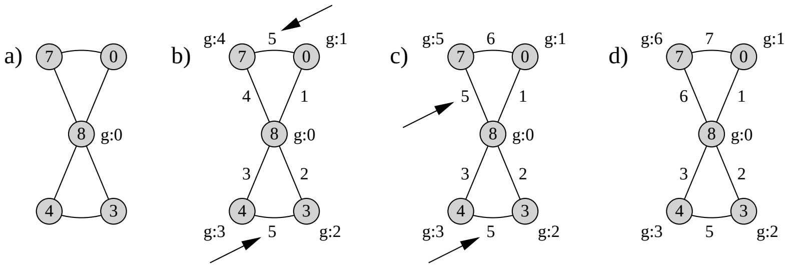


Figure 6.4: Example of the assignment of values to critical vertices.

In Figure 6.4(b), following the adjacency list of vertex 8, the unassigned vertex 0 is reached. At this point, we collect in the temporary variable Y all adjacencies of vertex 0 that have been assigned an x value, and $Y = \{8\}$. Next, for all $u \in Y$, we check if $g(u) + x \notin A_E$. Since $g(8) + 1 = 1 \notin A_E$, then $g(0)$ is set to 1, x is incremented by 1 (now $x = 2$) and $A_E = A_E \cup \{1\} = \{1\}$. Next, vertex 3 is reached, $g(3)$ is set to 2, x is set to 3 and $A_E = A_E \cup \{2\} = \{1, 2\}$. Next, vertex 4 is reached and $Y = \{3, 8\}$. Since $g(3) + 3 = 5 \notin A_E$ and $g(8) + 3 = 3 \notin A_E$, then $g(4)$ is set to 3, x is set to 4 and $A_E = A_E \cup \{3, 5\} = \{1, 2, 3, 5\}$. Finally, vertex 7 is reached and $Y = \{0, 8\}$. Since $g(0) + 4 = 5 \in A_E$, x is incremented by 1 and set to 5, as depicted in Figure 6.4(c). Since $g(8) + 5 = 5 \in A_E$, x is again incremented by 1 and set to 6, as depicted in Figure 6.4(d). These two reassignments are indicated by the arrows in Figure 6.4. Since $g(0)+6 = 7 \notin A_E$ and $g(8) + 6 = 6 \notin A_E$, then $g(7)$ is set to 6 and $A_E = A_E \cup \{6, 7\} = \{1, 2, 3, 5, 6, 7\}$. This finishes the algorithm.

Assignment of Values to Non-Critical Vertices

As G_{ncrit} is acyclic, we can impose the order in which addresses are associated with edges in G_{ncrit}, making this step simple to solve by a standard depth first search algorithm. Therefore, in the assignment of values to vertices in G_{ncrit} we benefit from the unused addresses in the gaps left by the assignment of values to vertices in G_{crit}. For that, we start the depth-first search from the vertices in V_{scrit} because the g values for these critical vertices have already been assigned and cannot be changed.

Considering the subgraph G_{ncrit} in Figure 6.3(b), a step by step example of the assignment of values to vertices in G_{ncrit} is presented in Figure 6.5. Figure 6.5(a) presents the initial state of the algorithm. The critical vertex 8 is the only one that has non-critical neighbors. In the example presented in Figure 6.4, the addresses $\{0, 4\}$ were not used. So, taking the first unused address 0 and the vertex 1, which is reached from the vertex 8, $g(1)$ is set to $0 - g(8) = 0$, as shown in Figure 6.5(b). The only vertex that is reached from vertex 1 is vertex 2, so taking the unused address 4 we set $g(2)$ to $4 - g(1) = 4$, as shown in Figure 6.5(c). This process is repeated until the UnAssignedAddresses list becomes empty.

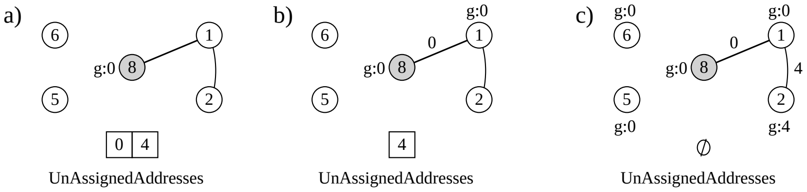


Figure 6.5: Example of the assignment of values to non-critical vertices.

Analysis of the Searching Step

We shall demonstrate that (i) the maximum value assigned to an edge is at most $n - 1$ (that is, we generate a minimal perfect hash function), and (ii) the perfect assignment problem (determination of g) can be solved in expected time $O(n)$ if the number of edges in G_{crit} is at most $\frac{1}{2}|E(G)|$.

We focus on the analysis of the assignment of values to critical vertices because the assignment of values to non-critical vertices can be solved in linear time by a depth first search algorithm.

We now define certain complexity measures. Let $I(v)$ be the number of times a candidate value x for $g(v)$ is incremented. Let N_t be the total number of times that candidate values x are incremented. Thus, we have $N_t = \sum I(v)$, where the sum is over all $v \in V(G_{\text{crit}})$.

For simplicity, we shall suppose that G_{crit}, the 2-core of G, is connected.[2] The fact that every edge is either a tree edge or a back edge (see, e.g., [24]) then implies the following.

[2] The number of vertices in G_{crit} outside the giant component is provably very small for $c = 1.15$; see [8, 49, 65].

Theorem 8 The number of back edges N_{bedges} of $G = G_{\text{crit}} \cup G_{\text{ncrit}}$ is given by $N_{\text{bedges}} = |E(G_{\text{crit}})| - |V(G_{\text{crit}})| + 1$.

Our next result concerns the maximal value $A_{\max}$ assigned to an edge $e \in E(G_{\text{crit}})$ after the assignment of g values to critical vertices.

Theorem 9 We have $A_{\max} \leq 2|V(G_{\text{crit}})| - 3 + 2N_t$.

Proof: The assignment of g values to critical vertices starts from 0, and each edge e receives the label *mphf*(e) as given by Eq. (6.7). The g value for each vertex v in $V(G_{\text{crit}})$ is assigned only once. Consider now two possibilities: (i) If $N_t = 0$, (that is, no increment for a candidate value was necessary) then the g values will be assigned to the vertices sequentially. Therefore, the greatest and the second greatest values assigned to two vertices v and u are $g(v) = |V(G_{\text{crit}})| - 1$ and $g(u) = |V(G_{\text{crit}})| - 2$, respectively. Thus, $A_{max} \leq (|V(G_{\text{crit}})| - 1) + (|V(G_{\text{crit}})| - 2)$ in the worst case. (ii) If $N_t > 0$ then a candidate value x is incremented by one each time the value is forbidden. Thus, in the worst case, $A_{max} \leq |V(G_{\text{crit}})| - 1 + N_t + |V(G_{\text{crit}})| - 2 + N_t \leq 2|V(G_{\text{crit}})| - 3 + 2N_t$. □

Maximal Value Assigned to an Edge

In this section we present the following conjecture.

Conjecture 1 For a random graph G with $|E(G_{\text{crit}})| \leq n/2$ and $|V(G)| = 1.15n$, it is always possible to generate a minimal perfect hash function because the maximal value $A_{\max}$ assigned to an edge $e \in E(G_{\text{crit}})$ is at most $n - 1$.

Let us assume for the moment that $N_t \leq N_{\text{bedges}}$. Then, from Theorems 8 and 9, we have $A_{\max} \leq 2|V(G_{\text{crit}})| - 3 + 2N_t \leq 2|V(G_{\text{crit}})| - 3 + 2N_{\text{bedges}} \leq 2|V(G_{\text{crit}})| - 3 + 2(|E(G_{\text{crit}})| - |V(G_{\text{crit}})| + 1) \leq 2|E(G_{\text{crit}})| - 1$. As by hypothesis $|E(G_{\text{crit}})| \leq n/2$, we have $A_{\max} \leq n - 1$, as required.

In the mathematical analysis of our algorithm, what is left open is a single problem: prove that $N_t \leq N_{\text{bedges}}$.[3]

We now show experimental evidence that $N_t \leq N_{\text{bedges}}$. Considering Eqs (6.4) and (6.5), the expected values for $|V(G_{\text{crit}})|$ and $|E(G_{\text{crit}})|$ for $c = 1.15$ are $0.401n$ and $0.501n$, respectively. From Theorem 8, $N_{\text{bedges}} = 0.501n - 0.401n + 1 = 0.1n + 1$. Table 6.3 presents

[3] Bollobás and Pikhurko [9] have investigated a very close vertex labelling problem for random graphs. However, their interest was on denser random graphs, and it seems that different methods will have to be used to attack the sparser case that we are interested in here.

the maximal value of N_t obtained during 10,000 executions of the algorithm for different sizes of S. The maximal value of N_t was always smaller than $N_{\text{bedges}} = 0.1n + 1$ and tends to $0.059n$ for $n \geq 1{,}000{,}000$.

n	Maximal value of N_t
10,000	$0.067n$
100,000	$0.061n$
1,000,000	$0.059n$
2,000,000	$0.059n$

Table 6.3: The maximal value of N_t for different number of URLs.

Time Complexity

We now show that the time complexity of determining $g(v)$ for all critical vertices $x \in V(G_{\text{crit}})$ is $O(|V(G_{\text{crit}})|) = O(n)$. For each unassigned vertex v, the adjacency list of v, which we call Adj(v), must be traversed to collect the set Y of adjacent vertices that have already been assigned a value. Then, for each vertex in Y, we check if the current candidate value x is forbidden because setting $g(v) = x$ would create two edges with the same endpoint sum. Finally, the edge linking v and u, for all $u \in Y$, is associated with the address that corresponds to the sum of its endpoints. Let $d_{\text{crit}} = 2|E(G_{\text{crit}})|/|V(G_{\text{crit}})|$ be the average degree of G_{crit}, note that $|Y| \leq |\mathrm{A}dj(v)|$, and suppose for simplicity that $|\mathrm{A}dj(v)| = O(d_{\text{crit}})$. Then, putting all these together, we see that the time complexity of this procedure is

$$\begin{aligned} C(|V(G_{\text{crit}})|) &= \textstyle\sum_{v \in V(G_{\text{crit}})} \left[\, |\mathrm{A}dj(v)| + (I(v) \times |Y|) + |Y| \right] \\ &\leq \textstyle\sum_{v \in V(G_{\text{crit}})} (2 + I(v))|\mathrm{A}dj(v)| = 4|E(G_{\text{crit}})| + O(N_t d_{\text{crit}}). \end{aligned}$$

As $d_{\text{crit}} = 2 \times 0.501n/0.401n \simeq 2.499$ (a constant) we have $O(|E(G_{\text{crit}})|) = O(|V(G_{\text{crit}})|)$. Supposing that $N_t \leq N_{\text{bedges}}$, we have, from Theorem 8, that $N_t \leq |E(G_{\text{crit}})| - |V(G_{\text{crit}})| + 1 = O(|E(G_{\text{crit}})|)$. We conclude that $C(|V(G_{\text{crit}})|) = O(|E(G_{\text{crit}})|) = O(|V(G_{\text{crit}})|)$. As $|V(G_{\text{crit}})| \leq |V(G)|$ and $|V(G)| = cn$, the time required to determine g on the critical vertices is $O(n)$.

6.1.3 Comparing the BKZ and CHM Algorithms

In this section we compare the BKZ algorithm with the CHM algorithm experimentally. For this reason the two algorithms were implemented in the C language and are available as part of the C Minimal Perfect Hashing Library, which can be downloaded at `http://cmph.sf.net`. Our data consists of a collection of 100 million universe resource locations (URLs) collected from the Web. The average length of a URL in the collection is 63 bytes. All experiments were carried out on a computer running the Linux operating system, version 2.6.7, with a 2.4 gigahertz processor and 4 gigabytes of main memory.

Table 6.4 presents the main characteristics of the two algorithms. The number of edges in the graph $G = (V, E)$ is $|S| = n$, i.e., the number of keys in the input set S. The number of vertices of G is equal to $1.15n$ and $2.09n$ for the BKZ algorithm and the CHM algorithm, respectively. This measure is related to the amount of space to store the array g. This improves the space required to store a function in the BKZ algorithm to 55% of the space required by the CHM algorithm. The number of critical edges is $\frac{1}{2}|E(G)|$ and 0 for the BKZ algorithm and the CHM algorithm, respectively. The BKZ algorithm generates random graphs that contain cycles with high probability and the CHM algorithm generates acyclic random graphs. Finally, the CHM algorithm generates order preserving functions whereas the BKZ algorithm does not preserve order.

	c	$\vert E(G)\vert$	$\vert V(G)\vert = \vert g\vert$	$\vert E(G_{\rm crit})\vert$	G	Order preserving
BKZ algorithm	1.15	n	cn	$0.5\vert E(G)\vert$	cyclic	no
CHM algorithm	2.09	n	cn	0	acyclic	yes

Table 6.4: Main characteristics of the algorithms.

Table 6.5 presents time measurements to generate the MPHFs. All times are in seconds. The table entries are averages over 50 trials. The column labelled N_i gives the number of iterations to generate the random graph G in the mapping step of the algorithms. The next columns give the running times for the mapping plus ordering steps together and the searching step for each algorithm. The last column gives the percentage gain of the BKZ algorithm over the CHM algorithm.

The mapping step of the BKZ algorithm is faster because the expected number of iterations in the mapping step to generate G are 2.13 and 2.92 for the BKZ algorithm and the CHM algorithm, respectively. The graph G generated by the BKZ algorithm has $1.15n$ vertices, against $2.09n$ for the CHM algorithm. These two facts make the BKZ algorithm faster in the mapping step. The ordering step of the BKZ algorithm is approximately equal to the time to check if G is acyclic for the CHM algorithm. The searching step of the CHM algorithm is faster, but the total time of the BKZ algorithm is, on average, approximately 59% faster than the CHM algorithm.

n	BKZ algorithm				CHM algorithm				Gain (%)
	N_i	Map+Ord	Search	Total	N_i	Map+Ord	Search	Total	
1,562,500	2.28	8.54	2.37	10.91	2.70	14.56	1.57	16.13	48
3,125,000	2.16	15.92	4.88	20.80	2.85	30.36	3.20	33.56	61
6,250,000	2.20	33.09	10.48	43.57	2.90	62.26	6.76	69.02	58
12,500,000	2.00	63.26	23.04	86.30	2.60	117.99	14.94	132.92	54
25,000,000	2.00	130.79	51.55	182.34	2.80	262.05	33.68	295.73	62
50,000,000	2.07	273.75	114.12	387.87	2.90	577.59	73.97	651.56	68
100,000,000	2.07	567.47	243.13	810.60	2.80	1,131.06	157.23	1,288.29	59

Table 6.5: Time measurements for the BKZ algorithm and the CHM algorithm to generate MPHFs.

n	BKZ algorithm $c = 1.00$				BKZ algorithm $c = 0.93$			
	N_i	Map+Ord	Search	Total	N_i	Map+Ord	Search	Total
12,500,000	2.78	76.68	25.06	101.74	3.04	76.39	25.80	102.19

Table 6.6: Time measurements for the BKZ algorithm to generate MPHFs, tuned with $c = 1.00$ and $c = 0.93$.

The experimental results fully backs the theoretical results. It is important to notice the times for the searching step: for both algorithms they are not the dominant times, and the experimental results clearly show a linear behavior for the searching step.

We now present a heuristic that reduces the space requirement to any given value between $1.15n$ words and $0.93n$ words. The heuristic reuses, when possible, the set of x values that caused reassignments, just before trying $x + 1$ (see Section 6.1.2). The lower limit $c = 0.93$ was obtained experimentally. We generate 10,000 random graphs for each size n ($n = 10^5$, 5×10^5, 10^6, 2×10^6). With $c = 0.93$ we were always able to generate an MPHF, but with $c = 0.92$ we never succeeded. Decreasing the value of c leads to an increase in the number of iterations to generate G. For example, for $c = 1$ and $c = 0.93$, the analytical expected number of iterations are 2.72 and 3.17, respectively (for $n = 12{,}500{,}000$, the number of iterations are 2.78 for $c = 1$ and 3.04 for $c = 0.93$). Table 6.6 presents the total times to construct a function for $n = 12{,}500{,}000$, with an increase from 86.31 seconds for $c = 1.15$ (see Table 6.5) to 101.74 seconds for $c = 1$ and to 102.19 seconds for $c = 0.93$.

We compared the BKZ algorithm with the ones proposed by Pagh [61] and Dietzfelbinger and Hagerup [29], respectively. The authors sent to us their source code. In their implementation the key set is a set of random integers. We modified our implementation to generate an MPHF from a set of random integers in order to make a fair comparison. For a set of 10^6 random integers, the times to generate a minimal perfect hash function were $2.7s$, $4s$ and $4.5s$ for the BKZ algorithm, Pagh's algorithm and Dietzfelbinger and Hagerup's algorithm, respectively. Thus, the BKZ algorithm was 48% faster than Pagh's algorithm and 67% faster than Dietzfelbinger and Hagerup's algorithm, on average. This gain was maintained for sets with different sizes. The BKZ algorithm needs kn ($k \in [0.93, 1.15]$) words to store the resulting function, while Pagh's algorithm needs kn ($k > 2$) words and Dietzfelbinger and Hagerup's algorithm needs kn ($k \in [1.13, 1.15]$) words. The time to generate the functions is inversely proportional to the value of k.

6.2 The RAM Algorithm: Dealing with Connected Components with a Single Cycle for $r = 2$

Although the BKZ algorithm still generates MPHFs that require $O(n \log n)$ bits to be stored, the techniques used in its design can also be used to speedup the execution time of the RAM algorithm, which generates MPHFs that require $(3 + \epsilon)n$ bits of storage space, where $\epsilon > 0$. Remember that the RAM algorithm originally works on random bipartite graphs with no cycles. But, if each connected component of the random graph has just one cycle with the same number of edges and vertices, then it is possible to build MPHFs 40% faster on average. In Section 6.2.1 we present the design of the optimized version of the RAM algorithm. In Section 6.2.2 we experimentally compare the optimized version of the RAM algorithm with the version of the RAM algorithm presented in Chapter 2.

6.2.1 Design of the Optimized Version of The RAM Algorithm

The first two steps of the RAM algorithm builds an one-to-one mapping between a key set S (or, equivalently, the edge set E) and the vertex set V of an acyclic bipartite random graph $G = (V, E)$, $|E| = n$, $|V| = m = cn$ and $c > 2$. But if each connected component of G has just one cycle with the same number of edges and vertices, then it is possible to create an one-to-one mapping between edges and vertices in this case. This is interesting because the RAM algorithm will run much faster for values of c close to 2. We now show how to adapt the first two steps of the RAM algorithm to deal with connected components of G containing a single cycle.

Definition 25 Let $C = \{G' = (V', E') \mid V' \subseteq V, E' \subseteq E\}$ be the set of connected components of G.

We now use the same idea presented in Section 6.1. For each connected component $G' \in C$, we split G' into two subgraphs $G' = G'_{\text{crit}} \cup G'_{\text{ncrit}}$, where $G'_{\text{crit}} = (V_{\text{crit}}, E_{\text{crit}})$ is the subcomponent of G' that contains cycles and $G'_{\text{ncrit}} = (V_{\text{ncrit}} \cup V_{\text{scrit}}, E_{\text{ncrit}})$ is the subcomponent with no cycles. The algorithm presented in Section 2.1.1 to test whether a graph contains cycles can be easily adapted to obtain G'_{crit} and G'_{ncrit}. The resulting graph of the test corresponds to G'_{crit} and $G'_{\text{ncrit}} = G' - G'_{\text{crit}}$. Now we do not restart the mapping step because G'_{crit} is not empty. Instead, we first use a depth-first search algorithm to build an one-to-one mapping for E_{crit} and V_{crit} and, then, use the assigning step of the RAM algorithm for E_{ncrit} and V_{ncrit}. We just restart from the mapping step if G'_{crit} is not assignable (i.e., G'_{crit} contains more than one cycle).

Figure 6.6 illustrates the assignment of G'_{crit}. Figure 6.6(b) shows the order in which a depth-first search algorithm will visit each vertex. The algorithm starts from a given vertex $v \in V_{\text{crit}}$, lets say $v = 0$, and set $g[v]$ to 0. Then, the depth-first search goes on one of the vertices adjacent to vertex $v = 0$. Let $u = 2$ be that vertex. Then, $g[u] = (x - g[v]) \bmod 2$, where:

$$x = \begin{cases} 0, \text{ if } u < |V|/2 \\ 1, \text{ otherwise.} \end{cases}$$

This will associate vertex $u = 2$ with the current edge $\{0, 2\}$. Note that when we are visiting edge $e = \{0, 3\}$, which closes the cycle, its two vertices were already assigned. Therefore, we cannot change the value assigned to vertex 0 and vertex 0 is supposed to be associated with e. In this case, there is no problem because $g[0]$ received the same value previously assigned and the algorithm ends because all edges were visited.

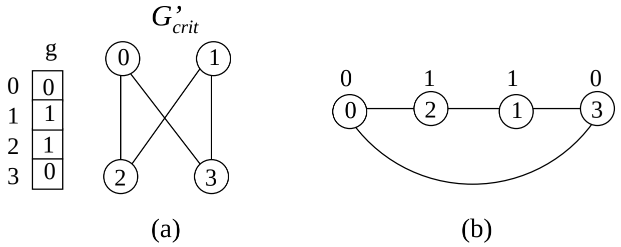


Figure 6.6: (a) Assignment for a connected component with a single cycle with 4 vertices and 4 edges. (b) Order in which a depth-first search algorithm will visit each vertex starting from vertex 0.

The assignment of G'_{crit} is not possible when the length of the cycle is not a multiple of four. Figure 6.7 illustrates a case where it is not possible to finish the assignment successfully. In Figure 6.7(b), if we traverse the cycle in the opposite way the depth-first search algorithm did, the values of g for the current visited vertex are: $0, 1, 0, 0, 1, 1$. Note that vertex 5 was associated with two edges: $\{0, 5\}$ and $\{2, 5\}$. It happens because $(g[0] + g[5]) \bmod 2 = 1$ and $(g[2] + g[5]) \bmod 2 = 1$. To avoid this we must have a sequence of g values alternating double zeros and double ones with the same number of zeros and ones, i.e., $0, 0, 1, 1, \ldots, 0, 0, 1, 1$. In Figure 6.6(b), if we traverse the cycle in the opposite way the depth-first search algorithm did, the values of g are: $0, 0, 1, 1$. Therefore, the length of the cycles must be a multiple of four.

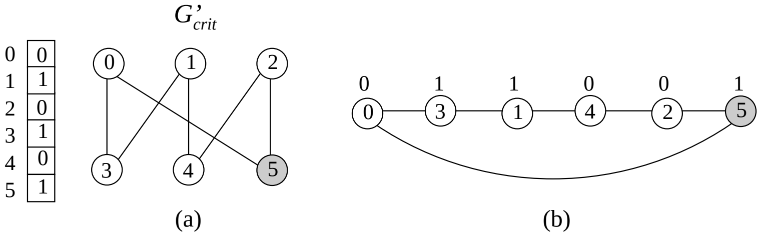


Figure 6.7: (a) A non-assignable cycle with 6 vertices and 6 edges. (b) Order in which a depth-first search algorithm will visit each vertex starting from vertex 0.

Definition 26 A connected component $G' \in C$ is *assignable* if and only if it contains a single cycle with the same number of vertices and edges, and its length is a multiple of four.

The ranking step of the RAM algorithm does not need to be changed. To finish we just need to show that it is possible to obtain a bipartite random graph G with no non-assignable connected component with high probability. This is equivalent to show that, with high probability, G has no cycle of length $2(2l-1)$ for $l = 1, 2, 3, \ldots,$ and each vertex $v \in V$ will be present in just one cycle. In [27] it is shown that, for $c \geq 2$ and with probability tending to one, a vertex $v \in V$ cannot participate in two different cycles of size two or higher. Then, it remains to prove the following theorem.

Theorem 10 Let $G = (V, E)$ be a bipartite random graph with n edges and m vertices. Then, if $m = cn$ holds for $c > 2$, the probability that G has no cycle of length $2(2l-1)$ for $l = 1, 2, 3, \ldots,$ for $n \to \infty$, is:

$$Pr_b = \frac{\sqrt{1 - \left(\frac{2}{c}\right)^2}}{\left(1 - \left(\frac{2}{c}\right)^4\right)^{\frac{1}{4}}} \tag{6.8}$$

PROOF. As shown in Theorem 4, the random variable $\mathcal{C}_e(G)$ that measures the number of cycles of any even length in G converges to a Poisson distribution with parameter:

$$\lambda_e = \sum_{l=1}^{\infty} \frac{1}{2l} \left(\frac{2}{c}\right)^{2l} = -\frac{1}{2} \ln \left(1 - \left(\frac{2}{c}\right)^2\right). \tag{6.9}$$

Corresponding results hold for cycles with lengths in a given subset of $\{2, 4, 6, \dots\}$, as can be derived from the results of [48]. Let $\mathcal{C}_b(G)$ be a random variable that measures the number of bad cycles in G (cycles with lengths that are not multiple of four), which converges to a Poisson distribution with parameter:

$$\lambda_b = \sum_{l=1,3,5,7,\dots} \frac{1}{2l} \left(\frac{2}{c}\right)^{2l}. \tag{6.10}$$

From Eq. (6.9) we know that:

$$\begin{aligned}
\lambda_e &= \sum_{l=1,3,5,7,\dots} \frac{1}{2l} \left(\frac{2}{c}\right)^{2l} + \sum_{l=2,4,6,8,\dots} \frac{1}{2l} \left(\frac{2}{c}\right)^{2l} = -\frac{1}{2} \ln \left(1 - \left(\frac{2}{c}\right)^2\right) \\
\lambda_b &= -\frac{1}{2} \ln \left(1 - \left(\frac{2}{c}\right)^2\right) - \sum_{l=2,4,6,8,\dots} \frac{1}{2l} \left(\frac{2}{c}\right)^{2l} \\
&= -\frac{1}{2} \ln \left(1 - \left(\frac{2}{c}\right)^2\right) - \frac{1}{2} \sum_{l=1}^{\infty} \frac{1}{2l} \left(\left(\frac{2}{c}\right)^2\right)^{2l} \\
&= -\frac{1}{2} \ln \left(1 - \left(\frac{2}{c}\right)^2\right) + \frac{1}{4} \ln \left(1 - \left(\frac{2}{c}\right)^4\right)
\end{aligned}$$

Therefore, the probability that G has no bad cycle is given by:

$$Pr_b(\mathcal{C}_b(G) = 0) = e^{-\lambda_b} = \frac{\sqrt{1 - \left(\frac{2}{c}\right)^2}}{\left(1 - \left(\frac{2}{c}\right)^4\right)^{\frac{1}{4}}}.$$

□

For $c = 2.09$ we have $Pr_b = 0.458$, whereas the probability to obtain an acyclic bipartite random graph $Pr_a = 0.29$. This implies that $1/Pr_b = 2.18$ iterations are required on average to succeed in the version that deals with one single cycle of length multiple of four per connected component, whereas $1/Pr_a = 3.45$ iterations are required on average in the version that requires an acyclic bipartite random graph. Experimentally, we obtained $Pr_b = 0.463$ by generating $1,000$ random bipartite 2-graphs with $n = 10^7$ keys (edges), which is very close to the theoretical value.

6.2.2 Comparing the two Versions of the RAM Algorithm

In this section we evaluate the performance of the RAM algorithm when used to generate MPHFs [4]. We will consider two versions of the RAM algorithm: (i) the version that that works on random bipartite graphs with a single cycle per connected component; and (ii) the version that works on random bipartite graphs with no cycles, which is presented in Chapter 2.

For this reason the two versions of the RAM algorithm were implemented in the C language and are available under the GNU Lesser General Public License (LGPL) at `http://-cmph.sf.net`. The experiments were carried out on a computer running the Linux operating system, version 2.6, with a 1.86 gigahertz Intel Core 2 processor with a 4 megabyte L2 cache and 1 gigabyte of main memory. For the experiments we used two collections: (i) a set of 150 million randomly generated 4 byte long IP addresses, and (ii) a set of $1,024$ million 64 byte long (on average) URLs collected from the Web.

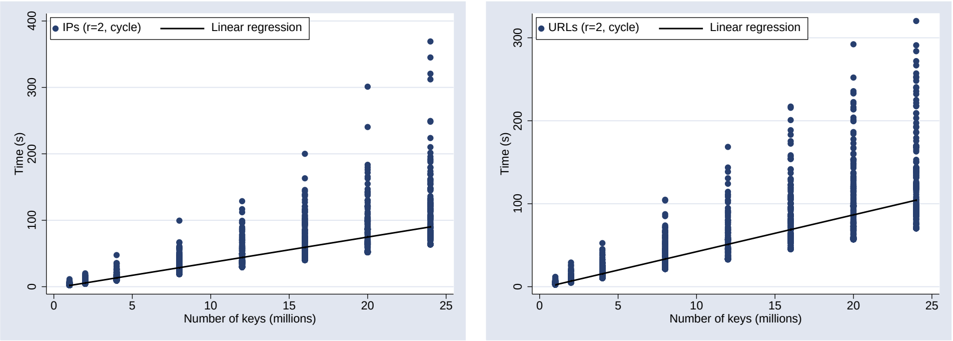
The runtime of the version of the RAM algorithm that deals with a single cycle per connected component has the same form of the one presented in Chapter 2, which is $\alpha n Z$ for an input of n keys, where α is some machine dependent constant that further depends on the length of the keys and Z is a random variable with geometric distribution. But now the mean of the geometric distribution is $1/Pr_b$ instead of $1/Pr_a$, where Pr_b and Pr_a are given in Eq. (6.8) and Eq. (2.7), respectively. All results in the experiments to compare the two versions of the RAM algorithm were obtained taking $c = 2.09$; the larger is c the faster are both versions of the RAM algorithm because both Pr_b and Pr_a increase continuously with c.

The values chosen for n were $1, 2, 4, 8, 12, 16, 20$ and 24 millions. Although we have 150 millions of random IPs and $1,024$ millions of URLs, on a PC with 1 gigabyte of main memory, both versions of the RAM algorithm are able to handle an input with at most 24 millions of keys. This is mainly because the sparse random hypergraph required to generate the functions is memory demanding. By using the same technique used in Chapter 2 to estimate the number of trials for each value of n we also obtained 300 trials in order to have a confidence level of 95%.

Figure 6.8 presents the runtime for each trial. In addition, the solid line corresponds to a linear regression model obtained from the experimental measurements. As we can see, the runtime for a given n has a considerable fluctuation, which gives a coefficient of determination $R^2 = 71\%$. However, the fluctuation also grows linearly with n. The observed

[4] The same conclusions are achieved when PHFs are generated.

fluctuation in the runtimes is as expected; recall that this runtime has the form $\alpha n Z$ with Z a geometric random variable with mean $1/Pr_b = 1/0.458$ for $c = 2.09$. Thus, the runtime has mean $\alpha n/Pr_b = 2.18\alpha n$ and standard deviation $\alpha n\sqrt{(1 - Pr_b)/(Pr_b)^2} = 1.61\alpha n$. Therefore, the standard deviation also grows linearly with n, as experimentally verified in Figure 6.8. It is important to remark that this version of the RAM algorithm has a smaller fluctuation than the version presented in Chapter 2 because $Pr_b > Pr_a$.



(a) IPs collection (b) URLs collection

Figure 6.8: Number of keys in S versus generation time for the RAM algorithm that works on random hypergraphs with a single cycle per connected component for $r = 2$. The solid line corresponds to a linear regression model for the generation time ($R^2 = 71\%$).

The version of the RAM algorithm that works on random bipartite graphs with a single cycle per connected component has the same behavior of the version that works on random acyclic bipartite graphs (see Figures 2.13 and 6.8), but runs considerably faster. This is because the geometric distribution now has mean $1/Pr_b$, where $Pr_b = 0.458$, whereas for the version of the RAM algorithm presented in Chapter 2 the geometric distribution has mean $1/Pr_a$, where $Pr_a = 0.29$.

To end this section we now compare the two versions of the RAM algorithm by taking $n = 1, 12$ and 24 millions of keys in the two collections and by considering generation time and storage space as metrics. Table 6.7 presents the respective confidence intervals for the generation time, which is given by the average time ± the distance from average time considering a confidence level of 95%, and the respective values for the storage space. It is possible to see that when cycles are allowed the RAM algorithm is approximately 40% faster to generate the functions. Also, the generation time, as expected, is influenced by the key length (IPs are 4 bytes long and URL are 64 bytes long on average), and the storage space is not. Finally, the most compact functions are generated when $r = 2$ and cycles are allowed.

n	RAM algorithm		Generation Time (sec)		Storage Space	
			IPs	URLs	Bits/Key	Size (MB)
1×10^6	$r = 2$	cycle	3.26 ± 0.16	3.69 ± 0.18	3.35	0.40
		no cycle	4.45 ± 0.42	4.75 ± 0.41	3.60	0.43
12×10^6	$r = 2$	cycle	41.33 ± 2.02	47.96 ± 2.45	3.35	4.79
		no cycle	58.70 ± 5.20	64.22 ± 6.37	3.60	5.15
24×10^6	$r = 2$	cycle	91.32 ± 5.2	104.77 ± 5.58	3.35	9.58
		no cycle	135.92 ± 13.2	146.93 ± 14.09	3.60	10.30

Table 6.7: Comparison of the two versions of the RAM algorithm considering generation time and storage space, and using $n = 1, 12,$ and 24 millions of keys for the two collections.

6.3 Conclusions

In this chapter we presented techniques that allow the generation of MPHFs based on random graphs with cycles. This implies that the functions are generated faster and are more compact than the ones generated based on acyclic random graphs. The techniques were applied to the design of two algorithms: the BKZ algorithm and the RAM algorithm.

First we showed how the BKZ algorithm improves the space requirement of the MPHFs generated by the algorithm proposed by Czech, Havas and Majewski [25] from $cn \log n$ bits, for $c > 2$ to $c'n \log n$, where $c' \in [0.93, 1.15]$. That is, our resulting functions are stored in approximately 55% of the space required to store the ones generated by the CHM algorithm. However, the resulting MPHFs still requires $O(n \log n)$ bits to be stored, that is a factor $\log n$ from the optimal. We also showed that the BKZ algorithm runs approximately 59% faster than the CHM algorithm on average.

Second, we used techniques similar to the ones used in the design of the BKZ algorithm to speedup the execution time of the RAM algorithm presented in Chapter 2, which generates MPHFs that require $(3 + \epsilon)n$ bits of storage space, where $\epsilon > 0$. We showed that if each connected component of the random graph has just one cycle with the same number of edges and vertices, then it is possible to tune the RAM algorithm to build MPHFs 40% faster on average.

Chapter 7

Indexing Internal Memory With MPHFs

The objective of this chapter is to show that MPHFs provide the best tradeoff between space usage and lookup time when compared to other hashing schemes. It was not the case in the past because the space overhead to store MPHFs was $O(\log n)$ bits per key for practical algorithms [25, 55]. Therefore, a better performace in terms of time and space was obtained by using a single hash function and resolving collisions with linear probing [45, 51]. However, the new results on MPHFs presented in Chapter 2 have motivated this work, since the resulting MPHFs require approximately 2.6 bits per key of space overhead and can be evaluated in $O(1)$ time.

We obtained interesting results in two scenarios: (i) when the MPHF description fits in the CPU cache and (ii) when it cannot be entirely placed in the CPU cache. In the first scenario we show that the other hashing schemes cannot outperform minimal perfect hashing when the hash table occupancy is greater than 55%. An MPHF requiring just 2.6 bits per key of storage space permits to store sets on the order of 10 million keys in a 4 megabyte CPU cache, which is enough for a large range of applications. In the second scenario, other hashing schemes require a hash table occupancy lower than 75% to obtain the same performance attained by minimal perfect hashing. For both scenarios, the space overhead of minimal perfect hashing is within a factor of $O(\log n)$ bits lower than other hashing schemes. A preliminary version of these results was presented in [13].

This chapter is organized as follows. In Section 7.1 we describe the hashing schemes used in the study. In Section 7.2 we present the experimental results to compare the considered hashing schemes. Finally, in Section 7.3 we conclude this chapter.

7.1 The Algorithms

In this section we describe the hashing methods we used to compare minimal perfect hashing with, namely, linear hashing, quadratic hashing, double hashing, dense hashing, cuckoo hashing and sparse hashing. The hash table entries store items, and each item is composed by a key and possibly some data, i.e., a pair $< k, d >$. All the methods analyzed use collision resolution by open addressing, that is, they look at various positions of the hash table one by one until it either finds the key k being searched for or it finds an empty position [51]. In contrast, collision resolution could also be made by chaining, in which a linked list is used to store items that collided in the same table position. Open addressing is preferred over chaining if we are interested in lookup time, since it has a better locality of reference and reduces the number of cache misses.

The hash table structure used by linear hashing, quadratic hashing, double hashing, dense hashing and cuckoo hashing is shown in Figure 7.1. Every table position has a pointer, initially pointing to an empty value. When an item is inserted in the table, the pointer of the corresponding position starts to refer to it. The hash table structures for sparse hashing and minimal perfect hashing are presented in Sections 7.1.5 and 7.1.6, respectively.

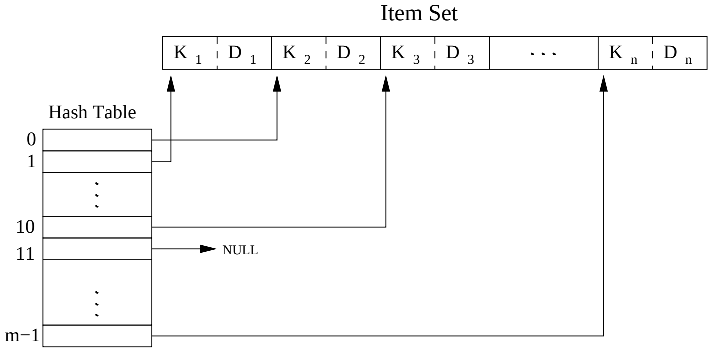


Figure 7.1: Hash table used for linear hashing, quadratic hashing, double hashing, dense hashing and cuckoo hashing.

Note that we should not insert the item itself in the table, since the allocated empty positions would cause an expressive waste of memory space, especially if the item occupies several bytes. Hence, the wasted space is reduced by using only one pointer per empty position. If we define p as the pointer size in bits, the space overhead for methods that use the structure in Figure 7.1 is $p \times m$ bits for a hash table of size m. For a 64 bits architecture, $p = 64$ bits.

Throughout this section we shall use $\oplus_m$ as a notation for an addition modulus m. For instance, we may describe the operation $(a + b) \bmod m$ as $a \oplus_m b$.

7.1.1 Linear Hashing

Linear hashing is considered one of the simplest open addressing schemes available [51, 76]. It uses a hash function $h : S \rightarrow [0, m-1]$ and tests positions $h(k), h(k) \oplus_m 1, h(k) \oplus_m 2, ...$ sequentially until it finds the term k being searched. Otherwise, if it finds an empty position, or if the sequential search reaches position $h(k)$ after running over all other positions, the item being searched does not exist in the hash table.

The pseudocode shown below represents how this method works:

1. Calculate $i = h(k)$.

2. If the i-th position is empty or $h(k)$ is reached again after running over all table positions, then the search is concluded and the item relative to k is not in the hash table.

3. If the i-th position contains the item with key k, then the search is concluded and the item relative to k is in position i.

4. Else, $i = i \oplus_m 1$. Go to step 2.

The efficiency of a search for a given key $k \in S$ in the linear hashing method depends on the number of probes performed during the search. This is highly sensitive to the hash table load factor $\alpha = n/m$ (i.e., the ratio between the number of items and the number of entries in the hash table.) The higher is α, the larger is the number of probes. According to Knuth [51], the expected number of probes performed for successful and unsuccessful searches are $\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$ and $\frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right)$, respectively. The main problem with this method is that it degenerates in a sequential search when the number of terms n gets closer to the table size m, which causes a waste of time. Another issue is the waste of space caused by empty positions in the hash table.

7.1.2 Quadratic Hashing

Quadratic hashing is very similar to linear hashing, however, it uses two additional parameters, r and q, besides the hash function $h(k) : S \rightarrow [0, m-1]$. Parameter r indicates how many positions ahead the current position the next search for the term k will be performed, and parameter q indicates the value which parameter r will be added to after each iteration. Quadratic hashing is expected to have a better performance when compared to linear hashing for higher load factors, since it prevents the production of clusters which delay the search for items. However, this method shares some problems found in linear hashing, e.g., the waste of space due to empty positions and the waste of time due to successive collisions when n gets closer to m [46, 51]. The quadratic hashing method may also have a smaller locality of reference when compared to linear hashing, as the pace r may become much larger than one.

The period of search is defined as the number of entries that appear in a sequence from a particular initial position before an entry is encountered twice. The period of search should preferably be the same as the table size m or, at least, as large as possible. Otherwise, the table may appear to be full when there is still space available. If m is a prime number then the period of search for the quadratic hash method is $m/2$.

The pseudocode shown below represents how this method works:

1. Calculate $i = h(k)$.

2. If the i-th position is empty or $h(k)$ is reached again after running over all reachable positions, then the search is concluded and the item relative to k is not in the hash table.

3. If the i-th position contains the item with key k, then the search is concluded and the item relative to k is in position i.

4. Else, $i = i \oplus_m r$, $r = r \oplus_m q$. Go to step 2.

Given a hash table load factor $\alpha = n/m$, the expected number of probes in quadratic hashing is $1 - \ln(1-\alpha) - \frac{\alpha}{2}$ for successful searches and $\frac{1}{1-\alpha} - \ln(1-\alpha) - \alpha$ for unsuccessful searches, according to [51]. Furthermore, in [51] it was proposed a variation of quadratic hashing, which was also compared with perfect hashing in our experiments. We used an implementation available in [72], which is called dense hashing.

7.1.3 Double Hashing

Double hashing also works in a way very similar to linear hashing, but with the difference that, instead of one function, it uses two: $h_1(k)$ and $h_2(k)$. The first one produces values in the range $[0, m-1]$, mapping the term into its position in the hash table, the same way as the hash function in linear hashing does. The additional function $h_2(k)$ produces values in the range $[1, m-1]$, which are used as steps in the process of finding empty positions. Values produced by $h_2(k)$ are relatively primes to the table size m. This is necessary to ensure that the period of search will be of the same as m, which guarantees that any given item can be inserted in any table position (see, e.g., [51]). Furthermore, we can check if the table is full by counting the number of collisions, since m successive collisions indicates a full structure.

This method tests positions using a distance $h_2(k)$, i.e., it tests positions $h_1(k), h_1(k) \oplus_m h_2(k), h_1(k) \oplus_m 2h_2(k), ...$, until it finds an empty position or until it finds the term k being searched for.

The method is described bellow:

1. Calculate $i = h_1(k)$, $d = h_2(k)$.
2. If the i-th position is empty or $h_1(k)$ is reached again after running over all table positions, then the search is concluded and the item relative to k is not in the table.
3. If the i-th position contains the item with key k, then the search is concluded and the item relative to k is in position i.
4. Else, $i = i \oplus_m d$. Go to step 2.

Double hashing reduces the problem of clustering in a better way than quadratic hashing does. This is because function $h_2(k)$ provides a different step d for each key k, and the multiple step sizes produce a more uniform distribution of used positions. This method still shares some problems with previously cited methods, such as the waste of space due to unused positions and the possibility of successive collisions when the structure is almost full. Knuth [51] estimated the expected number of successful probes in searches for double hashing as $-\left(\frac{1}{\alpha}\ln(1-\alpha)\right)$, and the number of unsuccessful probes in searches as $\frac{1}{1-\alpha}$.

7.1.4 Cuckoo Hashing

Cuckoo hashing uses two hash functions, $h_1(k)$ and $h_2(k)$, to get two possible table positions for a given term. When a term x has to be inserted in the structure, one of the two possible positions ($h_1(x)$ or $h_2(x)$) is chosen. If the chosen position is already occupied, the term y contained there will be removed from the structure, yielding an empty position to the term

x being inserted. Term y, in turn, has two possible positions, given by $h_1(y)$ and $h_2(y)$. Consequently, y can be inserted in a position different from its former one. However, that position can be occupied too. Thus, this process must continue until all terms are inserted in one of their possible positions, or until some item can not be inserted [77, 63].

In case we need to search for a term k, the two possible positions for k (namely $h_1(k)$ and $h_2(k)$) are checked. If neither one contains the term, then it is not in the structure. Insertion in cuckoo hashing is better described bellow:

1. Calculate $i = h_1(k)$
2. If the i-th position is empty, insert the term k in that position
3. Else,

    Swap the term k with the term x contained in the i-th position

    If $h_1(x) == i$, then $i = h_2(x)$

    Else, $i = h_1(x)$

    Go to step 2

A problem with this method is that it is possible that it gets into an infinite loop during the insertion of a term, since it can cause a sequence of items to be expelled indefinitely in a cyclical manner. It was shown that in practical situations with a load factor lower than or equal to 50% this is highly unlikely [63]. However, we may prevent this by allowing only a maximum amount of iterations during term insertion. Notwithstanding, cuckoo hashing still will not be able to insert the term with the same hash function values, and the table needs to be rebuilt with different functions if the term is to be inserted.

7.1.5 Sparse Hashing

Sparse hashing is based on a sparse array structure which uses little memory space. It is implemented as an array of groups A, where the number of groups in a sparse array of m entries is calculated as $G = \lceil m/M \rceil$. Each group stored in $A[g]$, $0 \leq g < G$, is responsible for M indexes of the hash table, i.e., $A[0]$ is responsible for the items in the range $[0, M-1]$, $A[1]$ for the items in the range $[M, 2M-1]$, and so on. Each group g contains an array I_g that stores the actual items and a bitmap B_g of size M. The bitmap B_g indicates the assigned indexes in the range $[0, M-1]$. If $B_g[f] = 1$, $0 \leq f < M$, then index f has a corresponding value stored in I_g. Note that an item in group g with an offset f is not necessarily placed in position f of I_g, but in the position $I_g[j]$, where j is the number of bits set from $B_g[0]$ to $B_g[f-1]$. Therefore, the array I_g is dynamically reallocated when new items are inserted in it. Thus, the size of I_g can differ among groups. Figure 7.2 illustrates these data structures.

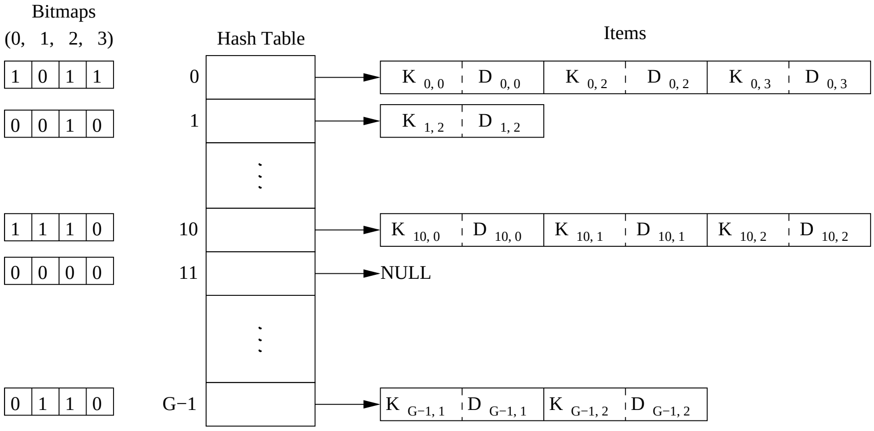

Bitmaps
(0, 1, 2, 3)
Hash Table
Items
1 0 1 1
0 0 1 0
1 1 1 0
0 0 0 0
0 1 1 0
0
1
10
11
G−1
K 0, 0 D 0, 0 K 0, 2 D 0, 2 K 0, 3 D 0, 3
K 1, 2 D 1, 2
K 10, 0 D 10, 0 K 10, 1 D 10, 1 K 10, 2 D 10, 2
NULL
K G−1, 1 D G−1, 1 K G−1, 2 D G−1, 2

Figure 7.2: Hash table used in the sparse hashing method.

A lookup for an item with key k is performed by first calculating its position $i = h(k)$, in which $h(k) : S \rightarrow [0, m-1]$. The group g to which the item belongs is defined as $g = \lfloor i/M \rfloor$, and its offset inside g is $f = i \bmod M$. In this way, we need to check the value of $B_g[f]$. If it is set to 0, then the item is not present in the hash table. Otherwise, it is possibly present in group g and we need to check if there is a collision. This can be done by checking if the item with key k is present in I_g. The position j of the item in this array is calculated by counting the number of bits set between $B_g[0]$ and $B_g[f-1]$. If the item in position j is not the one with key k, then there is a collision, which will be resolved by quadratic probing on i (see Section 7.1.2).

Insertion is performed in a similar fashion. First, we must check if the item is present with a lookup. If not, we shall insert the item in I_g in the position calculated by counting the number of bits set between $B_g[0]$ and $B_g[f-1]$, in the same way it is done in the lookup. An insertion may require the displacement of all items with internal offset j such that $j \geq f$. Let us take Figure 7.2 as an example. Suppose we want to insert a certain item with key k for which $g = 0$ and $f = 1$. Then the item must be inserted in position 1 of group 0, but that position is already occupied. To solve this, we need to move the items with key $K_{0,2}$ and $K_{0,3}$ one position ahead of their current position. The item with $K_{0,3}$ will be moved to the position allocated for the new term, i.e., the forth position. The item with key $K_{0,2}$ will be moved to the position just left of the item with key $K_{0,3}$, i.e., the third position. Finally, the position calculated for the item with key k will be free and we can place the new item there. Figure 7.3 shows the situation of group 0 after the insertion of the item with key $K_{0,1}$.

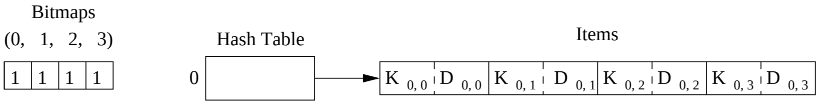


Figure 7.3: Group 0 after an insertion.

This method differs from the others in the sense that it prioritizes efficient memory usage. It allocates as little space as possible to represent unassigned positions, and the arrays containing the actual items grow only when it is needed. If each pointer has a size of p bits, the space overhead of sparse hashing for a hash table of size m and G groups is $m + G \times p$. That is, m bits to represent the bitmaps, and G pointers, one for each group.

Although being very efficient in memory usage, sparse hashing is not designed to be efficient in time: each lookup needs to perform a sequential search through B_g to find the position of an item I_g.

7.1.6 Minimal Perfect Hashing

The hash table structure used by minimal perfect hashing is shown in Figure 7.4. In this structure there is no need for pointers, i.e., all the items are inserted directly in the table. This is only possible because there are no empty entries in the hash table, and therefore we will not lose any space if we increase the capacity of the table entries to fit the items themselves. This is not the case for the other methods, in which any increase in the capacity of the table entries would cause even more space to be wasted. Moreover, the minimal perfect hashing avoids the use of memory space to keep the pointers, which is an additional advantage. However, there is still the need to store the MPHF representation in main memory, and the space overhead for this method is approximately $2.62n$ bits for a set of n keys, as can be seen in Chapter 2.

The minimal perfect hash function $h : S \to [0, n-1]$ used to index the hash table presented in Figure 7.4 is taken from the family of MPHFs proposed in Chapter 2. The MPHFs are generated based on random r-partite hypergraphs where each edge connects $r \geq 2$ vertices (see Definition 13). In our experiments we used a version that employs hypergraphs with $r = 3$, since it generates the fastest and most compact MPHFs. However, for simplicity of exposition, we will now illustrate the MPHF construction when $r = 2$.

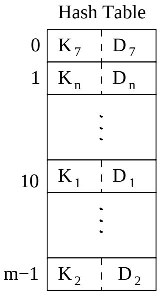


Figure 7.4: Hash table used in the perfect hashing method.

Figure 7.5 gives an overview of the MPHF construction for $r = 2$, taking as input a key set $S \subseteq U$ containing the first four month names abbreviated to the first three letters, i.e., $S = \{\text{jan}, \text{feb}, \text{mar}, \text{apr}\}$. The mapping step in Figure 7.5(a) assumes that it is possible to find $r = 2$ hash functions, h_0 and h_1, with independent values uniformly distributed in the intervals [0,3] and [4,7], respectively. These functions are used to assign each key in S to an edge of an acyclic random bipartite graph $G = (V, E)$[1], such that $|V| = m = 8$ and $|E| = n = 4$. In our example, January is mapped to edge $\{h_0(jan), h_1(jan)\} = \{2, 5\}$, February is mapped to $\{h_0(jan), h_1(jan)\} = \{2, 6\}$, and so on.

The assigning step in Figure 7.5(b) builds an array g representing a function $g : [0, m - 1] \to \{0, 1, 2\}$, which is used to uniquely assign an edge with key k to one of its $r = 2$ incident vertices. The value r is used to represent unassigned vertices. Note that a vertex for a key k is either given by $h_0(k)$ or $h_1(k)$. The decision of which function $h_i(k)$ to be used for k is made by calculating $i = (g[h_0(k)] + g[h_1(k)]) \bmod 2$. In our example, January is mapped to 2 because $(g[2] + g[5]) \bmod 2 = 0$ and $h_0(jan) = 2$. Similarly, February is mapped to 6 because $(g[2] + g[6]) \bmod 2 = 1$ and $h_1(feb) = 6$, and so on.

The ranking step builds a data structure used to compute a function $\mathit{rank}(v)$, which counts in $O(1)$ time the number of assigned positions in g before a given position $v \in [0, m - 1]$. To illustrate, $\mathit{rank}(7) = 3$ means that there are three positions assigned before position 7 in g, namely $0, 2$ and 6.

In our experiments, the MPHF is constructed based on hypergraphs with $r = 3$, and we use three hash functions $h_i : S \to [i\frac{m}{3}, (i + 1)\frac{m}{3} - 1]$, in which $0 \leq i < 3$ and $m = 1.23n$. The value $1.23n$ is required to generate an acyclic random 3-partite hypergraph with high probability, as shown in Chapter 2. Here again, the functions are assumed to have independent values uniformly distributed. The MPHF has the following form: $h(k) = \mathit{rank}(\mathit{phf}(k))$, where $\mathit{phf} : S \to [0, 1.23n - 1]$ is a perfect hash function defined

[1] See Chapter 2 for details on how to obtain such a graph with high probability.

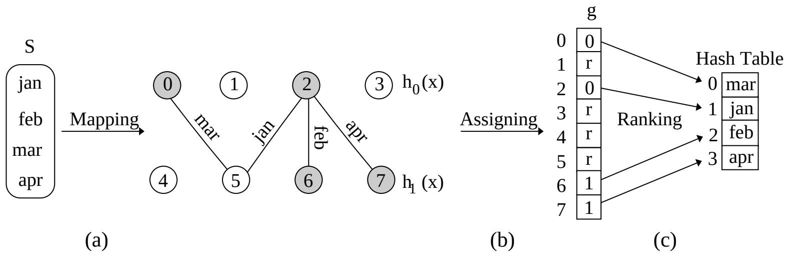


Figure 7.5: (a) The mapping step generates an acyclic bipartite random 2-graph. (b) The assigning step builds an array g so that each edge is uniquely assigned to a vertex. (c) The ranking step builds the data structure used to compute function $rank : V \to [0, n-1]$ in $O(1)$ time.

as $phf(k) = h_i(k)$ and $i = (g[h_0(k)] + g[h_1(k)] + g[h_2(k)]) \bmod 3$. The array g is now representing a function $g : V \to \{0, 1, 2, 3\}$, and $rank : V \to [0, n-1]$ is now the cardinality of $\{u \in V \mid u < v \wedge g[u] \neq 3\}$. Notice that a vertex u is assigned if $g[u] \neq 3$.

7.2 Experimental Results

In this section we present the key sets used in the experiments and the results of the comparative study. All experiments were carried out on a computer running Linux version 2.6, with a 1.86 gigahertz Intel Core 2 64 bits processor, 4 gigabytes of main memory and 4 megabytes of L2 cache. All results presented are averages on 50 trials and were statistically validated with a confidence level of 95%. Table 7.1 summarizes the symbols and acronyms used throughout this section.

The linear hashing, quadratic hashing, double hashing, cuckoo hashing and minimal perfect hashing structures were all implemented using the C language. We used the CMPH library available at `http://cmph.sf.net` to generate the MPHFs used in the minimal perfect hashing structure. For sparse hashing and dense hashing we used the original implementation available in [72].

It is important to notice that we are interested in the performance of lookups and therefore we do not present results concerning the time to build the data structures. Nevertheless, it is important to stress that the MPHF construction is very fast, as can be seen in Chapter 2. We consider two situations: (i) when only successful lookups are performed (i.e., the key is always found in the hash table) and (ii) when only unsuccessful lookups are involved (i.e., a key is never found in the hash table). The results are evaluated for

Symbol	Meaning
α	Load factor
n	Number of keys in a key set
N	Number of keys used in the lookup step
Probes/N	Average number of probes per key during the lookup
T(s)	Average time (in seconds) spent during the lookup of N keys
$\mathcal{S}_o$(bits/key)	Space Overhead in bits per key
LH	Linear Hashing
QH	Quadratic Hashing
DH	Double Hashing
CH	Cuckoo Hashing
SH	Sparse Hashing
DeH	Dense Hashing
MPH	Minimal Perfect Hashing

Table 7.1: Symbols and acronyms used throughout this section.

each data structure in terms of the average number of lookups, the average lookup time and the space overhead.

The experimental results are presented in three distinct subsections. First, in Section 7.2.2, we compare the minimal perfect hashing structure with linear hashing, quadratic hashing and double hashing structures. Second, in Section 7.2.3, we compare it with sparse hashing and dense hashing structures. Finally, in Section 7.2.4, we compare it with cuckoo hashing structure. The three sets of experiments use the key sets described in Section 7.2.1.

7.2.1 Key Sets

In our experiments we used three key sets: (i) a key set of $5,424,923$ unique query terms extracted from the AllTheWeb[2] query log, referred to as AllTheWeb key set; (ii) a key set of $37,294,116$ unique URLs collected from the Brazilian Web by the TodoBr[3] search engine, referred to as URLs-37 key set; and (iii) a smaller key set of 10 million URLs randomly selected from the URLs-37 key set, which is referred to as URLs-10 key set. Table 7.2 shows the main characteristics of each key set, namely the smallest key size, the largest key size and the average key size in bytes.

[2] AllTheWeb (`www.alltheweb.com`) is a trademark of Fast Search & Transfer company, which was acquired by Overture Inc. in February 2003. In March 2004 Overture itself was taken over by Yahoo!.

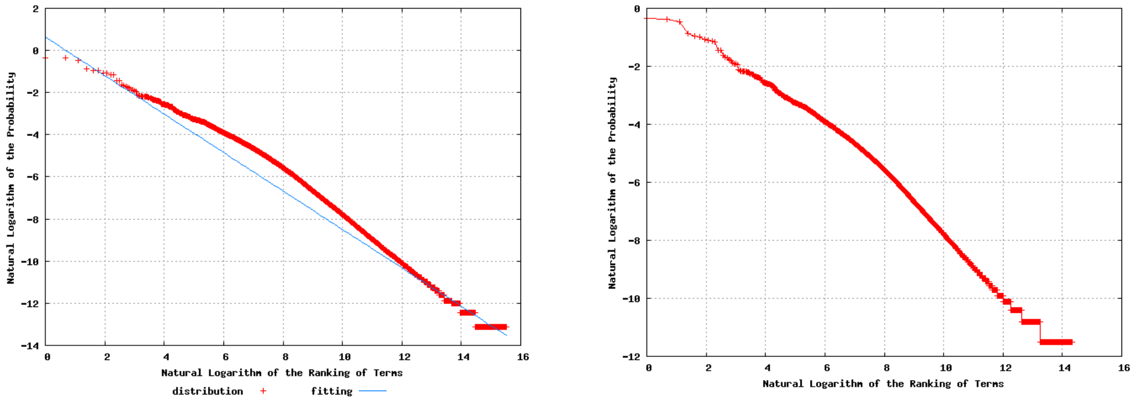
[3] TodoBr (www.todobr.com.br) is a trademark of Akwan Information Technologies, which was acquired by Google Inc. in July 2005.

Key Set	n	Shortest Key	Largest Key	Average Size of the Keys
AllTheWeb	5,424,923	2	31	17.46
URLs-10	10,000,000	8	494	58.36
URLs-37	37,294,116	8	496	58.77

Table 7.2: Characteristics of the key sets used for the experiments.

In order to test the lookup performance of the considered hash structures in a real world environment, we need to look up keys in a way similar to the real access patterns of actual applications. In the case of the AllTheWeb key set, the probability distribution of query term lookups was extracted from the AllTheWeb query log. Similarly, the distribution of URL lookups must be equivalent to the access pattern performed by a web crawler that needs to check whether a URL extracted from a web page is new, i.e., whether it has not been collected before. Therefore, we decided to use an automatic generator to simulate these lookup patterns found in search engines.

The probability distribution of query term lookups for the AllTheWeb key set is shown in Figure 7.6 (a). It is plotted in a log-log scale, constituting a power law distribution with inclination -0.91. This same distribution was used to simulate the lookup stream submitted to the hashing data structures in order to evaluate their performance, as can be seen in Figure 7.6 (b). We generated 10 million keys to be looked up in a hashing data structure storing the AllTheWeb key set.



(a) Extracted from AllTheWeb query log.

(b) Generated automatically.

Figure 7.6: Probability distribution of query term lookups.

Pages arriving in a crawling system are known to have a few very popular URLs and many not so popular URLs, which also constitutes a power law behavior [17]. Consequently, we employed the same distribution found for query terms to describe the probability of arrival of a URL in a crawler. We generated 250 million and 20 million URLs to be looked up in the hashing data structures that store the URLs-37 key set and the URLs-10 key set, respectively.

So far we have described how to generate key sets to perform successful searches in hashing data structures. In order to test the performance of the data structures when unsuccessful searches are involved, we have randomly generated three additional key sets: (i) 10 million keys of average size equal to 17.46 bytes to be looked up when the structures are storing the AlltheWeb key set, (ii) 20 million keys of average size equal to 58.36 bytes to be looked up when the structures are storing the URLs-10 key set, and (iii) 250 million keys of average size equal to 58.77 bytes to be looked up when the structures are storing the URLs-37 key set. They were created based on the average key sizes presented in Table 7.2.

In our experiments we used an 8-byte fingerprint of the key instead of the key itself. The use of fingerprints was motivated by two reasons: (i) to guarantee that all keys have the same size, since in this way we can allocate a fixed size for each key without waste of space; and (ii) to reduce the amount of memory used to store each key, as the average key size in all key sets used is greater than 8 bytes. A point worth noting is that each key set was stored entirely in main memory, but the set of automatically generated keys is too big to be stored in the same way, and had to be kept in disk.

7.2.2 Minimal Perfect Hashing Versus Linear Hashing, Quadratic Hashing and Double Hashing

In this section we compare the minimal perfect hashing structure with linear hashing, quadratic hashing and double hashing. Linear hashing, quadratic hashing and double hashing methods were tested with different load factors, ranging from 50 to 90%. We considered both successful and unsuccessful searches to measure the average number of probes and the amount of time spent (on average) to look up 10, 20 and 250 million keys in the AllTheWeb, URLs-10 and URLs-37 key sets, respectively.

The results for successful and unsuccessful searches are presented in Tables 7.3 and 7.4, respectively. As expected, quadratic hashing and double hashing perform better than linear hashing for high load factors, since they avoid the creation of clusters in this case. Nevertheless, linear hashing is a better option when we use lower load factors. Furthermore, we can see that double hashing always has a smaller number of collisions per key when compared to quadratic hashing and linear hashing, but it is slower since it needs to compute two hash functions instead of one. The average number of probes measured for both successful and unsuccessful searches are very close to the expected according to the equations presented in Sections 7.1.1, 7.1.2 and 7.1.3 (this is not shown in the tables).

		LH		QH		DH	
Key Set	α	Probes/N	T(s)	Probes/N	T(s)	Probes/N	T(s)
AllTheWeb	85%	3.78	5.67	2.40	5.27	2.17	5.42
	80%	2.91	5.26	2.09	5.09	2.02	5.28
	75%	2.47	5.04	1.93	4.97	1.90	5.11
	70%	2.13	4.84	1.78	4.81	1.71	4.98
	65%	1.89	4.70	1.69	4.69	1.61	4.83
	60%	1.73	4.58	1.58	4.60	1.52	4.72
	55%	1.62	4.46	1.51	4.52	1.45	4.64
	50%	1.48	4.34	1.42	4.40	1.40	4.56
URLs-10	85%	3.63	18.98	2.27	17.87	2.16	18.36
	80%	2.83	18.32	2.09	17.67	1.96	18.02
	75%	2.37	17.69	1.87	17.29	1.83	17.69
	70%	2.05	17.31	1.76	17.01	1.69	17.34
	65%	1.80	17.00	1.61	16.81	1.62	17.14
	60%	1.70	16.84	1.53	16.61	1.50	16.92
	55%	1.57	16.34	1.47	16.33	1.42	16.58
	50%	1.51	16.33	1.39	16.19	1.35	16.39
URLs-37	85%	3.94	269.19	2.37	253.18	2.29	263.80
	80%	3.00	255.53	2.12	247.48	2.01	257.31
	75%	2.46	247.95	1.89	242.51	1.83	250.60
	70%	2.11	243.02	1.82	240.55	1.71	246.58
	65%	1.94	238.15	1.65	235.54	1.66	244.10
	60%	1.74	235.21	1.57	234.20	1.50	239.84
	55%	1.62	232.83	1.50	231.63	1.45	236.31
	50%	1.55	229.62	1.43	228.92	1.37	233.79

Table 7.3: Load factor influence on the time to successfully look up 10, 20 and 250 million keys in the AllTheWeb, URLs-10 and URLs-37 key sets, respectively.

We now compare the minimal perfect hashing structure with linear hashing, quadratic hashing and double hashing. Tables 7.5 and 7.6 show the results for successful and unsuccessful searches, respectively. Two interesting results are remarkable. First, when the MPHF description fits in the L2 cache, which is the case for the AllTheWeb key set and URLs-10 key set, the minimal perfect hashing structure outperforms the others in terms of lookup time for load factors greater than 55% for both successful and unsuccessful searches. Second, in the converse situation in which the MPHF description does not fit in the L2

		LH		QH		DH	
Key Set	α	Probes/N	T(s)	Probes/N	T(s)	Probes/N	T(s)
AllTheWeb	85%	22.80	14.82	7.54	8.60	6.67	8.89
	80%	13.02	10.43	5.59	7.36	5.00	7.65
	75%	8.44	8.17	4.43	6.67	4.00	6.95
	70%	6.05	7.03	3.66	6.20	3.33	6.42
	65%	4.59	6.32	3.11	5.87	2.86	6.06
	60%	3.63	5.84	2.71	5.60	2.50	5.74
	55%	2.97	5.48	2.39	5.36	2.22	5.48
	50%	2.50	5.19	2.13	5.14	2.00	5.25
URLs-10	85%	22.61	34.81	7.54	22.68	7.25	23.71
	80%	12.93	25.78	5.59	20.16	5.00	20.87
	75%	8.49	21.93	4.43	18.77	4.00	19.27
	70%	6.05	19.42	3.66	17.77	3.33	18.18
	65%	4.58	17.94	3.11	17.07	2.86	17.40
	60%	3.62	16.91	2.70	16.57	2.50	16.70
	55%	2.97	16.14	2.39	15.98	2.22	16.14
	50%	2.50	15.59	2.13	15.57	2.00	15.63
URLs-37	85%	22.53	526.05	7.55	333.49	6.67	379.17
	80%	13.01	387.93	5.59	294.89	5.19	330.74
	75%	8.51	318.94	4.43	270.53	4.00	296.62
	70%	6.06	281.93	3.66	253.64	3.33	274.55
	65%	4.58	258.15	3.12	242.66	2.86	257.75
	60%	3.62	242.04	2.71	232.46	2.50	245.06
	55%	2.97	230.90	2.39	225.05	2.22	233.96
	50%	2.50	220.64	2.13	217.66	2.00	222.92

Table 7.4: Load factor influence on the time to unsuccessfully look up 10, 20 and 250 million keys in the AllTheWeb, URLs-10 and URLs-37 key sets, respectively.

cache, which is the case for the URLs-37 key set, the same thing happens for load factors greater than 75% and 65% for successful searches and unsuccessful searches, respectively. Therefore, as can be seen, the use of MPHFs saves a significant amount of space with almost no loss in the lookup time.

Data	AllTheWeb			URLs-10			URLs-37		
Structure	α(%)	T(s)	$\mathcal{S}_o$(bits/key)	α(%)	T(s)	$\mathcal{S}_o$(bits/key)	α(%)	T(s)	$\mathcal{S}_o$(bits/key)
MPH	100	4.48	2.62	100	16.34	2.62	100	250.36	2.62
LH	55	4.46	116.36	55	16.34	116.36	75	247.95	85.33
QH	55	4.52	116.36	55	16.33	116.36	80	247.48	80
DH	50	4.56	128	50	16.39	128	75	250.60	85.33

Table 7.5: Comparison of MPH with LH, QH and DH, considering the space overhead and the time to successfully look up 10, 20 and 250 million keys in the AllTheWeb, URLs-10 and URLs-37 key sets, respectively.

Data	AllTheWeb			URLs-10			URLs-37		
Structure	α(%)	T(s)	$\mathcal{S}_o$(bits/key)	α(%)	T(s)	$\mathcal{S}_o$(bits/key)	α(%)	T(s)	$\mathcal{S}_o$(bits/key)
MPH	100	5.33	2.62	100	16.38	2.62	100	252.65	2.62
LH	55	5.48	116.36	60	16.91	106.67	65	258.15	98.46
QH	55	5.36	116.36	60	16.57	106.67	70	253.64	91.43
DH	55	5.48	116.36	60	16.70	106.67	65	257.75	98.46

Table 7.6: Comparison of MPH with LH, QH and DH, considering the space overhead and the time to unsuccessfully look up 10, 20 and 250 million keys in the AllTheWeb, URLs-10 and URLs-37 key sets, respectively.

7.2.3 Minimal Perfect Hashing Versus Dense and Sparse Hashing

Sparse hashing and dense hashing were tested with their default load factor only, which is 80%. Table 7.7 shows the time spent to execute the lookup step for each method for successful searches only. As expected, sparse hashing had the worst performance in lookup time when compared to the other methods, as it is designed to be efficient in space but not in execution time. The same is true for unsuccessful searches, as displayed in Table 7.8. It is important to note that perfect hashing has clearly outperformed the other methods in all aspects. Experiments were performed using only the AllTheWeb and URLs-10 key sets. We decided not to use the URLs-37 key set since we did not expect any improvements on the results.

Data	AllTheWeb			URLs-10		
Structure	α(%)	T(s)	$\mathcal{S}_o$(bits/key)	α(%)	T(s)	$\mathcal{S}_o$(bits/key)
MPH	100	4.48	2.62	100	16.34	2.62
SH	80	11.47	2,92	80	35.76	2,92
DeH	80	6.51	80	80	27.48	80

Table 7.7: Comparison of MPH with DeH and SH, considering the space overhead and the time to successfully look up 10 and 20 million keys in the AllTheWeb and URLs-10 key sets, respectively.

Data	AllTheWeb			URLs-10		
Structure	α(%)	T(s)	$\mathcal{S}_o$(bits/key)	α(%)	T(s)	$\mathcal{S}_o$(bits/key)
MPH	100	5.33	2.62	100	16.38	2.62
SH	80	15.48	2,92	80	48.27	2,92
DeH	80	7.59	80	80	30.26	80

Table 7.8: Comparison of MPH with DeH and SH, considering the space overhead and the time to unsuccessfully look up 10 and 20 million keys in the AllTheWeb and URLs-10 key sets, respectively.

7.2.4 Minimal Perfect Hashing Versus Cuckoo Hashing

Cuckoo hashing has a different behavior when compared to any of the methods analyzed, as it cannot work if the load factor is high, i.e., at most 50% [63]. Therefore, we decided to show the comparison between this method and perfect hashing in this separated subsection. Cuckoo hashing was tested with load factors ranging from 20% to the maximum load factor with which it works.

Table 7.9 shows the average number of probes and the average lookup time to successfully search for 10, 20 and 250 million keys in the AllTheWeb, URLs-10 and URLs-37 key sets, respectively. We can see that cuckoo hashing performs slightly faster for all key sets used, but the space overhead for the MPH structure is much lower than for cuckoo hashing in all experiments. The same happens for unsuccessful searches, as we can see in Table 7.10.

Data	AllTheWeb			URLs-10			URLs-37		
Structure	α(%)	T(s)	$\mathcal{S}_o$(bits/key)	α(%)	T(s)	$\mathcal{S}_o$(bits/key)	α(%)	T(s)	$\mathcal{S}_o$(bits/key)
MPH	100	4.48	2.62	100	16.34	2.62	100	250.36	2.62
CH	20	4.08	320	20	15.99	320	20	222.40	320
CH	30	4.13	213	30	16.05	213	30	224.98	213
CH	40	4.28	160	40	16.22	160	40	228.76	160
CH	50	4.38	128	50	16.34	128	50	233.89	128

Table 7.9: Comparison of MPH with CH, considering the space overhead and the time to successfully look up 10, 20 and 250 million keys in the AllTheWeb, URLs-10 and URLs-37 key sets, respectively.

Data	AllTheWeb			URLs-10			URLs-37		
Structure	α(%)	T(s)	$\mathcal{S}_o$(bits/key)	α(%)	T(s)	$\mathcal{S}_o$(bits/key)	α(%)	T(s)	$\mathcal{S}_o$(bits/key)
MPH	100	5.33	2.62	100	16.38	2.62	100	252.65	2.62
CH	20	5.06	320	20	15.79	320	20	222.46	320
CH	30	5.10	213	30	15.92	213	30	227.21	213
CH	40	5.30	160	40	16.07	160	40	229.58	160
CH	50	5.34	128	50	16.17	128	50	231.26	128

Table 7.10: Comparison of MPH with CH, considering the space overhead and the time to unsuccessfully look up 10, 20 and 250 million keys in the AllTheWeb, URLs-10 and URLs-37 key sets, respectively.

7.3 Conclusions

In this chapter we have presented a thorough study of data structures that are suitable for indexing internal memory in an efficient way in terms of both space and lookup time when we have a key set that is fixed for a long period of time (i.e., a static key set) and each key is associated with satellite data. This is widely used in data warehousing and search engine applications (see [71] for other examples).

It is well known that an efficient way to represent a key set in terms of lookup time is by using a table indexed by a hash function. For static key sets it is possible to pay the price of pre-computing an MPHF to find any key in a table in one single probe. We have shown that minimal perfect hashing has a clear advantage in memory usage when compared to other hashing methods, since there are no empty entries in its hash table and thus space overhead is greatly reduced by avoiding the use of pointers. This implies in a gain of $O(\log n)$ bits.

In our study, we compared MPHFs with linear hashing, quadratic hashing, double hashing, dense hashing, cuckoo hashing and sparse hashing. We have shown that MPHFs provide the best tradeoff between space usage and lookup time among these hashing schemes. As an example, minimal perfect hashing have a better performance in all measured aspects when compared to sparse hashing, which has been designed specifically for efficient memory usage. Furthermore, if the MPHF can be stored in cache, minimal perfect hashing outperforms linear hashing, quadratic hashing and double hashing in all aspects when these methods have a hash table occupancy of 55% or higher. The same happens for hash table occupancies greater than or equal to 75% if the MPHF does not fit in cache. This implies in a significant memory overhead due to a great number of unused positions in the hash table.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this work we have presented two classes of algorithms for constructing PHFs and MPHFs. The first class contains internal memory based algorithms that assume uniform hashing to construct the functions. The algorithms read a key set stored in external memory and maps it to data structures that are handled in the internal memory. Then, the generation of the functions are done based on these internal data structures. The second class contains a cache-aware external memory algorithm that generates the functions without assuming uniform hashing. The algorithm uses data structures stored in both internal and external memory, but the key set is still kept in the external memory.

In Chapter 2 we presented an internal random access memory algorithm (RAM algorithm) that generates a family $\mathcal{F}$ of near space-optimal PHFs or MPHFs. The RAM algorithm uses acyclic random hypergraphs given by function values of r uniform random hash functions on S for generating PHFs and MPHFs that require $O(n)$ bits to be stored. We have improved in a factor of $O(\log n)$ the well known result by Majewski et al [55]. They generate MPHFs based on acyclic hypergraphs that are stored in $O(n \log n)$ bits whereas the ones generated by the RAM algorithm requires $O(n)$ bits. All the resulting functions are evaluated in constant time. For $r = 2$ the resulting MPHFs are stored in approximately $3.6n$ bits. For $r = 3$ we have got still more compact MPHFs, which are stored in approximately $2.6n$ bits. This is within a factor of 2 from the information theoretical lower bound of approximately $1.44n$ bits for MPHFs.

For applications where a PHF of range $[0, m-1]$, where $m = 1.23n$, is sufficient, more compact and even simpler representations can be achieved. For example, for $m =$

$1.23n$ we can get a space usage of $1.95n$ bits. This is also within a small constant factor from the information theoretical lower bound of approximately $0.89n$ bits. The bounds for $r = 3$ assume a conjecture about the emergence of a 2-core in a random 3-partite hypergraph, whereas the bounds for $r = 2$ are fully proved. Choosing $r > 3$ does not give any improvement of these results.

There is a gap between theory and practice among all previous methods on perfect hashing. On one hand, there are good theoretical results without experimentally proven practicality for large key sets. On the other hand, there are the algorithms that assume unrealistic assumptions, as the assumption that uniform hash functions are available to be used with no extra cost of storage space (see Section 1.4), to theoretically analyze their run time and space usage. The methods also require $O(n)$ computer words for the construction process.

To design a cache-aware external memory algorithm (EM algorithm) that gives an important step on the way of bridging the gap between theory and practice on perfect hashing, works with high probability for every key set and scales for key sets on the order of billions of keys we used a two-step approach: the *partitioning step* and the *searching step*. In the partitioning step, we use a universal hash function to split the incoming key set S into small buckets with a bounded maximum bucket size that fits in the CPU cache. Then, we use the techniques presented in Chapter 3 to simulate uniform hash functions on the small buckets. Therefore, as we are able to use uniform hash functions on the small buckets, in the searching step we use the RAM algorithm to build an MPHF for each bucket with high probability, and using an *offset* array we obtain an MPHF for the whole key set S. In order to scale for sets on the order of billions of keys we generate runs of an external multi-way merge sort during the partitioning step and merge them in the searching step when the buckets are read from disk.

The EM algorithm requires $O(n^\epsilon)$ computer words, where $0 < \epsilon < 1$, for constructing the functions in linear time. Typically $\epsilon = 0.5$ and that is the main reason that makes the EM algorithm to scale. The resulting PHFs and MPHFs require approximately 2.7 and 3.3 bits per key to be stored and are evaluated in constant time. All together makes the EM algorithm the first one that demonstrates the capability of generating MPHFs for sets on the order of billions of keys on a commodity PC. For instance, considering a set of 1.024 billion URLs the EM algorithm constructs an MPHF on a commodity PC in approximately 50 minutes. The complete description of the EM algorithm is presented in Chapter 4.

The EM algorithm presents a high degree of parallelism to be exploited. Then, in

Chapter 5, we presented a parallel implementation of the EM algorithm (PEM algorithm). The PEM algorithm distributes both the construction and the description of the resulting functions. For instance, by using a 14-computer cluster the PEM algorithm generates a PHF or an MPHF for 1.024 billion URLs in approximately 4 minutes, achieving an almost linear speedup. Also, for 14.336 billion 16-byte random integers evenly distributed among the 14 participating machines the PEM algorithm outputs a PHF or an MPHF in approximately 50 minutes, resulting in a performance degradation of 20%.

In Chapter 6, we designed techniques to generate MPHFs based on random graphs with cycles. The BKZ algorithm was the first algorithm we came up with to generate MPHFs based on random graphs with cycles. It improves the space requirement of the algorithm by Czech, Havas and Majewski [25], referred to as CHM, at the expense of generating functions in the same form that are not order preserving, but are computed in $O(1)$ time. We have improved the space required to store a function in the BKZ algorithm to 55% of the space required by the CHM algorithm. The BKZ algorithm is also linear on n and runs 59% faster than the CHM algorithm. However, the resulting MPHFs still need $O(n \log n)$ bits to be stored and the algorithm needs $O(n)$ computer words to construct the functions.

In the same trend, also in Chapter 6, we used techniques similar to the ones used in the design of the BKZ algorithm to speedup the execution time of RAM algorithm that works on random acyclic bipartite graphs. In this case, by allowing a single cycle with the same number of vertices and edges per connected component in the random bipartite graph we were able to generate PHFs and MPHFs 40% faster than when cycles are not allowed.

Minimal perfect hash functions were not considered a good option to index internal memory in the past [45]. However, in Chapter 7, we showed that the new MPHFs proposed in this work, specially the ones generated by the RAM algorithm, have a clear advantage in memory usage when compared to other hashing methods with almost no loss in terms of lookup time, since there are no empty entries in its hash table and thus space overhead is greatly reduced by avoiding the use of pointers. This implies in a gain of $O(\log n)$ bits.

8.2 Future Work

In this work we designed, analyzed and implemented algorithms to build compact and practical PHFs and MPHFs. On the way we left some points open to be exploited as future work. In the following we present the future steps to be taken:

1. In Chapter 2 the threshold for the moment that the random acyclic r-partite hyper-

graphs dominate the space of random r-partite hypergraphs have not been completely proved for $r \geq 3$. The problems for $r < 3$ and for $r \geq 3$ have different natures and involve a phase transition, as reported to us by Kohayakawa [52]. We are on the way of obtaining a fully proof of the threshold for $r \geq 3$. This is being done in a joint work with Professor Nicholas C. Wormald from the Department of Combinatorics and Optimization at University of Waterloo.

2. The main technical ingredient of the family of algorithms presented in Chapter 2 is the use of acyclic r-partite random hypergraphs. We have shown how to deal with cycles when $r = 2$. However, we aim to extend the techniques presented in Chapter 6 to deal with cycles when $r = 3$. We believe that we can get still more compact functions in this case.

3. In Chapter 6 we were not able to prove the Conjecture 1 and Professor Jayme Szwarcfiter pointed out in the thesis presentation that the literature related to graceful labeling can be a good source to find insights on the way of the proof. Therefore, we want to study this literature and try to find a proof for the the Conjecture 1.

4. A problem with all algorithms we have designed is that we need to know the key set a priori. That is, they are designed to work with static sets. Then, we aim to study how to extend the algorithms to work with dynamic key sets to build compact dynamic minimal perfect hash functions. In this case, keys can be inserted or removed from the key set and this operations would be carried out in our methods with a linear cost. Then, our objective is to look for algorithms that generate functions as compact as possible, which should allow lookups, insertions and deletions in amortized constant time.

5. We believe that MPHFs can potentially be applied to applications where we need to index similar objects previously clustered with respect to some similarity metric (e.g., Euclidean distance). In these cases, we can compute a key for each cluster based on the similarity metric and, then, to compute an MPHF for the resulting key set. We aim to exploit this problem because we believe that the resulting key sets can be built in such way that put them in between static and dynamic key sets. For example, this situation would occur if we were able to build the key set so that a key is added to it whenever a new cluster is created and deleted only when a cluster disappear, but no change is made in the key of a given cluster when objects are added to or removed from the cluster.

6. An MPHF can be used to implement a data structure with the same functionality as a Bloom filter[1][60, 37]. In many applications where a set S of elements is to be stored, it is acceptable to include in the set some false positives with a small probability by storing a signature for each perfect hash value. Theoretically speaking, as shown in [60], this data structure requires around 30% less space usage when compared to Bloom filters, plus the space for the MPHF. Then we aim to study if this data structure outperforms the Bloom filters in practice when lookup time and space usage are considered as metrics. Preliminary results indicates that the data structure that uses MPHFs just outperforms the Bloom filters for false positive rates smaller than 2^6 in terms of space usage. We still do not have conclusive preliminary results for lookup time.

[1] The Bloom filter, conceived by Burton H. Bloom in 1970 [6], is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. False positives are possible, but false negatives are not. False positives are elements that appear to be in S but are not and false negatives are elements that are not in S but a data structure storing S says that they are. Elements can be added to the set, but not removed (though this can be addressed with a counting filter [10]). The more elements that are added to the set, the larger the probability of false positives. Bloom filters have applications in distributed databases and data mining (association rule mining [21, 22]).

Bibliography

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] N. Alon, M. Dietzfelbinger, P. B. Miltersen, E. Petrank, and G. Tardos. Linear hash functions. *Journal of the ACM*, 46(5):667–683, 1999.

[3] N. Alon and M. Naor. Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16(4-5):434–449, 1996.

[4] M. Atici, D. R. Stinson, and R. Wei. A new practical algorithm for the construction of a perfect hash function. *Journal Combin. Math. Combin. Comput.*, 35:127–145, 2000.

[5] Djamal Belazzougui. Private communication, September 30, 2006.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[7] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *Proceedings of the 13th International World Wide Web Conference (WWW'04)*, pages 595–602, 2004.

[8] B. Bollobás. *Random graphs*, volume 73 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, second edition, 2001.

[9] B. Bollobás and O. Pikhurko. Integer sets with prescribed pairwise differences being distinct. *European Journal of Combinatorics*. To Appear.

[10] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. In *Proceedings of the 14th conference on Annual European Symposium (ESA'06)*, pages 684–695, London, UK, 2006. Springer-Verlag.

[11] F. C. Botelho, D. Galinkin, W. Meira Jr., and N. Ziviani. Distributed perfect hashing for very large key sets. In *Proceedings of the 3rd International ICST Conference on Scalable Information Systems (InfoScale'08)*. ACM Press, 2008.

[12] F. C. Botelho, Y. Kohayakawa, and N. Ziviani. A practical minimal perfect hashing method. In *Proceedings of the 4th International Workshop on Efficient and Experimental Algorithms (WEA'05)*, pages 488–500. Springer LNCS vol. 3503, 2005.

[13] F. C. Botelho, H. R. Langbehn, G. V. Menezes, and N. Ziviani. Indexing internal memory with minimal perfect hash functions. In *Proceedings of the 23rd Brazilian Symposium on Database (SBBD'08)*, October 2008.

[14] F. C. Botelho, R. Pagh, and N. Ziviani. Simple and space-efficient minimal perfect hash functions. In *Proceedings of the 10th Workshop on Algorithms and Data Structures (WADS'07)*, pages 139–150. Springer LNCS vol. 4619, 2007.

[15] F. C. Botelho and N. Ziviani. External perfect hashing for very large key sets. In *Proceedings of the 16th ACM Conference on Information and Knowledge Management (CIKM'07)*, pages 653–662. ACM Press, 2007.

[16] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th International World Wide Web Conference (WWW'98)*, pages 107–117, April 1998.

[17] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications netowrking (WWW'00)*, pages 309–320, Amsterdam, The Netherlands, The Netherlands, 2000. North-Holland Publishing Co.

[18] A. Z. Broder, M. Charikar, A. Frieze, and M. Mitzenmacher. Min-wise independent permutations. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing (STOC'98)*, pages 327–336, 1998.

[19] J. Cain and N. C. Wormald. Encores on cores. *Electronic Journal of Combinatorics*, 13(1), 2006.

[20] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[21] C. C. Chang and C. Y. Lin. A perfect hashing schemes for mining association rules. *The Computer Journal*, 48(2):168–179, 2005.

[22] C. C. Chang, C. Y. Lin, and H. Chou. Perfect hashing schemes for mining traversal patterns. *Journal of Fundamenta Informaticae*, 70(3):185–202, 2006.

[23] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: An efficient data structure for static support lookup tables. In *Proceedings of the 15th annual ACM-SIAM symposium on Discrete algorithms (SODA'04)*, pages 30–39, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[25] Z. J. Czech, G. Havas, and B. S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257–264, 1992.

[26] Z. J. Czech, G. Havas, and B. S. Majewski. Fundamental study perfect hashing. *Theoretical Computer Science*, 182:1–143, 1997.

[27] A. M. Daoud. Perfect hash functions for large web repositories. In G. Kotsis, D. Taniar, S. Bressan, I. K. Ibrahim, and S. Mokhtar, editors, *Proceedings of the 7th International Conference on Information Integration and Web Based Applications Services (iiWAS'05)*, volume 196, pages 1053–1063. Austrian Computer Society, 2005.

[28] E. Demaine, F. Meyer auf der Heide, R. Pagh, and M. Pătraşcu. De dictionariis dynamicis pauco spatio utentibus. In *Proceedings of the Latin American Symposium on Theoretical Informatics (LATIN'06)*, pages 349–361, 2006.

[29] M. Dietzfelbinger and T. Hagerup. Simple minimal perfect hashing in less space. In *Proceedings of the 9th European Symposium on Algorithms (ESA'01)*, pages 109–120. Springer LNCS vol. 2161, 2001.

[30] M. Dietzfelbinger and C. Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. In *Proceedings of 32nd International Colloquium on Automata, Languages and Programming (ICALP'05)*, pages 166–178, 2005.

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

[32] J. Ebert. A versatile data structure for edges oriented graph algorithms. *Communication of The ACM*, (30):513–519, 1987.

[33] P. Erdős and A. Rényi. On random graphs I. *Pub. Math. Debrecen*, 6:290–297, 1959.

[34] P. Erdős and A. Rényi. On the evolution of random graphs. *Magyar Tud. Akad. Mat. Kutató Int. Közl.*, 5:17–61, 1960.

[35] P. Erdős and A. Rényi. On the strength of connectedness of a random graph. *Acta Mathematica Scientia Hungary*, 12:261–267, 1961.

[36] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley, January 1968.

[37] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci. Multilayer compressed counting bloom filters. In *Proceedings of the 27th IEEE Conference on Computer Communications (INFOCOM'08)*, pages 311–315. IEEE Press, 2008.

[38] E. A. Fox, Q. F. Chen, and L. S. Heath. A faster algorithm for constructing minimal perfect hash functions. In *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'92)*, pages 266–273, 1992.

[39] E. A. Fox, L. S. Heath, Q. Chen, and A. M. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1):105–121, 1992.

[40] M. L. Fredman and J. Komlós. On the size of separating systems and families of perfect hashing functions. *SIAM Journal on Algebraic and Discrete Methods*, 5:61–68, 1984.

[41] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with O(1) worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.

[42] N. Galli, B. Seybold, and K. Simon. Tetris-hashing or optimal table compression. *Discrete Applied Mathematics*, 110(1):41–58, june 2001.

[43] T. Hagerup and T. Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Proceedings of the 18th Symposium on Theoretical Aspects of Computer Science (STACS'01)*, pages 317–326. Springer LNCS vol. 2010, 2001.

[44] G. Havas, B. S. Majewski, N. C. Wormald, and Z. J. Czech. Graphs, hypergraphs and hashing. In *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 153–165. Springer LNCS vol. 790, 1993.

[45] Y. Ho. Application of minimal perfect hashing in main memory indexing. Technical report, Cambridge, MA, USA, 1994.

[46] F. R. A. Hopgood and J. Davenport. The quadratic hash method when the table size is a power of 2. *The Computer Journal*, 15(4):314–315, November 1972.

[47] R. Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling.* John Wiley, first edition, 1991.

[48] S. Janson. Poisson convergence and poisson processes with applications to random graphs. *Stochastic Processes and their Applications*, 26:1–30, 1987.

[49] S. Janson, T. Łuczak, and A. Ruciński. *Random graphs.* Wiley-Inter., 2000.

[50] B. Jenkins. Algorithm alley: Hash functions. *Dr. Dobb's Journal of Software Tools*, 22(9), september 1997.

[51] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, second edition, 1973.

[52] Yoshiharu Kohayakawa. Private communication, 2007.

[53] P. Larson and G. Graefe. Memory management during run generation in external sorting. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 472–483. ACM Press, 1998.

[54] S. Lefebvre and H. Hoppe. Perfect spatial hashing. *ACM Transactions on Graphics*, 25(3):579–588, 2006.

[55] B. S. Majewski, N. C. Wormald, G. Havas, and Z. J. Czech. A family of perfect hashing methods. *The Computer Journal*, 39(6):547–554, 1996.

[56] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: Memory access. *The VLDB journal*, 9:231–246, 2000.

[57] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* Springer-Verlag, 1984.

[58] R. Motwani and P. Raghavan. *Randomized algorithms.* Cambridge University Press, New York, NY, USA, 1995.

[59] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, 2007.

[60] A. Pagh, R. Pagh, and S. S. Rao. An optimal bloom filter replacement. In *Proceedings of the 16th annual ACM-SIAM symposium on Discrete algorithms (SODA'05)*, pages 823–829, Philadelphia, PA, USA, 2005.

[61] R. Pagh. Hash and displace: Efficient evaluation of minimal perfect hash functions. In *Workshop on Algorithms and Data Structures (WADS'99)*, pages 49–54, 1999.

[62] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

[63] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[64] E. M. Palmer. *Graphical Evolution: An Introduction to the Theory of Random Graphs.* John Wiley & Sons, New York, 1985.

[65] B. Pittel and N. C. Wormald. Counting connected graphs inside-out. *Journal of Combinatorial Theory Series B*, 93(2):127–172, 2005.

[66] B. Prabhakar and F. Bonomi. Perfect hashing for network applications. In *Proceedings of the IEEE International Symposium on Information Theory.* IEEE Press, 2006.

[67] Michael J. Quinn. *Parallel computing: theory and practice.* McGraw-Hill, Inc., New York, NY, USA, second edition, 1994.

[68] J. Radhakrishnan. Improved bounds for covering complete uniform hypergraphs. *Information Processing Letters*, 41:203–207, 1992.

[69] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA'02)*, pages 233–242, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[70] J. P. Schmidt and A. Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM Journal on Computing*, 19(5):775–786, October 1990.

[71] M. Seltzer. Beyond relational databases. *ACM Queue*, 3(3), April 2005.

[72] C. Silverstein. An extremely memory-efficient hash_map implementation (google-sparsehash). `http://code.google.com/p/google-sparsehash`, November 2007.

[73] D. R. Stinson, R. Wei, and L. Zhu. New constructions for perfect hash families and related structures using combinatorial designs and codes. *Journal Combin. Designs.*, 8:189–200, 2000.

[74] J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, pages 1–38. American Mathematical Society Press, Providence, RI, 1999.

[75] P. Woelfel. Maintaining external memory efficient hash tables. In *Proceedings of the 10th International Workshop on Randomization and Computation (RANDOM'06)*, pages 508–519. Springer LNCS vol. 4110, 2006.

[76] N. Ziviani. *Projeto de Algoritmos com implementações em Java e C++*. Thompson Learning, São Paulo, first edition, 2006. Consultoria em Java e C++ de F. C. Botelho.

[77] M. Zukowski, S. Héman, and P. Boncz. Architecture-conscious hashing. In *Second DAMON workshop (SIGMOD 2006)*, Chicago, USA, june 2006.