Fabiano Cupertino Botelho
Orientador - Nivio Ziviani

Algoritmos de Espaço Quase Ótimo Para Hashing Perfeito

Proposta de tese de doutorado apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de doutor em Ciência da Computação.

Belo Horizonte
January 7, 2008

Abstract

A *perfect hash function* (PHF) $h : U \to [0, m-1]$ for a key set S is a function that maps the keys of S to unique values. A minimal perfect hash function (MPHF) is a PHF with the smallest range, i.e., $m = n$. The minimum amount of space to represent a PHF for a given set S is known to be approximately $1.4427n^2/m$ bits, where $n = |S|$. 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.

The main results of this thesis proposal are the design, analysis and implementation of: (i) internal memory based algorithms that assume uniform hashing to build PHFs (for $m = 1.23n$) and MPHFs (for $m = n$) based on random graphs, and (ii) an external memory based algorithm that has experimentally proven practicality for sets in the order of billions of keys and has time and space usage carefully analyzed without assuming uniform hashing, which is an unrealistic assumption because each uniform hash function requires $\Omega(|U| \log m)$ bits of storage space.

Both internal and external algorithms have the following properties: (i) evaluation of a PHF or a MPHF requires constant time, (ii) the algorithms are simple to describe and implement, and generate the functions in linear time, (iii) the amount of space needed to represent a PHF is $1.95n$ bits and a MPHF is $2.62n$ bits, which is around a factor of 2 from the information theoretical minimum of approximately $1.17n$ and $1.4427n$ bits, respectively. Therefore, the algorithms give low space usage for realistic values of n. No previously known algorithm has these properties. To our knowledge, any algorithm in the literature with the third property either: (i) requires exponential time for construction and evaluation, or (ii) uses near-optimal space only asymptotically, for extremely large n.

The external memory based algorithm achieves an order-of-magnitude increase in the size of the problem to be solved compared to previous "practical" methods. We demonstrate the scalability of our algorithm by constructing minimum perfect hash functions for a set of 1.024 billion URLs from the World Wide Web of average length 64 characters in approximately 62 minutes, using a commodity PC.

Resumo

Uma *função hash perfeita* (FHP) $h : U \to [0, m-1]$ para um conjunto de chaves S é uma função que mapeia as chaves de S para valores únicos. Uma função hash perfeita mínima (FHPM) é uma FHP com o menor intervalo, isto é, $m = n$. A quantidade mínima de espaço para representar uma FHP para um dado conjunto S é aproximadamente $1.4427n^2/m$ *bits*, onde $n = |S|$. 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.

Os principais resultados desta proposta de tese são o projeto, análise e implementação de: (i) algoritmos baseados em memória interna que assumem hashing uniforme para construir FHPs (para $m = 1.23n$) e FHPMs (para $m = n$) baseados em grafos randômicos, e (ii) um algoritmo baseado em memória externa que pode ser utilizado na prática para conjuntos contendo bilhões de chaves que tem complexidade de tempo e espaço cuidadosamente analisados sem assumir hashing uniforme, a qual é uma suposição não realista porque cada função hash uniforme requer $\Omega(|U| \log m)$ bits para ser armazenada.

Ambos algoritmos internos e externo tem as seguintes propriedades: (i) avaliação de uma FHP ou uma FHPM requer tempo constante, (ii) os algoritmos são simples para descrever e implementar, e geram as funções em tempo linear, (iii) a quantidade de espaço necessário para representar uma FHP é $1.95n$ bits e uma FHPM é $1.4427n$ bits, o qual está em torno de um fator de 2 distante do mínimo teórico de aproximadamente $1.17n$ e $1.4427n$ bits, respectivamente. Portanto, os algoritmos geram funções com baixa utilização de espaço na representação e na geração para valores de n realísticos. Nenhum algoritmo conhecido anteriormente possui estas propriedades. Pelo que sabemos, qualquer algoritmo da literatura com a terceira propriedade tem um dos seguintes problemas: (i) requer tempo exponencial para construir e avaliar a função resultante, ou (ii) usa espaço quase ótimo somente assintóticamente, para valores de n extremamente grandes.

O algoritmo baseado em memória externa alcança um almento de uma ordem de magnitude no tamanho do problema a ser resolvido, quando comparado à métodos "práticos" anteriormente publicados. Nós demonstramos a escalabilidade do nosso algoritmo gerando uma FHPM para um conjunto com 1.024 bilhões de URLs da *World Wide Web*, cada uma com 64 caracteres na média, em aproximadamente 62 minutos, usando um simples PC.

Accepted and Submitted Papers

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. Proceedings of the 10th Workshop on Algorithms and Data Structures (WADs'07), pages 139–150. Springer LNCS vol. 4619, 2007. To appear.

3. F.C. Botelho, and N. Ziviani. External Perfect Hashing for Very Large Key Sets. Submitted to 16th Conference on Information and Knowledge Management, 2007.

Near Space-Optimal Perfect Hashing Algorithms

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ubiquitous in areas including artificial intellegence, data structures, database, data mining and information retrieval is the need to access items based on the value of a key. Some types of databases are updated only rarely, typically by periodic batch updates. This is true, for example, for most data warehousing applications (see [53] 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 set of keys 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.

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.

1.1 Motivation

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

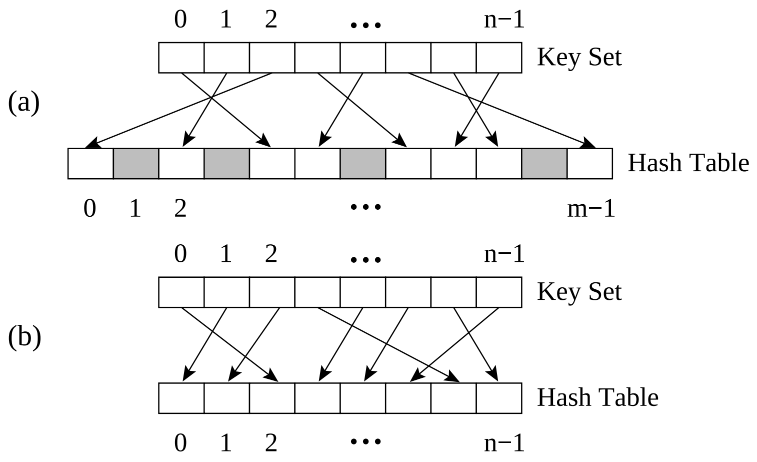


Figure 1.1: (a) Perfect hash function (b) Minimal perfect hash function

in programming languages or interactive systems, universal resource locations (URLs) in web search engines, or item sets in data mining techniques. Search engines are nowadays indexing tens of billions of pages and algorithms like PageRank [11], which uses the web link structure to derive a measure of popularity for Web pages, would benefit from a MPHF for storage and retrieval of billions of URLs.

Until now, because of the limitations of current algorithms, the use of MPHFs is restricted to scenarios where the set of keys being hashed is relatively small. However, in many cases it is crucial to deal in an efficient way with very large sets of keys. In the IR community, the work with huge collections is a daily task. For instance, the simple assignment of number identifiers to web pages of a collection can be a challenging task. While traditional databases simply cannot handle more traffic once the working set of URLs does not fit in main memory anymore, one of the algorithms we propose here to construct MPHFs can easily scale to billions of entries.

We now present some examples where minimal perfect hash functions have successfully been applied to:

- A perfect hash function can be used to implement a data structure with the same functionality as a Bloom filter [45][1]. 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. This data structure

[1]The Bloom filter, conceived by Burton H. Bloom in 1970, 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. Elements can be added to the set, but not removed (though this can be addressed with a counting filter). The more elements that are added to the set, the larger the probability of false positives.

requires around 30% less space usage when compared with Bloom filters, plus the space for the perfect hash function. Bloom filters have applications in distributed databases and data mining (association rule mining [14, 15]).

- Perfect hash functions have also been used to speed up the partitioned hash-join algorithm presented in [43]. By using a perfect hash function to reduce the targeted hash-bucket size from 4 tuples to just 1 tuple they have avoided following the bucket-chain during hash-lookups that causes too many cache and translation lookaside buffer (TLB) misses.

- Suppose there is a composite foreign key to a table T of size n. Then the size of the key needed in T will typically be larger than log n. For example, suppose tuples of R contain geographical coordinates that are used as foreign key references. Replacing the coordinates with a surrogate key may be a bad choice if common queries on R involve conditions on the coordinates, as a join would be required to retrieve the coordinates. In general, if we have a natural foreign key that carries information relevant for queries, we can avoid the cost of additionally storing a surrogate key.

The perfect hash function used depends on the set S of distinct key values that occur. It is known that maintaining a perfect hash function dynamically under insertions into S is only possible using space that is super-linear in n [20]. However, in this thesis proposal we consider the case where S is fixed, and construction of a perfect hash function can be done as part of the preprocessing of data (e.g., in a data warehouse). To the best of our knowledge, previously suggested perfect hashing methods have not been able to generate functions for realistic data sizes that require a number of bits to be stored close to the theoretical lower bound, which is around $1.4427n$ bits (see in Section 1.2 an intuitive proof and refer to [19] for a complete proof). Second, all previous methods suffer from either an incomplete theoretical understanding (so there is no guarantee that it works well on a given data set) or seems impractical due to a very intricate and time-consuming evaluation procedure.

Though there has been considerable work on how to construct good perfect hash functions, there is a gap between theory and practice among all previous methods on minimal perfect hashing. On one side, there are good theoretical results without experimentally proven practicality for large key sets. On the other side, there are the theoretically analyzed time and space usage algorithms that assume that uniform random hash functions are available for free (see Definition 3 in Section 1.2), which is an unrealistic assumption.

Our aim in this thesis proposal is twofold. First, we present new algorithms for constructing MPHFs in the internal memory that, as usual, assume uniform hashing and require $O(n)$ computer words in the generation process. The algorithms outperform the main practical algorithms available in the literature. Second, we have used a number of techniques from the literature to obtain a scalable external memory based algorithm that is theoretically well-understood because it does not assume uniform hashing. Also, the algorithm requires just $O(N)$ computer words, where $N \ll n$ in the generation process. That is why it achieves an order-of-magnitude in the size of the greatest key set for which a MPHF was obtained in the literature [8] on a commodity PC. This improvement comes from a combination of a novel, theoretically optimal perfect hashing scheme that greatly simplifies previous methods, and the fact that our algorithm is designed to make good use of the memory hierarchy.

1.2 Basic Concepts and Notation

In this section we present the basic concepts and notation used throughout this thesis proposal.

Definition 1 Let U be an *universe of keys* of size u and let S be a subset of U containing n keys, where $n \ll u$. Each key is made up by symbols from a finite an ordered alphabet Σ of size $|\Sigma|$. The maximum size of a key is denoted by L.

Definition 2 Let $h : U \to M$ be a *hash function* that maps the keys from U to a given interval of integers $M = [0, m-1] = \{0, 1, \ldots, m-1\}$. Given a key $x \in S$, the hash function h computes an integer in $[0, m-1]$.

Definition 3 *Uniform hash assumption*: the classic analysis of hashing schemes often entails the assumption that the hash functions used are uniformly chosen at random among all the functions from U to M, where $u = |U|$ and $m = |M|$. There are m^u possible hash functions because each element of U can be mapped to anyone of the m integers from the range M. This assumption is impractical since just specifying such a function requires $O(u \log m)$ bits[2], which usually far exceeds the available storage.

[2]Throughout the thesis proposal we denote $\log_2 x$ as $\log x$

Fortunately in most cases heuristic hash functions behave very closely to the expected behavior of random hash functions, but there are cases when rigorous probabilistic guarantees are necessary [12]. 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 [13] to the concept of universal hashing.

Definition 4 A family of hash functions $\mathcal{H}$ is defined as *weakly universal* if for any pair of distinct elements $x_1, x_2 \in U$ and h is chosen uniformly at random from $\mathcal{H}$ then

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

Definition 5 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 an universal family, rather than the assumption that h is chosen uniformly at random from among all possible functions. In other words, limited randomness suffices in practice [52].

Definition 6 A *perfect hash function* phf : $S \to M$ is an injection on S. That is, for all pair s_1, $s_2 \in S$ such that $s_1 \neq s_2$, then phf(s_1) $\neq$ phf(s_2), where $m \geq n$. For being an injection, phf maps each key in S to an unique integer in M, as shown in Figure 1.1 (a). As no collisions occur, 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* mphf : $S \to M$ is a bijection on S. That is, each key in S is mapped to an unique integer in M, and $m = n$, as shown in Figure 1.1 (b).

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

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

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

$$Pr_{ph}(n,m) = \frac{(m-1)(m-2)\dots(m-n+1)}{m^n} = \left(1-\frac{1}{m}\right)\left(1-\frac{2}{m}\right)\dots\left(1-\frac{n-1}{m}\right)$$

When the table is large (i.e. $m \gg n$), we can use the approximation $e^x = 1 + x$ for small x to obtain:

$$\begin{aligned} Pr_{ph}(n,m) &\approx e^{-1/m} \cdot e^{-2/m} \cdots e^{-(n-1)/m} \\ &\approx e^{-(1+2+3+(n-1))/m} \\ &\approx e^{-(n^2-n)/2m} \end{aligned}$$

Thus, the presence of a hash collision is highly likely when the table size m is much less than n^2. This is an instance of the well known "birthday paradox", which says that in a group of only 23 people have more than 50% chance of having at least one shared birthday. Therefore, at least $1/Pr_{ph}(n,m) = e^{(n^2-n)/2m}$ hash functions are required to generate a PHF. Thus, at least $\frac{n^2-n}{2m}\log e$ bits are required to encode that set of hash functions. $\square$

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

Proof: The probability of finding a minimal perfect hash (where $n = m$) is:

$$Pr_{mph}(n,n) = \frac{n!}{n^n} = e^{\log n! - n\log n} \approx e^{(n\log n - n) - n\log n} = e^{-n}$$

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

The above proofs have been previously reported by Fox et al [28] based on earlier analysis by Mehlhorn [44].

1.3 Related work

There is a gap between theory and practice among minimal perfect hashing methods. On one side, 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 side, there are two categories of practical algorithms: the theoretically analyzed time and space usage algorithms that assume uniform random hash functions for their methods, which is an unrealistic assumption, and the 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.

1.3.1 Theoretical results

In this section we review some of the most important theoretical results on minimal perfect hashing. For a complete survey until 1997 refer to Czech, Havas and Majewski [19].

Fredman, Komlós and Szemerédi [30] proved, using a counting argument, that at least $n \log e + \log \log u - O(\log n)$ bits are required to represent a MPHF, provided that $u \geq n^{\alpha}$ for some $\alpha > 2$. In general, for $m > n$ the space required to represent a PHF is around $(1 + (m/n - 1) \ln(1 - n/m))\, n \log e$ bits. A simpler proof of this was later given by Radhakrishnan [51].

Mehlhorn [44] has made this bound almost tight by providing an algorithm that constructs a MPHF that can be represented with at most $n \log e + \log \log u + O(\log n)$ bits. However, his algorithm is far away from practice because its construction and evaluation time is exponential on n (i.e., $n^{\theta(ne^n u \log u)}$).

Schmidt and Siegel [52] proposed the first algorithm for constructing a 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 [31]. In this model an element of the universe U fits into one machine word, and arithmetic operations and memory accesses have unit cost. From a practical point of view, the algorithm of Schmidt and Siegel 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 [52] aims to describe its algorithmic ideas in the clearest possible way, not trying to optimize the constant, it appears hard to improve the space usage significantly.

More recently, Hagerup and Tholey [32] 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 construction time is $O(n + \log \log u)$ using $O(n)$ words of space. Again, the terms involving u are negligible. In spite of its theoretical importance, the Hagerup and Tholey [32] algorithm is also not practical, as it emphasizes asymptotic space complexity only. (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.

1.3.2 Practical results that assume uniform hashing

Let us now describe the main practical results analyzed with the unrealistic assumption that uniform random hash functions are available for free.

Schemes with space usage known analytically

The algorithm proposed by Czech, Havas and Majewski [18] uses the uniform hashing assumption to construct order preserving MPHFs. The method uses two uniform random hash functions $h_1(x) : S \to [0, cn-1]$ and $h_2(x) : S \to [0, cn-1]$ to generate MPHFs in the following form: $\mathrm{mphf}(x) = (g[h_1(x)] + g[h_2(x)] \bmod n$ 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 [8] 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 [18] and the resulting MPHF 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.

Majewski et al [42] have proposed a family of MPHF methods that generalizes the work in [18]. Although the resulting functions are almost as compact as the ones generated by the work in [8], they still require $O(n \log n)$ bits to be stored. Botelho, Pagh and Ziviani [8] have 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 random 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 [46]. He proposed a family of randomized algorithms for constructing MPHFs of the form $\mathrm{mphf}(x) = (f(x) + d[g(x)]) \bmod n$, where f and g are chosen from a family of universal hash functions [13] 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 time $O(n)$ and stored in $(2 + \epsilon)n \log n$ bits.

Dietzfelbinger and Hagerup [21] improved the algorithm proposed in [46], reducing from $(2 + \epsilon)n \log n$ to $(1 + \epsilon)n \log n$ the number of bits required to store the function, but in their approach f and g must be chosen from a class of hash functions that meet additional requirements. Woelfel [55] 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.

Prabhakar and Bonomi [50] have 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.

Schemes with space usage not known analytically

Fox et al. [29] 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 a 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 a 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 [19, 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 [28] improved the above result to get a function that can be stored in cn bits. The method uses four uniform random hash functions $h_{10} : S \to [0, n-1]$, $h_{11} : [0, p_1 - 1] \to [0, p_2 - 1]$, $h_{12} : [p_1, n-1] \to [p_2, b-1]$ and $h_{20} : S \times \{0, 1\} \to [0, n-1]$

to construct a MPHF that has the following form:

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

where $p_1 = 0.6n$ and $p_2 = 0.3n$ were experimentally determined, and $\lceil b = cn/(\log n + 1)\rceil$. 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 no guarantee on the size of the resulting MPHF is provided.

1.3.3 Empirical results

In this section we discuss results that present only empirical evidences for specific applications. Lefebvre and Hoppe [41] have recently designed MPHFs that require up to 7 bits per key to be stored and are tailored to represent sparse spatial data. As the works by Fox et al [28, 29], the space usage is not known analytically.

In the same trend, Chang, Lin and Chou [14, 15] have designed MPHFs tailored for mining association rules and traversal patterns in data mining techniques.

1.4 Objectives and Initial Contributions

Our primary objective is to design 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.

The attractiveness of using PHFs and MPHFs depend on the following issues [32]:

1. The amount of CPU time required by the algorithms for constructing the functions.

2. The space requirements of the algorithms for constructing the functions.

3. The amount of CPU time required by a function for each retrieval.

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

Our initial contributions correspond to the design, analysis and implementation of three new algorithms to generate PHFs and MPHFs. The very first result presented in [8] improved the space requirement of the algorithm by Czech, Havas and Majewski [18] at the expense of generating functions in the same form that are not order preserving, but are computed in time $O(1)$. The algorithm generates MPHFs based on simple random graphs that may have cycles, while the algorithm in [18] uses acyclic random graphs. Both algorithms are linear on n, but our algorithm runs 60% faster than the one in [18], 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 [18], the algorithm assumes uniform hashing and needs $O(n)$ computer words of the Word RAM model to construct the functions. The complete description of the algorithm is presented in Chapter 2.

The second work [9] presents a simple, efficient, near space-optimal and practical family $\mathcal{F}$ of algorithms for generating PHFs and MPHFs. The algorithms in $\mathcal{F}$ use 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. This idea is not new, see e.g. [42], but we proceed differently to achieve a space usage of $O(n)$ bits rather than $O(n \log n)$ bits. Therefore, we have reduced by a factor of $O(\log n)$ the complexity order of the algorithms in [42]. Evaluation time for all schemes considered is constant. For $r = 2$ we obtain a space usage of around $3.6n$ bits for a MPHF, and for $r = 3$ we obtain a space usage of less than $2.7n$ bits for a MPHF. This is within a factor of 2 from the information theoretical lower bound of approximately $1.4427n$ bits.

More compact, and even simpler, representations can be achieved for larger m. For example, for $m = 1.23n$ we can get a space usage of $1.95n$ bits. This is slightly more than two times the information theoretical lower bound of around $1.17n$ 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. As usual, the algorithms assume uniform hashing and require $O(n)$ computer words in the construction process. The complete description of the family is presented in Chapter 3.

In our third work [10] we present a new external memory based algorithm, which is referred to as *EPH algorithm*. The algorithm split the incoming key set S into small buckets containing at most 256 keys. Then, a MPHF is generated for each bucket and using an *offset* array we obtain a MPHF for S. To the best of our knowledge the algorithm is the first one that demonstrates the capability of generating MPHFs for sets in the order of billions of keys, and the generated functions require less than 4 bits per key to be stored. This improvement comes mainly from the fact that our method is designed to make good use of the memory hierarchy because it operates on small buckets of keys, increasing the probability of cache hits.

Differently from our two previous work and the practical results in the literature, the EPH algorithm does not assume uniform hashing and needs $O(N)$ computer words, where $N \ll n$, for the construction process. Therefore, the algorithm is theoretically well-understood and increases one order of magnitude in the size of the greatest key set for which a MPHF was obtained in the literature [8] on a commodity PC. Notice that both space usage for representing the MPHF and the construction time are carefully proven. As a consequence, the algorithm will work for every key set. Additionally, the EPH algorithm is much simpler than previous theoretical well-founded schemes, as the ones presented in [32, 52]. The complete description of the algorithm is presented in Chapter 4.

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. We have received very good feedbacks about the practicality of the library. Second, we realized that there was a lack of similar libraries in the open source community.

1.5 Organization of this Work

This text is organized as follows: Chapter 2 presents the first algorithm we came up with to generate MPHFs. Chapter 3 presents a family of algorithms that generates near space-optimal MPHFs. Chapter 4 presents the first algorithm that is theoretically well-understood and can be applied to sets in the order of billions of keys, which is our main partial result. Finally, in Chapter 5 we conclude and present some suggestions regarding future steps to be taken in this research.

Chapter 2

A practical minimal perfect hashing method

In this chapter we describe a new way of constructing minimal perfect hash functions. The algorithm shares several features with the one due to Czech, Havas and Majewski [18], from now on referred to as CHM algorithm. In particular, our 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 wish to generate the hash function. The two main differences between our algorithm and theirs are as follows: (i) we generate random graphs $G = (V, E)$ with $|V| = cn$ and $|E| = |S| = n$, where $c = 1.15$, and hence G contains cycles with high probability[1], while they generate *acyclic* random graphs $G = (V, E)$ with $|V| = cn$ and $|E| = |S| = n$, with a greater number of vertices: $|V| > 2n$; (ii) they generate order preserving minimal perfect hash functions while our algorithm does not preserve order. Thus, our algorithm improves the space requirement at the expense of generating functions that are not order preserving.

Our algorithm is efficient and may be tuned to yield a MPHF with a very economical description. As the algorithm in [18], our algorithm produces a 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. Our scheme 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, our algorithm running on a commodity PC takes 811 seconds on average. In Section 2.1 we

[1]In the sequel, we write “with high probability” to mean with probability $1 - n^{-\delta}$ for $\delta > 0$

present the CHM algorithm and in Section 2.2 we present our algorithm. We finish this chapter comparing the two algorithms in Section 4.3.

2.1 The CHM algorithm

In this section we briefly present the CHM algorithm. For that, consider now 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 $\mathrm{mphf} : E \to \{0, 1, \ldots, n - 1\}$, defined as

$$\mathrm{mphf}(e) = (g(a) + g(b)) \bmod n \tag{2.1}$$

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 [28]. 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 2.2.

The ordering and searching steps of the MOS approach are a very simple way of solving the perfect assignment problem. Czech, Havas and Majewski [18] 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 $|x| \le L$ and $j \in \{1, 2\}$, we let

$$h_j(x) = \left(\textstyle\sum_{i=0}^{|x|-1} \mathrm{table}_j[i, x[i]] \right) \bmod m. \tag{2.2}$$

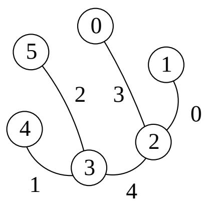
Thus, set S has a corresponding graph $G = G(h_1, h_2)$, with $V = \{0, 1, \ldots, m - 1\}$, where $|V| = m$, and $E = \{\{h_1(x), h_2(x)\} : x \in S\}$. In order to guarantee acyclicity the algorithm repeatedly selects h_1 and h_2 until the corresponding graph is acyclic. For the solution to be useful 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. [33] proved that if $m = cn$ holds with $c > 2$ the probability that G is acyclic is

$$Pr_a = e^{1/c}\sqrt{\frac{c-2}{c}}. \tag{2.3}$$

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 $\mathrm{mphf}(e) \in [0, n-1]$ in the order of the keys of S to obtain an order preserving function. Figure 2.1 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 2.1 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 $\mathrm{mphf}(e)$, set $g(b)$ to $(\mathrm{mphf}(e) - g(a)) \bmod n$. In Figure 2.1, 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 2.1: 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 2.1 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.

2.2 Our 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. (2.2)). We build a random graph $G = G(h_1, h_2)$ on S, whose edge set is $\{\{h_1(x), h_2(x)\} : x \in S\}$. There is an edge in G for each key in the set of keys 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., [4, 36]). 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 neighbour. 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 $\text{mphf}(x) = g(h_1(x)) + g(h_2(x))$ $(x \in S)$ is a 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 3.2 presents a pseudo code for the algorithm. The procedure GenerateMPHF (S, g) receives as input the set of keys 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 2.2: Main steps of the algorithm for constructing a minimal perfect hash function

2.2.1 Mapping Step

The procedure Mapping (S, G) receives as input the set of keys 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. (2.2)). 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 [24, 25, 26] (for a modern treatment, see [4, 36]). 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[2] 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 [49] 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 [49], the number of vertices of G_{crit} is

$$|V(G_{\text{crit}})| = (1 + o(1))(1 - T)bm \tag{2.4}$$

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

$$|E(G_{\text{crit}})| = (1 + o(1))\Big((1 - T)b + b(d + T - 2)/2\Big)m \tag{2.5}$$

[2]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 $m \to \infty$'.

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 2.1 gives some values for $|V(G_{\text{crit}})|$ and $|E(G_{\text{crit}})|$ using Eqs (2.4) and (2.5). The theoretical value for c is around 1.152, which is remarkably close to the empirical results presented in Table 2.2. 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 most pleasant, and this is why we consider that this 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 2.1: 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

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 2.2: 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

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 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 [48] 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,

$$\lim_{n\rightarrow\infty} p(n) \simeq e^{-\frac{1}{c^2}}. \tag{2.6}$$

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.

2.2.2 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 done.

Figure 2.3(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 2.3(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 2.3(b).

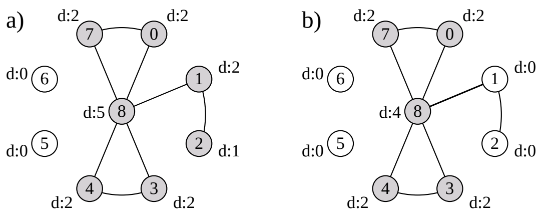


Figure 2.3: Ordering step for a graph with 9 vertices and 8 edges

Analysis of the Ordering Step

The time complexity of the ordering step is $O(|V(G)|)$ (see [19]). As $|V(G)| = m = cn$, the ordering step takes $O(n)$ time.

2.2.3 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

$$\text{mphf}(e) = g(a) + g(b) \qquad (e = \{a, b\}) \tag{2.7}$$

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_{crit}, G_{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 2.3(b), a step by step example of the assignment of values to vertices in G_{crit} is presented in Figure 2.4. 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 2.4(a) is chosen, the assignment $g(8) = 0$ is made and x is set to 1.

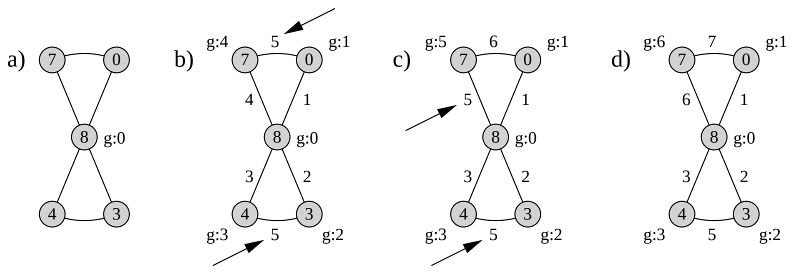


Figure 2.4: Example of the assignment of values to critical vertices

In Figure 2.4(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 2.4(c). Since $g(8) + 5 = 5 \in A_E$, x is again incremented by 1 and set to 6, as depicted in Figure 2.4(d). These two reassignments are indicated by the arrows in Figure 2.4. 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 2.3(b), a step by step example of the assignment of values to vertices in G_{ncrit} is presented in Figure 2.5. Figure 2.5(a) presents the initial state of the algorithm. The critical vertex 8 is the only one that has non-critical neighbours. In the example presented in Figure 2.4, 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 2.5(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 2.5(c). This process is repeated until the UnAssignedAddresses list becomes empty.

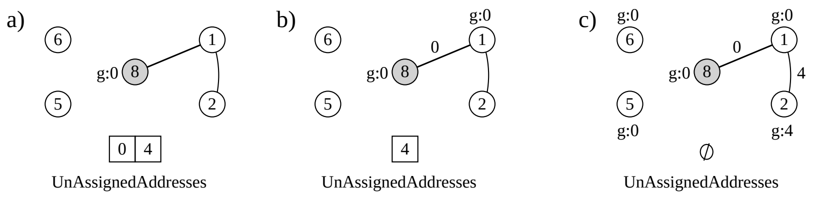


Figure 2.5: Example of the assignment of values to non-critical vertices

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.[3] The fact that every edge is either a tree edge or a back edge (see, e.g., [17]) then implies the following.

Theorem 3 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 4 We have $A_{\text{max}} \leq 2|V(G_{\text{crit}})| - 3 + 2N_t$.

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. (3.1). 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, $A_{max} \leq |V(G_{\text{crit}})| - 1 + N_t + |V(G_{\text{crit}})| - 2 + N_t \leq 2|V(G_{\text{crit}})| - 3 + 2N_t$. $\square$

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 3 and 4, we have $A_{\text{max}} \leq 2|V(G_{\text{crit}})| - 3 + 2N_t \leq 2|V(G_{\text{crit}})| - 3 + 2N_{\text{bedges}} \leq 2|V(G_{\text{crit}})| - 3 + 2(|E(G_{\text{crit}})| - |V(G_{\text{crit}})| + 1) \leq 2|E(G_{\text{crit}})| - 1$. As by hypothesis $|E(G_{\text{crit}})| \leq n/2$, we have $A_{\text{max}} \leq n - 1$, as required.

[3]The number of vertices in G_{crit} outside the giant component is provably very small for $c = 1.15$; see [4, 36, 49].

In the mathematical analysis of our algorithm, what is left open is a single problem: prove that $N_t \leq N_{\text{bedges}}$.[4]

We now show experimental evidence that $N_t \leq N_{\text{bedges}}$. Considering Eqs (2.4) and (2.5), 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 3, $N_{\text{bedges}} = 0.501n - 0.401n + 1 = 0.1n + 1$. Table 2.3 presents 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 2.3: 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 3, that $N_t \leq |E(G_{\text{crit}})| - |V(G_{\text{crit}})| +$

[4]Bollobás and Pikhurko [5] 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.

$1 = O(|E(G_{\text{crit}})|)$. 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 $O(n)$.

2.3 Experimental Results

We now present some experimental results. The same experiments were run with our algorithm and the CHM algorithm. The two algorithms were implemented in the C language and are available in the C Minimal Perfect Hashing Library 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 2.4 presents the main characteristics of the two algorithms. The number of edges in the graph $G = (V, E)$ is $|S| = n$, the number of keys in the input set S. The number of vertices of G is equal to $1.15n$ and $2.09n$ for our 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 our 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 our algorithm and the CHM algorithm, respectively. Our 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 while our algorithm does not preserve order.

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

Table 2.4: Main characteristics of the algorithms

Table 2.5 presents time measurements. 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 our algorithm over the CHM algorithm.

n	Our 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 2.5: Time measurements for our algorithm and the CHM algorithm

The mapping step of the new algorithm is faster because the expected number of iterations in the mapping step to generate G are 2.13 and 2.92 for our algorithm and the CHM algorithm, respectively. The graph G generated by our algorithm has $1.15n$ vertices, against $2.09n$ for the CHM algorithm. These two facts make our algorithm faster in the mapping step. The ordering step of our 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 our algorithm is, on average, approximately 58% faster than the CHM algorithm.

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 2.2.3). 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 a 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 2.6 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 2.5) to 101.74 seconds for $c = 1$ and to 102.19 seconds for $c = 0.93$.

We compared our algorithm with the ones proposed by Pagh [46] and Dietzfelbinger and Hagerup [21], respectively. The authors sent to us their source code. In their

n	Our algorithm $c = 1.00$				Our 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 2.6: Time measurements for our tuned algorithm with $c = 1.00$ and $c = 0.93$

implementation the set of keys is a set of random integers. We modified our implementation to generate our 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 our algorithm, Pagh's algorithm and Dietzfelbinger and Hagerup's algorithm, respectively. Thus, our 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. Our 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.

2.4 Conclusion

We have presented a practical method for constructing minimal perfect hash functions for static sets that is efficient and may be tuned to yield a function with a very economical description. The algorithm improves the space requirement of the algorithm proposed by Czech, Havas and Majewski 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. Our next result presented in Chapter 3 shows how to generate simple and space-optimal MPHFs.

Chapter 3

A family of near space-optimal algorithms

In this chapter we describe a simple and near space-optimal family $\mathcal{F}$ of algorithms for generating minimal perfect hash functions for a set S of n elements[1]. The algorithms from $\mathcal{F}$ use acyclic *hypergraphs* given by function values of r uniform random hash functions on S, also called $r-$*graphs*, for constructing PHFs and MPHFs that can be stored in near optimal space (i.e., $O(n)$ bits.) A $r-$graph is a generalization of a standard graph where each edge connects r vertices instead of only two. Indeed, acyclic random hypergraphs has been used in previous PHF constructions [42], but we will proceed differently to achieve a space usage of $O(n)$ bits rather than $O(n \log n)$ bits.

3.1 The family of algorithms

In this section we present the family $\mathcal{F}$ that contains three-step algorithms for constructing near space-optimal MPHFs. Figure 4.6 gives an overview of the algorithms on a key set $S \subseteq U$ containing the first 4 month names abbreviated to the first three characters.

The first step, referred to as the *Mapping Step*, maps the key set S to a set of $n = |S|$ edges forming an acyclic r-partite hypergraph $G_r = (V, E)$, where $|E(G_r)| = n$, $|V(G_r)| = m$ and $r \geq 2$. In the example of Figure 4.6 (a) we used $r = 2$ uniform hash functions

[1]Chazelle et al [16] present a way of constructing PHFs that is equivalent to ours. It is explained as a modification of the "Bloomier Filter" data structure at the end of Section 3.3, but they do not make explicit that a PHF is constructed. Thus, the simple construction of a PHF described must be attributed to Chazelle et al. 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 that PHFs.

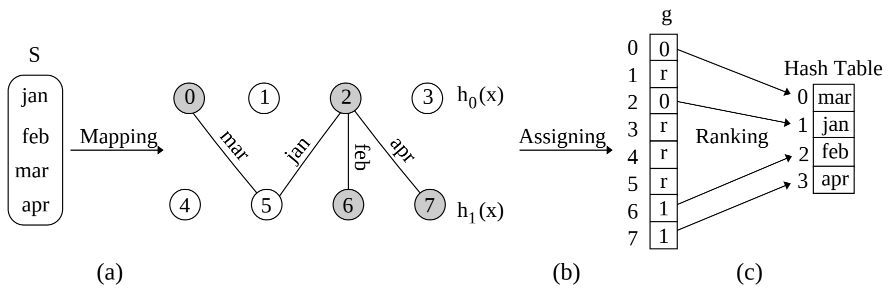


Figure 3.1: (a) Mapping step generates a bipartite 2−graph. (b) Assigning step builds a labeling g so that each edge is uniquely assigned to a vertex. (c) Ranking step builds a function rank : $V \to [0, n-1]$

$h_0 : U \to [0, m/2 - 1]$ and $h_1 : U \to [m/2, m-1]$, where $m = 8$, to build a bipartite random graph $G_r = G_r(h_0, h_1)$ with vertex set $V(G_r) = [0, m-1]$ and edge set $E(G_r) = \{\{h_0(x), h_1(x)\} \mid x \in S\}$. Note that each key in S is associated with an edge in G_r.

The second step, referred to as the *Assigning Step*, associates uniquely each edge with one of its r vertices. Here, "uniquely" means that no two edges may be assigned to the same vertex. Thus, the Assigning Step finds a PHF for S with range $V(G_r)$. As shown in Figure 4.6 (b), the Assigning step outputs a labeling $g : V(G) \to \{0, 1, r = 2\}$ so that each edge or key is uniquely mapped to a vertex in G_r. For instance, jan is mapped to 2, feb to 6, mar to 0, and apr to 7. By construction of g we guarantee that each vertex $v \in V(G)$ assumes just two disjoint states: *assigned* and *unassigned*, and exactly n vertices are assigned. A vertex v is defined as unassigned when $g(v) = r$.

Therefore, a function rank that counts how many vertices are assigned before a given vertex $v \in V(G_r)$, which is uniquely associated with a key $x \in S$, is a MPHF on S. Note that the four keys of the example presented in Figure 4.6 (c) are placed into a hash table of size 4 without collisions. For example, rank($v = 7$) = 3 and it means that there are three vertices assigned before vertex 7. Finally, the third step, referred to as the *Ranking Step* is responsible for building the data structures used to compute function rank in constant time.

For the analysis, we assume that we have at our disposal r hash functions $h_i : U \to [i\frac{m}{r}, (i+1)\frac{m}{r} - 1]$, $0 \leq i < r$, which are independent and uniformly distributed function values. (This is the "uniform hashing" assumption, see Section 3.2 for justification.) The r functions and the set S define, in a natural way, a random r−partite hypergraph. We define $G_r = G_r(h_0, h_1 \ldots, h_{r-1})$ as the hypergraph with vertex set $V(G_r) = [0, m-1]$ and edge set $E(G_r) = \{\{h_0(x), h_1(x), \ldots, h_{r-1}(x)\} \mid x \in S\}$. For the Mapping Step to work, we need G_r to be simple and acyclic, i.e., G_r should not have multiple edges and cycles. This is handled by choosing r new hash functions in the event that the Mapping Step fails.

We use an edge oriented data structure proposed by Ebert [23] to represent the hypergraphs. A detailed description of that data structure is presented in [6].

As mentioned before, there are exactly n assigned vertices. So, if we find out a PHF $\mathrm{phf} : S \rightarrow V(G_r)$ that returns all the assigned vertices we will be able to build a MPHF from it by using ranking. Let us define that PHF as

$$\mathrm{phf}(x) = h_i(x), \text{ where } i = (g(h_0(x)) + g(h_1(x)) + \cdots + g(h_{r-1}(x))) \bmod r. \tag{3.1}$$

The function $g : V(G_r) \rightarrow \{0, 1, \ldots, r\}$ is a labeling of the vertices of $V(G_r)$. We will show how to choose the labeling such that phf is 1-1 on S, given that G_r is acyclic. We use $g(v) = r$ to represent unassigned vertices because it is equal to zero in modulo r operations. Note that $i = i(x)$ is used to select the vertex from the edge associated with x that uniquely represent x. For instance, in the example of Figure 4.6, $\mathrm{phf}(\mathrm{apr}) = 7$ because $i(\mathrm{apr}) = (g(2) + g(7)) \bmod 2 = 1$, and $\mathrm{phf}(\mathrm{jan}) = 2$ because $i(\mathrm{jan}) = (g(2) + g(5)) \bmod 2 = 0$.

Therefore, our problem is reduced to computing the labeling g such that the following function is a bijection on S, i.e., a MPHF on S:

$$\mathrm{mphf}(x) = \mathrm{rank}(\mathrm{phf}(x)) \tag{3.2}$$

where $\mathrm{rank} : V(G_r) \rightarrow [0, n-1]$ is a function defined as:

$$\mathrm{rank}(x) = |\{y \in V(G_r) \mid y < x \text{ and } g(y) \neq r\}|. \tag{3.3}$$

Figure 3.2 presents a pseudo code for our family of minimal perfect hashing algorithms. If we strip off the third step we will build PHFs instead. The family of algorithms receives as input the key set S and the edge size r, and produces the resulting functions represented by the labeling g and a data structure, referred to as rankTable, used to allow the computation of Eq. (3.3) in time $O(1)$. We now describe each step in details.

```
procedure Generate ($S$, $r$, $g$, rankTable)
    Mapping ($S$, $G_r$, $L$);
    Assigning ($G_r$, $L$, $g$);
    Ranking ($g$, rankTable);
```

Figure 3.2: Main steps of the family of algorithms

3.1.1 Mapping Step

The Mapping step takes the key set S as input, and creates the random hypergraph G_r and a list of edges L. A sufficient condition to find the labeling g is to generate an *acyclic* random hypergraph. 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 [19, Page 103]. The list L stores the deleted edges in the order of deletions (i.e., the first edge in L was the first deleted edge, the second edge in L was the second deleted edge, and so on.) Indeed, the list L is obtained whenever we test whether G_r is acyclic. This test can be done in $O(n)$ time by the following algorithm:

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 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 3.3 presents one possible output of the aforementioned test when applied to the random hypergraph G_2 presented in Figure 4.6. The three edges containing vertices of degree one were firstly deleted and stored in L. Then the only edge containing a vertex of degree two was deleted and stored in L.

0	1	2	3	
{0,5}	{2,6}	{2,7}	{2,5}	L

Figure 3.3:

Figure 3.4 presents a pseudo code for the Mapping Step.

Analysis of the Mapping Step

When a cyclic random hypergraph 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 an acyclic random hyperfig:mapgraph G_r as a random variable Z that follows a geometric distribuition. Let Pr_a be the probability of generating an acyclic random hypergraph. 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.

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

Figure 3.4: Mapping step

We want to ensure that $Pr_a = \Omega(1)$. For that we define $m = cn$, where $c = c(r)$ is a function defined in the following. For $r = 2$, we can use the techniques presented in [35] to show that $Pr_a = \sqrt{1 - (2/c)^2}$. 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). For $r > 2$, it seems to be technically difficult to obtain a rigorous bound on Pr_a. However, the heuristic argument presented in [19, Theorem 6.5] also holds for our $r-$partite random hypergraphs. Their argument suggests that if $c = c(r)$ is given by

$$c(r) = \begin{cases} 2 + \varepsilon, \varepsilon > 0 & \text{for } r = 2 \\ r \left(\min_{x>0} \left\{ \frac{x}{(1-e^{-x})^{r-1}} \right\} \right)^{-1} & \text{for } r > 2, \end{cases} \tag{3.4}$$

then the acyclic random r-graphs dominate the space of random r-graphs. The value $c(3) \approx 1.23$ is a minimum value for Eq. (3.4), as shown in Figure 3.5. This implies that the 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 $1,000$ 3-partite random hypergraphs with $n = 10^7$ keys (hyperedges).

It is interesting to remark that the problems of generating 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 by Majewski et al [42] for general hypergraphs.

Finally, as Pr_a is $\Omega(1)$, the expected number of iterations is $O(1)$, and therefore it is safe to conclude that the mapping step takes $O(n)$ expected time because the acyclicity

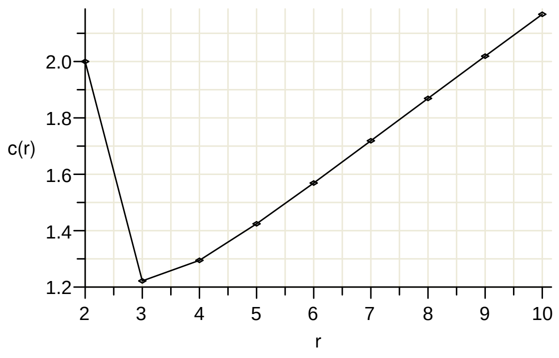


Figure 3.5: Values of $c(r)$ for $r \in \{2, 3, \ldots, 10\}$, previously reported in [42]

test also runs in $O(n)$ time.

3.1.2 Assigning Step

The Assigning step takes the acyclic random hypergraph G_r and the list of edges L as input, and produces the labeling g. To assign values to the vertices of G_r we traverse the edges in L from tail to head. 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 L in the following order: $e_1, e_2, \ldots, e_i, e_{i+1}, \ldots, e_n$ and we consider the 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 vertice 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 for the first time and such a vertex can be used to uniquely represent the edge.

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 $e \in L$ from tail to head, we look for the first vertex u belonging tfig:mapo 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 3.6 presents a step by step example for the list of edges of our example presented in Figure 3.3. The initial state is shown in Figure 3.6 (a). In Figure 3.6 (b), the vertices 2 and 5 of edge $L[3]$ are marked as visited and $g[2] = (0 - g[5]) \mod 2 = 0$. In Figure 3.6 (c), the vertex 7 of edge $L[2]$ is marked as visited and $g[7] = (1-g[2]) \mod 2 = 1$. In Figure 3.6 (d), the vertex 6 of edge $L[1]$ is marked as visited and $g[6] = (1-g[2]) \mod 2 = 1$. Finally, in Figure 3.6 (e), the vertices 0 and 5 of edge $L[0]$ are marked as visited and $g[0] = (0-g[5]) \mod 2 = 0$. Therefore, Eq. (3.1) will return the vertices $2, 6, 0$ and 7 for the keys jan, feb, mar and apr, respectively.

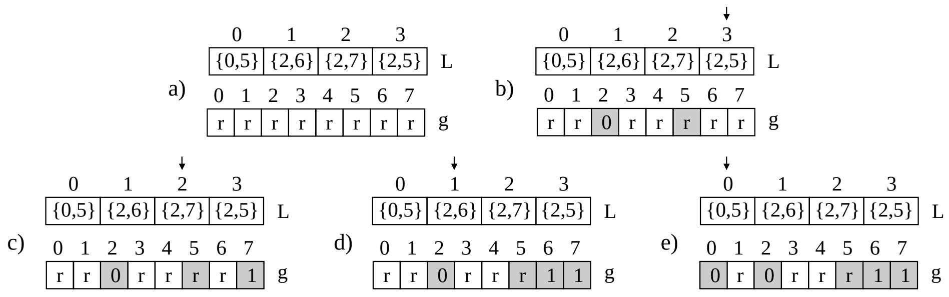


Figure 3.6: Example of the assigning step

Figure 3.7 presents a pseudo code for the Assigning step.

```
procedure Assigning ($G_r$, $L$, $g$)
    for $u = 0$ to $m-1$ do
        visited[$u$] = false;
        $g[u]$ = $r$;
    for i = $|L|-1$ to 0 do
        $e$ = 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 3.7: Assigning step

Analysis of the Assigning Step

As each edge is handled once, the number of vertices in G_r is a linear function of n (i.e., the number of edges), and the involved operations have cost $O(1)$, then the Assigning step also runs in $O(n)$ time.

3.1.3 Ranking Step

The Ranking Step obtains MPHFs from the PHFs presented in Eq. (3.1). It receives the labeling g as input and produces the data structure rankTable as output. It is possible to build a data structure that allows the computation in constant time of function rank presented in Eq. (3.3) by using $o(m)$ additional bits of space. This is a well-studied primitive in succinct data structures (see e.g. [47]).

Implementation

We now describe a practical variant that uses $\epsilon\, m$ additional bits of space, where ϵ can be chosen as any positive number, to compute the data structure rankTable in linear time. 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 the rankTable the number of assigned vertices before every 256th entry in the labeling g. Figure 3.8 presents a pseudo code for the Ranking step.

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

Figure 3.8: Ranking step

Figure 3.9 illustrates the Ranking step on the labeling g of Figure 3.6 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 $g[6]$.

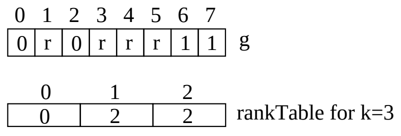


Figure 3.9: Example of the Ranking step

Analysis of the Ranking Step

The Ranking step also runs in time $O(n)$. This comes from the fact that it just loops over the $m = c(r)n$ entries of labeling g and $c(r)$ is a constant fixed a priori.

3.1.4 Evaluating the Resulting Functions

To compute rank(u), where u is given by Eq. (3.1), we look up in the 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 $b = \epsilon \log m$ bits in constant time for any $0 < \epsilon < 1$. For simplicity and without loss of generality we let b 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 3.10, where LS(i',β) stands for the value of the β least significant bits of i' and >> is the right shift of bits operation.

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

Figure 3.10: 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 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 3.1.1). 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 3.3.1 and 3.3.2.

Figure 3.11 presents the pseudo code for the resulting PHFs. Note that it is quite simple to be computed, an important characteristic at retrieval time.

Figure 3.12 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 $b = \beta\mathcal{E}$ bits. We use the notation

```
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 3.11: Pseudo code for the PHF presented in Eq. (3.1)

$g(i \to j)$ 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.

```
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 += $T_r[g(i \to i + \mathcal{E})]$;
    return rank;
```

Figure 3.12: Pseudo code for the MPHF presented in Eq. (3.2)

3.2 The Uniform Hashing Assumption

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, exceeding the space for the MPHFs. From a theoretical perspective, the full randomness assumption is not too harmful, as we can use the "split and share" approach of Dietzfelbinger and Weidling [22]. The additional space usage is then a lower order term of $O(n^{1-\Omega(1)})$. Specifically, the algorithm would split S into $O(n^{1-\delta})$ buckets of size n^{δ}, where $\delta < 1/3$, say, and create a perfect hash function for each bucket using a pool of $O(r)$ simple hash functions of size $O(n^{2\delta})$, where each acts like uniform random functions on each bucket, with high probability. From this pool, we can find r suitable functions for each bucket, with high probability. Putting everything together to form a perfect hash function for S can be done using an offset table of size $O(n^{1-\delta})$. This is the main idea supporting the analysis of the algorithm presented in Chapter 4.

Implementation

In practice, limited randomness is often as good as total randomness [52]. For our experiments we choose h_i from a family $\mathcal{H}$ of universal hash functions presented in [18], and we verify experimentally that the schemes behave well (see Section 3.4). The functions h_i, for $i \in [0, r-1]$, are constructed as follows. We impose some upper bound L on the lengths of the keys in S that are generated from an alphabet of size $|\Sigma|$. To define h_i, we generate an $L \times |\Sigma|$ table of random integers $table_i$ in the range $[0, m-1]$. For a key $x \in S$ of length $|x| \le L$, we let

$$h_i(x) = \left(\sum_{j=0}^{|x|-1} table_i[j, x[j]] \right) \bmod \tfrac{m}{r} + i\tfrac{m}{r}.$$

Each different table $table_i$ corresponds to a different hash function. Thus, the storage space requirement for each hash function h_i is $L \times |\Sigma| \times \log m$ bits.

There are other heuristic hash functions with very good performance in practice but with no theoretical foundation. For instance, the one proposed by Jenkins [37], where there is no upper bound for the key sizes and its description requires just the storage of an integer that is used as seed for a pseudo random number generator. The function just loops around the key doing bitwise operations on blocks of 12 bytes and, in the end, a 12 byte long integer is generated, which can be partially or integrally taken module $\frac{m}{r}$. In Chapter 4 we show experiments illustrating the practicality of the Jenkins function.

3.3 Storage Requirements for the Resulting Functions

In this section we present the space required to store the resulting MPHFs disregarding the space for storing the r uniform hash functions, which was discussed in Section 3.2.

The description of the resulting MPHFs is compounded by the function g, the rankTable and the lookup table T_r. The resulting labeling 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 $\beta = \lfloor \log(r) \rfloor + 1$ 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.

3.3.1 The 2-Graph Instance

The use of 2-graphs allows us to generate the PHFs of Eq.(3.1) that give values in the range $[0, m-1]$, where $m = (2+\varepsilon)n$ for $\varepsilon > 0$ (see Section 3.1.1). The significant values in the labeling g for a PHF are $\{0, 1\}$, because we do not need to represent information to calculate the ranking (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.

To generate the MPHFs of Eq. (3.2) 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.

Improving the space

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 construction 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 $u \in V(G_r)$ we use:

```
$byte \;\; = g(u/5)$;
$byte \;\; += x * pow3\_table[u \bmod 5]$;
$g(u/5) = byte$;
```

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 $T_{lookup}[i][j] = (j/pow3\_table[i]) \bmod 3$, where $0 \le i < 5$ and $0 \le j < 256$. In the experiments, for

$\epsilon = 0.125$ and $\varepsilon = 0.09$, the resulting functions are stored in approximately $3.6n$ bits.

3.3.2 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 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 3.1.1). Therefore, we will be able to generate the PHFs of Eq. (3.1) 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. (3.2) 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 (see Section 3.4).

Improving the space

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. (3.1) are $\{0, 1, 2\}$. Then, we can apply the arithmetic coding technique presented in Section 3.3.1 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.

3.4 Experimental Results

In this section we evaluate the performance of our algorithms. We compare them with the main practical minimal perfect hashing algorithms we found in the literature. They are: Botelho, Kohayakawa and Ziviani [8] (referred to as BKZ), Fox, Chen and Heath [28] (referred to as FCH), Majewski, Wormald, Havas and Czech [42] (referred to as MWHC), and Pagh [46] (referred to as PAGH). For the MWHC algorithm we used the version based on 3-graphs. We did not consider the one that uses 2-graphs because it is shown in [8] and in Chapter 2 that the BKZ algorithm outperforms it. We used the hash functions presented in Section 3.2 for all the algorithms.

The algorithms were implemented in the C language and are available at `http://cmph.sf.net` under the GNU Lesser General Public License (LGPL). The experiments were carried out on a computer running the Linux operating system, version 2.6, with a 3.2 gigahertz Intel Xeon Processor with a 2 megabytes L2 cache and 1 gigabyte of main memory. Each experiment was run for 100 trials. For the experiments we used two collections: (i) a set of randomly generated 4 bytes long IP addresses, and (ii) a set of 64 bytes long (on average) URLs collected from the Web.

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

We now compare our algorithms for constructing MPHFs with the other algorithms considering generation time and storage space. Table 3.1 shows that our algorithm for $r = 3$ and the MWHC algorithm are faster than the others to generate MPHFs. The storage space requirements for our algorithms with $r = 2$, $r = 3$ and the FCH algorithm are 3.6, 2.62 and 3.66 bits per key, respectively. For the BKZ, MWHC and PAGH algorithms they are $\log n$, $1.23 \log n$ and $2.03 \log n$ bits per key, respectively.

Algorithms		Generation Time (sec)		Storage Space	
		URLs	IPs	Bits/Key	Size (MB)
Our	$r = 2$	19.49 ± 3.750	18.37 ± 4.416	3.60	1.52
	$r = 3$	9.80 ± 0.007	8.74 ± 0.005	2.62	1.11
BKZ		16.85 ± 1.85	15.50 ± 1.19	21.76	9.19
FCH		5901.9 ± 1489.6	4981.7 ± 2825.4	3.66	1.55
MWHC		10.63 ± 0.09	9.36 ± 0.02	26.76	11.30
PAGH		52.55 ± 2.66	47.58 ± 2.14	44.16	18.65

Table 3.1: Comparison of the algorithms for constructing MPHFs considering generation time and storage space, and using $n = 3,541,615$ for the two collections

Now we compare the algorithms considering evaluation time. Table 4.5 shows the 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 [46] (it performs only 1 memory probe), it is important to note in this experiment that the evaluation time is smaller for the FCH and our algorithms because the generated functions fit entirely in the L2 cache of the machine (see the storage space size for our algorithms and the FCH algorithm in Table 3.1). Therefore, the more compact a MPHF is, the more efficient it is if its description fits in the cache. For example, for sets of size up to 6.5 million keys of any type the resulting functions generated by our algorithms will entirely fit in a 2 megabyte L2 cache.

Algorithms		Our		BKZ	FCH	MWHC	PAGH
		$r = 2$	$r = 3$				
Evaluation	IPs	1.35	1.36	1.45	1.01	1.46	1.43
Time (sec)	URLs	2.63	2.73	2.81	2.14	2.85	2.78

Table 3.2: Comparison of the algorithms considering evaluation time and using the collections IPs and URLs with $n = 3,541,615$

Now, we compare the PHFs and MPHFs generated by our family of algorithms considering generation time, storage space and evaluation time. Table 3.3 shows that the generation times for PHFs and MPHFs are almost the same, being the algorithms for $r = 3$ more than twice faster because the probability to obtain an acyclic 3-graph for $c(3) = 1.23$ tends to one while the probability for a $2 - graph$ where $c(2) = 2.09$ tends to 0.29 (see Section 3.1.1). For PHFs with $m = 1.23n$ instead of MPHFs with $m = n$, then the space storage requirement drops from 2.62 to 1.95 bits per key. The PHFs with $m = 2.09n$ and $m = 1.23n$ are the fastest ones at evaluation time because no ranking or packing information needs to be computed.

Finally, In a conversely 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 3.4.

3.5 Conclusions

We have presented an efficient family of algorithms to generate near space-optimal PHFs and MPHFs. The algorithms are simpler and has much lower constant factors than existing theoretical results for $n < 2^{300}$. In addition, it outperforms the main practical general

r	Packed	m	Generation Time (sec)		Eval. Time (sec)		Storage Space	
			IPs	URLs	IPs	URLs	Bits/Key	Size (MB)
2	no	$2.09n$	18.32 ± 3.352	19.41 ± 3.736	0.68	1.83	2.09	0.88
2	yes	n	18.37 ± 4.416	19.49 ± 3.750	1.35	2.63	3.60	1.52
3	no	$1.23n$	8.72 ± 0.009	9.73 ± 0.009	0.96	2.16	2.46	1.04
3	yes	$1.23n$	8.75 ± 0.007	9.95 ± 0.009	0.94	2.14	1.95	0.82
3	no	n	8.74 ± 0.005	9.80 ± 0.007	1.36	2.73	2.62	1.11

Table 3.3: 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 3.3.1 and 3.3.2

Algorithms		Our			BKZ	FCH	MWHC	PAGH
Function Type		Comp. PHF	PHF	MPHF	MPHF	MPHF	MPHF	MPHF
Size (megabytes)		3.52	4.40	4.95	42.63	-	52.44	86.54
Evaluation Time (sec)	IPs	5.94	5.75	8.02	4.86	-	6.29	4.60
	URLs	9.41	9.30	11.49	9.29	-	9.61	9.25

Table 3.4: Comparison of the algorithms considering evaluation time and using the collections IPs and URLs with $n = 15,000,000$

purpose algorithms found in the literature considering generation time and storage space as metrics. However, the resulting MPHFs still assume uniform hashing. Our next result presented in Chapter 4 shows how to generate simple and near space-optimal MPHFs without assuming uniform hashing.

Chapter 4

A scalable minimal perfect hashing method

In this chapter we use a number of techniques from the literature to obtain a novel external memory based perfect hashing algorithm, referred to as *EPH algorithm*. The EPH algorithm produces MPHFs using approximately 3.8 bits per key. Also, for PHFs with range $\{0, \ldots, 2n-1\}$ the space usage drops to approximately 2.7 bits per key. The main insight supporting the EPH algorithm is that it splits the incoming key set S into small buckets containing at most 256 keys. Then, a MPHF is generated for each bucket and using an *offset* array we obtain a MPHF for S. Therefore, the EPH algorithm works on subsets of size lower than 256 and this increases the probability of cache hits. That is why the EPH algorithm generates the functions as fast as the algorithms that operates only on data structures stored in internal memory.

The EPH algorithm increases one order of magnitude in the size of the greatest key set for which a MPHF was obtained in the literature [8]. This improvement comes from a combination of a novel perfect hashing scheme that greatly simplifies previous methods, and the fact that the EPH algorithm is designed to make good use of memory hierachy (see Section 4.3.2 for details). Also, the algorithm is theoretically sound because with the help of Rasmus Pagh [7] we have completely analyzed it's time and space usage without unrealistic assumptions.

We demonstrate the scalability of the EPH 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 62 minutes. If we use the range $\{0, \ldots, 2n-1\}$, 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. Thus we believe our MPHF method might be quite useful for a number of current and practical data management problems.

4.1 The EPH algorithm

Our algorithm uses the well-known idea of partitioning the key set into a number of small sets[1] (called "buckets") using a hash function h_0. Let $B_i = \{x \in S \mid h_0(x) = i\}$ denote the ith bucket. If we define $\mathit{offset}[i] = \sum_{j=0}^{i-1} |B_i|$ and let mphf_i denote a MPHF for B_i then clearly

$$\mathrm{mphf}(x) = \mathrm{mphf}_i(x) + \mathit{offset}[h_0(x)] \qquad (4.1)$$

is a MPHF for the whole set S. Thus, the problem is reduced to computing and storing the offset array, as well as the MPHF for each bucket.

Figure 4.1 illustrates the two steps of the EPH algorithm: *the partitioning* step and *the searching step*. The partitioning step takes a key set S and uses a hash function h_0 to partition S into 2^b buckets. The searching step generates a MPHF mphf_i for each bucket i, $0 \le i \le 2^b - 1$ and computes the offset array. To compute the MPHF of each bucket we used one algorithm from the family of algorithms presented in Chapter 3. We will describe the algorithm in more details in Section 4.1.2 because we have tuned it to generate more compact and faster functions at retrieval time.

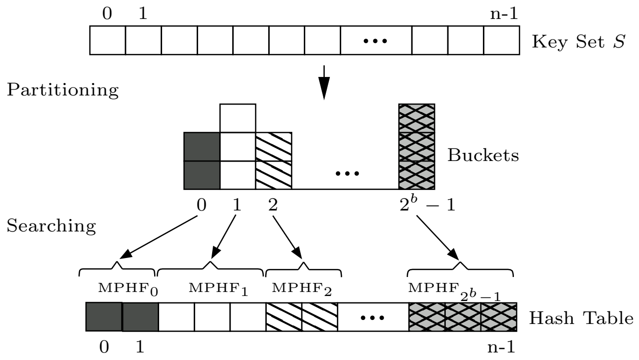


Figure 4.1: Main steps of the EPH algorithm

We will choose h_0 such that it has values in $\{0,1\}^b$, for some integer b. Since the offset array holds 2^b entries of at least $\log n$ bits we want 2^b to be less than around $n/\log n$, making the space used for the offset array negligible. On the other hand, to allow efficient

[1]Used in e.g. the perfect hash function constructions of Schmidt and Siegel [52] and Hagerup and Tholey [32], for suitable definition of "small".

implementation of the functions mphf_i we impose an upper bound ℓ on the size of any bucket. We will describe later how to choose h_0 such that this upper bound holds.

To create the MPHFs mphf_i we could choose from a number of alternatives, emphasizing either space usage, construction time, or evaluation time. We show that all methods based on the assumption of uniform hash functions can be made to work, with explicit and provably good hash functions. For the experiments we have implemented the algorithm described in Section 4.1.2. Since this computation is done on a small set, we can expect nearly all memory accesses to be "cache hits". We believe that this is the main reason why our method performs better than previous ones that access memory in a more "random" fashion.

We consider the situation in which the set of all keys may not fit in the internal memory and has to be written on disk. The EPH algorithm first scans the list of keys and computes the hash function values that will be needed later on 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 to large key sets, this is done using an implementation of an external memory mergesort [40]. If the merge sort works in two phases, which is the case for all reasonable parameters, the total work on the 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.

The detailed description of the EPH algorithm is presented in Section 4.1.1. The internal algorithm used to compute the MPHF of each bucket is presented in Section 4.1.2. The internal algorithm uses two hash functions h_{i1} and h_{i2} to compute a MPHF mphf_i. These hash functions as well as the hash function h_0 used in the partitioning step of the EPH algorithm are described in Section 4.1.3.

4.1.1 Implementation of the EPH algorithm

In this section we are going to present the implementation of the two-step external memory based algorithm and the values of the parameters related to the algorithm. The EPH algorithm is essentially a two-phase multi-way merge sort with some nuances to make it work in linear time.

The partitioning step performs two important tasks. First, the variable-length keys are mapped to 128-bit strings by using the linear hash function h' presented in Section 4.1.3. That is, the variable-length key set S is mapped to a fixed-length key set F. Second, the set S of n keys is partitioned into 2^b buckets, where b is a suitable parameter chosen to guarantee that each bucket has at most $\ell = 256$ keys with high probability (see Section 4.1.3). We have two reasons for choosing $\ell = 256$. The first one is to keep the buckets size small enough to be represented by 8-bit integers. The second one is to allow the memory accesses during the MPHF evaluation to be done in the cache most of the time. Figure 4.2 presents the partitioning step algorithm.

```
▶ 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 $N = \lceil \beta/\mu \rceil$ be the number of key blocks that will be read from disk into an
  internal memory area
1. **for** $j = 1$ **to** $N$ **do**
   1.1 Read a key block $S_j$ from disk (one at a time) and store $h'(x)$, for each $x \in S_j$,
       into $\mathcal{B}_j$, where $|\mathcal{B}_j| = \mu$
   1.2 Cluster $\mathcal{B}_j$ into $2^b$ 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 $h'(x)$)
   1.3 Dump $\mathcal{B}_j$ to the disk into File $j$
```

Figure 4.2: 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 by Larson and Graefe [40]. That is, the last block is kept in memory instead of dumping it to disk to be read again in the second step of the algorithm.

Figure 4.3(a) shows a *logical* view of the 2^b buckets generated in the partitioning step. In reality, the 128-bit strings 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 128-bit strings in bucket 0 appear in files 1 and N, the 128-bit strings in bucket 1 appear in files 1, 2 and N, and so on.

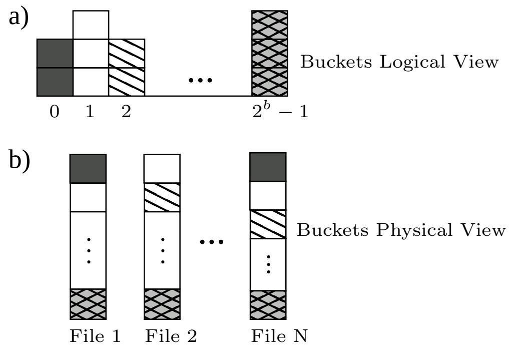


Figure 4.3: Situation of the buckets at the end of the partitioning step: (a) Logical view (b) Physical view

This scattering of the 128-bit strings in the buckets could generate a performance problem because of the potential number of seeks needed to read the 128-bit strings in each bucket from the N files on disk during the second step. But, as we show later on in Section 4.3.3, the number of seeks can be kept small using buffering techniques.

The searching step is responsible for generating a MPHF for each bucket and for computing the offset array. Figure 4.4 presents the searching step algorithm. Statement 1 of Figure 4.4 constructs the heap H of size N. This is well known to be linear on N. The order relation in H is given by the bucket address i (i.e., the b most significant bits of $x \in F$). Statement 2 has two important steps. In statement 2.1, a bucket is read from disk, as described below. In statement 2.2, a MPHF is generated for each bucket B_i using the internal memory based algorithm presented in Section 4.1.2. In statement 2.3, the next entry of the offset array is computed. Finally, statement 2.4 writes the description of MPHF_i and $\mathit{offset}[i]$ to disk. Note that to compute $\mathit{offset}[i+1]$ we just need the current bucket size and $\mathit{offset}[i]$. So, we just need to maintain two entries of vector *offset* in memory all the time.

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.

```
▶ Let $H$ be a minimum heap of size $N$, where the
  order relation in $H$ is given by
  $i = x[96, 127] >> (32 - b)$ for $x \in F$
1. for $j = 1$ to $N$ do { Heap construction }
   1.1 Read the first 128-bit string $x$ from File $j$ on disk
   1.2 Insert $(i, j, x)$ in $H$
2. for $i = 0$ to $2^b - 1$ do
   2.1 Read bucket $B_i$ from disk driven by heap $H$
   2.2 Generate a MPHF for bucket $B_i$
   2.3 $\mathit{offset}[i + 1] = \mathit{offset}[i] + |B_i|$
   2.4 Write the description of MPHF$_i$ and $\mathit{offset}[i]$
       to the disk
```

Figure 4.4: Searching step

Statement 1.1 extracts and removes triple (i, j, x) from H, where i is a minimum value in H. Statement 1.2 inserts x in bucket B_i. Statement 1.3 performs a seek operation in File j on disk for the first read operation and reads sequentially all 128-bit strings $x \in F$ that have the same index i and inserts them all in bucket B_i. Finally, statement 1.4 inserts in H the triple (i', j, x'), where $x' \in F$ is the first 128-bit string read from File j (in statement 1.3) that does not have the same bucket address as the previous keys.

```
1. while bucket $B_i$ is not full do
   1.1 Remove $(i, j, x)$ from $H$
   1.2 Insert $x$ into bucket $B_i$
   1.3 Read sequentially all 128-bit strings from File $j$
       that have the same $i$ and insert them into $B_i$
   1.4 Insert the triple $(i', j, x')$ in $H$, where $x'$ is
       the first 128-bit string read from File $j$ that
       does not have the same bucket index $i$
```

Figure 4.5: Reading a bucket

4.1.2 The algorithm used for the buckets

For the buckets we decided to use one of the algorithms from the family $\mathcal{F}$ of algorithms[2] presented in Chapter 3, because it outperforms the main practical algorithms we found in the literature (see Section 1.3), and also is a simple and near space-optimal way of constructing a minimal perfect hash function for a set S of n elements. The functions are constructed under the assumption that it is possible to create and access two truly random hash functions $f_0 : U \to [0, \frac{m}{2} - 1]$ and $f_1 : U \to [\frac{m}{2}, m - 1]$, where $m = cn$ for $c > 2$. The Functions f_0 and f_1 are used to map the keys in S to a bipartite graph $G = (V, E)$, where $V = [0, m - 1]$ and $E = \{\{f_0(x), f_1(x)\} \mid x \in S\}$ (i.e, $|E| = |S| = n$). Hence, each key in S is associated with only one edge from E. Figure 4.6(a) illustrates this step, referred to as *mapping step*, for a set S with three keys.

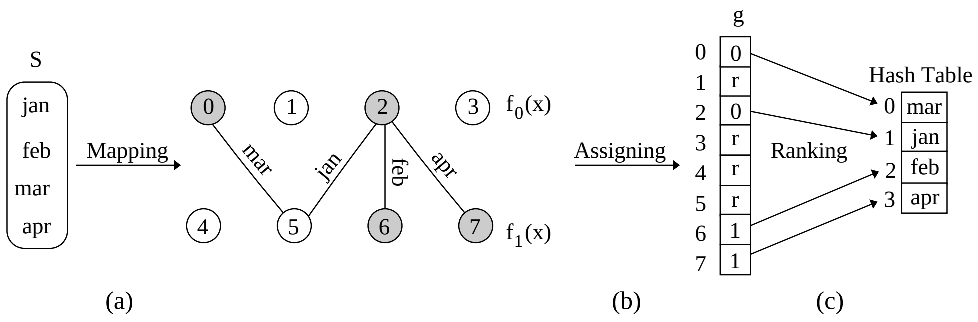


Figure 4.6: (a) Mapping step generates a bipartite graph (b) Assigning step generates a labeling g so that each edge is uniquely associated with one of its vertices (c) Ranking step builds a function rank : $V \to [0, n - 1]$

In the following step, referred to as *assigning step*, a function $g : V \to \{0, 1, 2\}$ is computed so that each edge is uniquely represented by one of its vertices. For instance, in Figure 4.6(b), edge $\{0, 3\}$ is associated with vertex 0, edge $\{0, 4\}$ with vertex 4 and edge $\{2, 4\}$ with vertex 2. Then, a function phf : $S \to V$ defined as $\text{phf}(x) = f_i(x)$, where $i = i(x) = (g(f_0(x)) + g(f_1(x))) \bmod 2$ is a perfect hash function on S. Note that $i = i(x)$ is used to select the vertex from the edge associated with x that uniquely represent x

The assigning step splits the set of vertices into two subsets: (i) the *assigned ones* and (ii) the *unassigned ones*. A vertex v is defined as assigned if $g(v) \neq 2$ and unassigned otherwise. Also, the number of assigned vertices is guaranteed by construction to be equal

[2]The algorithms in $\mathcal{F}$ use r-uniform random hypergraphs given by function values of r hash functions on the keys of S. An r-graph is a generalization of a standard graph where each edge connects $r \geq 2$ vertices. For the buckets in the EPH algorithm we used the 2-uniform hypergraph instance.

to $|S| = n$. Therefore, a function rank : $V \to [0, n-1]$ that counts how many vertices are assigned before a given assigned vertex $v \in V$ is a MPHF on V. For example, in Figure 4.6(c), rank(0) = 0, rank(2) = 1 and rank(4) = 2, which means that there is no assigned vertex before vertex 0, one before vertex 2, and there are two before vertex 4. This implies that a function $h : S \to [0, n-1]$ defined as mphf(x) = rank(phf(x)) is a MPHF on S. The last step of the algorithm, referred to as *ranking step*, is responsible for computing the data structures used to compute function rank in time $O(1)$.

In Chapter 3 we have shown that g can be generated in linear time if the bipartite graph $G = G(f_0, f_1)$ is acyclic. When a cyclic graph is generated a new pair (f_0, f_1) is randomly selected so that the values of f_0 and f_1 are truly random and independent. Hence the number of iterations to get an acyclic random graph must be bounded by a constant to finish the algorithm in linear time. They have shown that if $|V| = m = cn$, for $c > 2$, the probability of generating an acyclic bipartite random graph is $Pr_a = \sqrt{1 - (2/c)^2}$ and the number of iterations is on average $N_i = 1/Pr_a$. For $c = 2.09$ we have $Pr_a \approx 0.29$ and $N_i \approx 3.4$. Finally, we have used arithmetic coding to compress g and the data structures used to compute function rank in time $O(1)$ so that the resulting MPHFs are stored in $(3 + \epsilon)n$, where $\epsilon \approx 0.6$.

Improving the space

We now present other technique that will produce MPHFs slightly more compact. The first observation is that to compute phf(x) we do not need the value r used to represent unassigned vertices. Then, we replace it to 0 and now each entry of vector g is encoded with one bit. To compute mphf(x) we need to know what are the assigned vertices. Then, we use other bit vector T of size m to indicate the assigned vertices. That is, $T[v] = 1$ if $v \in V$ is assigned and $T[v] = 0$ otherwise. In this case we would require $2m + o(n)$ bits to store the resulting MPHFs. The $o(n)$ part comes from the fact that we need to store information to compute the rank (see Section 3.3 for details).

Now we can create a compressed representation g' that uses only n bits and enables us to compute any bit of g in constant time by using rank on the set of assigned vertices represented by T. That is, $g'[\text{rank}(v)] = g[v]$. This is possible since rank(v) is 1-1 on elements in $V(G)$, which are mapped into the range $[0, n-1]$. In conclusion, we can replace g by g' and reduce the space usage to $n + m + o(n)$ bits. As $m = (2 + \epsilon)n$ for any $\epsilon > 0$, then the resulting MPHFs are now stored in $(3 + \epsilon)n + o(n)$ bits, for any $\epsilon > 0$. In the experiments we have got functions that are stored in approximately $3.3n$ bits.

The parameters choice for the algorithm used for the buckets

The first parameter we are going to discuss is that c responsible for allowing us to construct an acyclic bipartite random graph with high probability. We have set c to 2.09 in order to get a probability of approximately 0.29 of generating a random graph with no cycles. As a consequence the expected number of iterations to generate an acyclic graph is approximately 3 (see Section 3.1.1 for details).

The larger is the value of c, the sparser is the random graph used and, consequently, the larger is the storage requirements of the resulting MPHFs and the faster is the algorithm because of the greater probability of getting an acyclic random graph. We have chosen a small value for c because we are interested in more compact functions and the runtime of the internal algorithm is dominated by the time spent with I/Os.

4.1.3 Hash functions used by the EPH algorithm

The aim of this section is threefold. First, in Section 4.1.3, we define the hash function h_0 used to split the key set S into 2^b buckets and the hash functions h_{i1} and h_{i2} used by the algorithm to generate the MPHF of each bucket, where $0 \leq i \leq 2^b - 1$. Second, in Section 4.1.3, we present the implementation details of those hash functions. Third, in Section 4.2.3, we show the conditions that parameter b must meet so that no bucket with more than ℓ keys is created by h_0. We also show that h_{i1} and h_{i2} are truly random hash functions for the buckets.

Definitions

We have made the design decision to make use of tabulation based hash functions, which seem to be a more practical alternative than hash functions based on integer multiplication of keys.[3] We will make extensive use of the linear hash functions analyzed by Alon, Dietzfelbinger, Miltersen and Petrank [1]. To do so we consider a key as a 0-1 vector of length L. The variable-length strings that we consider are conceptually made fixed length by padding with zeros at the end, which results in a unique vector since ascii character 0 does not appear in any string.

[3]For example, as far as we know the best way of implementing multiplication of two 64-bit integers on contemporary machines is by "school method" reduction to 4 multiplications of 32-bit integers. Similarly, a " mod p" operation on the resulting 128-bit integer, where p is a 32-bit integer, seems to require 3 multiplications of 32-bit integers and 4 modulo operations on 64-bit integers.

Mathematically, h_0 is a randomly chosen linear map over Galois field 2 (or simply GF(2)) from $\{0,1\}^L$ to $\{0,1\}^b$. To get an efficient implementation, we use a tabulation idea from [2] where we can get evaluation time $O(L/\log\sigma)$ by using space $L\sigma$ – see Section 4.1.3 for implementation details. Choosing $\sigma = n^{\Omega(1)}$ we obtain evaluation time $O(L/\log n)$. (In theory we could get evaluation time $O(L/w)$, where $w \geq \log n$ is the word length of the computer, by first hashing down to $O(\log n)$ bits using universal hashing; however, this does not seem to give an improvement in practice.) We choose b as small as possible such that the maximum bucket size is bounded by ℓ with reasonable probability (some constant close to 1). By a result of [1] we know that

$$b \leq \log n - \log(\ell/\log\ell) + O(1). \tag{4.2}$$

For the implementation, we will experimentally determine the smallest possible choices of b.

To define h_{i1} and h_{i2} we proceed as follows. Again use the linear hash function of [1] to implement hash functions $y_1, \ldots, y_k$ from $\{0,1\}^L$ to $\{0,1\}^{r-1}0$, where $r \gg \log\ell$ and k are parameters to be determined later. Note that the range is the set of r-bit strings ending with a 0. The purpose of the last 0 is to ensure that we can have no collision between $y_j(x_1)$ and $y_j(x_2) \oplus 1$, $1 \leq j \leq k$, for any pair of elements x_1 and x_2. Let p be a prime number much larger than the size of the desired range of h_{i1} and h_{i2}, which in our case is $|B_i|$, and let $t_1, \ldots, t_{2k}$ be tables of 2^r random values in $\{0, \ldots, p-1\}$. We then define:

$$\begin{aligned}
\rho(x,s,\Delta) &= \left(\sum_{j=1}^{k} t_j[y_j(x)\oplus\Delta] + s\sum_{j=k+1}^{2k} t_j[y_j(x)\oplus\Delta]\right) \bmod p\\
h_{i1}(x) &= \rho(x,s_i,0) \bmod |B_i|\\
h_{i2}(x) &= \rho(x,s_i,1) \bmod |B_i|
\end{aligned} \tag{4.3}$$

where s is a random integer seed number and the symbol $\oplus$ denotes exclusive-or and the variable s_i is specific to bucket i. To find s_i we choose random values from $\{1, \ldots, p-1\}$ until the functions h_{i1} and h_{i2} work with the internal algorithm of Section 4.1.2. It is known that a constant fraction of the set of all functions work; in Section 4.2.3 we will argue that this will also be the case when the hash functions are chosen as above.

Implementation details

In order to implement the functions h_0, $y_1, y_2, y_3, \ldots, y_k$ to be computed at once we use a function h' from a family of linear hash functions over GF(2) proposed by Alon, Dietzfelbinger, Miltersen and Petrank [1]. The function has the following form: $h'(x) = Ax$, where $x \in S$ and A is a $\gamma \times L$ matrix in which the elements are randomly chosen from $\{0, 1\}$. The output is a bit string of an a priori defined size γ. In our implementation $\gamma = 128$ bits. It is important to realize that this is a matrix multiplication over GF (2). The implementation can be done using a bitwise-and operator (&) and a function $f : \{0, 1\}^\gamma \to \{0, 1\}$ to compute parity instead of multiplying numbers. The parity function $f(a)$ produces 1 as a result if $a \in \{0, 1\}^\gamma$ has an odd number of bits set to 1, otherwise the result is 0. For example, let us consider $L = 3$ bits, $\gamma = 3$ bits, $x = 110$ and

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

The number of rows gives the required number of bits in the output, i.e., $\gamma = 3$. The number of columns corresponds to the value of L. Then,

$$h'(x) = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

where $b_1 = f(101 \ \& \ 110) = 1$, $b_2 = f(001 \ \& \ 110) = 0$ and $b_3 = f(110 \ \& \ 110) = 0$.

To get a fast evaluation time, some tabulation is required. Note that if x is short, e.g. 8 bits, we can simply tabulate all the function values and compute $h'(x)$ by looking up the value $h'[x]$ in an array h'. To make the same thing work for longer keys, split the matrix A into parts of 8 columns each: $A = A_1|A_2|\dots|A_{\lceil L/8 \rceil}$, and create a lookup table h'_i for each submatrix. Similarly split x into parts of 8 bits, $x = x_1x_2\dots x_{\lceil L/8 \rceil}$. Now $h'(x)$ is the exclusive-or of $h'_i[x_i]$, for $i = 1, \dots, \lceil L/8 \rceil$. Therefore, we have set σ to 256 so that keys of size L can be processed in chunks of $\log \sigma = 8$ bits. In our URL collection the largest key has 65 bytes, i.e., $L = 520$ bits.

The 32 most significant bits of $h'(x)$, where $x \in S$, are used to compute the bucket address of x, i.e., $h_0(x) = h'(x)[96, 127] >> (32 - b)$. We use the symbol $>>$ to denote the right shift of bits. The other 96 bits correspond to $y_1(x), y_2(x), \dots y_6(x)$, taking $k = 6$. This would give $r = 16$, however, to save space for storing the tables used for computing h_{i1} and h_{i2}, we hard coded the linear hash function to make the most and the least significant bit of each chunk of 16 bits equal to zero. Therefore, $r = 15$. This setup enable us to solving problems of up to 500 billion keys, which is plenty of for all the applications we know of. If our algorithm fails in any phase, we just restart it. As the parameters are chosen to have success with high probability, the number of reinitializations is $O(1)$.

Finally, the last parameter related to the hash functions we need to talk about is the prime number p. As p must be much larger than the range of h_{i1} and h_{i2}, then we set it to the largest 32-bit integer that is a prime, i.e, $p = 4294967291$.

4.2 Analytical results

The purpose of this section is threefold. First, we show that our algorithm runs in expected time $O(n)$. Second, we present the main memory requirements for constructing the MPHF. Third, we sketch the analysis of the hash functions used by the EPH algorithm.

4.2.1 The linear time complexity

First, we show that the partitioning step presented in Figure 4.2 runs in $O(n)$ time. Each iteration of the loop **for** in statement 1 runs in $O(|\mathcal{B}_j|)$ time, $1 \leq j \leq N$, where $|\mathcal{B}_j|$ is the number of 128-bit strings that fit in block $\mathcal{B}_j$ of size μ. This is because statement 1.1 just reads $|\mathcal{B}_j|$ keys from disk and stores them all into the internal memory area of size μ, statement 1.2 runs a radix sort algorithm that is well known to be linear in the number of keys it sorts (i.e., $|\mathcal{B}_j|$ 128-bit strings), and statement 1.3 just dumps $|\mathcal{B}_j|$ 128-bit strings to the disk into File j. Thus, the loop **for** runs in $\sum_{j=1}^{N} O(|\mathcal{B}_j|)$ time. As $\sum_{j=1}^{N} |\mathcal{B}_j| = n$, then the partitioning step runs in $O(n)$ time.

Second, we show that the searching step presented in Figure 4.4 also runs in $O(n)$ time. Let us firstly analyse the number of heap operations performed in statement 2.1 that reads $|B_i|$ 128-bit strings of bucket B_i and is detailed in Figure 4.5. It's well known that the heap construction of statement 1 runs in $O(N)$ time. Each iteration of statement 2 performs two heap operations in statement 2.1 (see statements 1.1 and 1.4 in Figure 4.5) and each one costs $O(\log N)$. So, the total cost of statement 2 in terms of heap operations is $2 \times 2^b \times O(\log N)$. Based on two observations: (i) $2^b < \frac{n}{\log n}$ and (ii) $N \ll n$, we can conclude that the number of heap operations is $O(n)$. However, the 128-bit strings of bucket i are distributed in at most ℓ files on disk in the worst case (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 1.3 of Figure 4.5, where a seek operation in File j may be performed by the first read operation.

In order to amortize the number of seeks performed we use a buffering technique [38]. We create a buffer j of size $\mathbb{B} = \mu/N$ for each file j, where $1 \leq j \leq N$ (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, $\mathbb{B}$ bytes are read from file j to buffer j. Hence, the number of seeks performed in the worst case is given by $\beta/\mathbb{B}$ (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 $16n/\mathbb{B}$, since after the partitioning step we are dealing with 128-bit (16-byte) strings instead of 64-byte URLs, on average. Therefore, the number of seeks is linear on n and amortized by $\mathbb{B}$.

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 $\mathbb{B}$ in practice. Second, almost all main memory is available to be used as file buffers because just the 128-bit strings of the bucket being processed and $O(N)$ words for the heap must be kept in main memory during the searching step, as we show in Section 4.2.2.

To conclude the searching step analysis we need to show that statements 2.2 and 2.4 perform a number of operations proportional to $|B_i|$. If it's true, then the rest of statement 2 runs in $\delta \sum_{i=0}^{2^b-1} |B_i|$ time, where δ is a machine-dependent constant.

Statement 2.2 runs the algorithm used to generate a MPHF for each bucket. That algorithm is linear in the number of keys it is applied to, as we have shown in Chapter 3. As it is applied to buckets with $|B_i|$ keys, then statement 2.2 performs a number of operations proportional to $|B_i|$.

Statement 2.4 has time complexity proportional to $|B_i|$ because it writes to disk the description of each generated MPHF and each description is stored in $O(|B_i|)$ bits (see Chapter 3 for details). As $\sum_{i=0}^{2^b-1} |B_i| = n$, then statement 2 runs in $O(n)$ time. In conclusion, our algorithm takes $O(n)$ time because both the partitioning and the searching steps run in $O(n)$ time.

4.2.2 Space used for constructing a MPHF

We need an 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). One could argue about 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 (i.e., the number of keys being hashed) and can be fixed a priori.

The additional space required is $O(N)$ computer words that corresponds to the size of the heap used to drive a N-way merge operation in the searching step, which allows the merge operation to be performed in one pass through each file. This comes from the fact that 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 256.

4.2.3 Analysis of the hash functions

In this section we sketch the analysis of the hash functions used in the EPH algorithm. Note that the hash functions h_0, y_1, y_2, $\ldots$, y_k have a range of $b + kr$ bits in total. Thus, by universality of linear hash functions [1], the probability that there exist two keys that have the same values under all functions is at most $\binom{n}{2}/2^{b+kr}$. We will choose r such that this probability becomes negligible. For simplicity, we assume that the zero vector 0^L is not in the set S – it is not hard to see that this assumption is insignificant.

A direct consequence of Theorem 5 in [1] is that, assuming $b \leq \log n - \log\log n$, the expected size of the largest bucket is $O(n \log b/2^b)$, i.e., a factor $O(\log b)$ from the average bucket size. This justifies the choice of b in Eq. (4.2), imposing the requirement that $\ell \geq \log n \log\log n$.

For any choice of the random seed s, we will now analyze the probability (over the choice of $y_1, \ldots, y_k$) that $x \mapsto \rho(x, s, 0)$ and $x \mapsto \rho(x, s, 1)$ map the elements of B_i uniformly and independently to $\{0, \ldots, p-1\}$. A sufficient criterion for this is that the sums $\sum_{j=1}^{k} t_j[y_j(x) \oplus \Delta]$ and $\sum_{j=k+1}^{2k} t_j[y_j(x) \oplus \Delta]$, $\Delta \in \{0, 1\}$, have values that are uniform in $\{0, \ldots, p-1\}$ and independent. This is the case if for every $x \in B_i$ there exists an index j_x such that neither y_{j_x} or $y_{j_x} \oplus 1$ belongs to $y_{j_x}(B_i - \{x\})$. Since $y_1, \ldots, y_k$ are universal hash functions, the probability that this is not the case for a given element $x \in B_i$ is bounded by $(|B_i|/2^r)^k \leq (\ell/2^r)^k$. If we choose, for example $r = \lceil \log(\sqrt[3]{n}\ell) \rceil$ and $k = 4$ we have that this probability is $o(1/n)$. Hence, the probability that this happens for *any* key in S is $o(1)$.

Finally, we need to argue that for each bucket i it is easy to find a value of s such that the pair h_{i1}, h_{i2} is good for the MPHF of the bucket. We know that with constant probability this is the case if the functions were truly random. Now, as argued above, with probability $1 - o(1)$ the functions $x \mapsto \rho(x, s, 0)$ and $x \mapsto \rho(x, s, 1)$ are random and independent on each bucket, for every value of s. Then, for a given bucket and a given value of s there is a probability $\Omega(1)$ that the pair of hash functions work for that bucket.

Now, for any $\Delta \in \{0, 1\}$ and $s \neq s'$, the functions $x \mapsto \rho(x, s, \Delta)$ and $x \mapsto \rho(x, s', \Delta)$ are independent. Thus, by Chebychev's inequality the probability that less than a constant fraction of the values of s work for a given bucket is $O(1/p)$. So with probability $1 - o(1)$ there is a constant fraction of "good" choices of s in every bucket, which means that trying an expected constant number of random values for s is sufficient in each bucket.

4.3 Experimental results

In this section we present the experimental results. We start presenting the experimental setup. We then present the performance of our algorithm considering construction time, storage space and evaluation time as metrics for the resulting functions. Finally, we discuss how the amount of internal memory available affects the runtime of our two-step external memory based algorithm.

4.3.1 The data and the experimental setup

The EPH algorithm was implemented in the C language and is available at `http://cmph.sf.net` under the GNU Lesser General Public License (LGPL). All experiments were carried out on a computer running the Linux operating system, version 2.6, with a 1 gigahertz AMD Athlon 64 Processor 3200+ and 1 gigabyte of main memory.

Our data consists of a collection of 1.024 billion URLs collected from the Web, each URL 64 characters long on average. The collection is stored on disk in 60.5 gigabytes of space.

4.3.2 Performance of the algorithms

We are firstly interested in verifying the claim that the EPH algorithm runs in linear time. Therefore, we run the algorithm for several numbers n of keys in S.

The values chosen for n were 1, 2, 4, 8, 16, 32, 64, 128, 512 and 1024 million. We limited the main memory in 512 megabytes for the experiments in order to show that the algorithm does not need much internal memory to generate MPHFs. The size μ of the a priori reserved internal memory area was set to 200 megabytes. In Section 4.3.3 we show how μ affects the runtime of the algorithm. The parameter b (see Eq. (4.2)) 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 $13, 14, 15, 16, 17, 18, 19, 20, 22$ and 23 bits.

In order to estimate the number of trials for each value of n we use a statistical method for determining a suitable sample size (see, e.g., [34, Chapter 13]). We got that just one trial for each n would be enough with a confidence level of 95%. However, we made 10 trials. This number of trials seems rather small, but, as shown below, the behavior of the EPH 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.

Table 4.1 presents the runtime average for each n, the respective standard deviations, and the respective confidence intervals given by the average time $\pm$ the distance from average time considering a confidence level of 95%. Observing the runtime averages we noticed that the algorithm runs in expected linear time, as we have claimed. Better still, it outputs the resulting MPHF faster than all practical algorithms we know of, because of the following reasons. First, the memory accesses during the generation of a MPHF for a given bucket cause cache hits, once the problem was broken down into problems of size up to 256. Second, at searching step we are dealing with 16-byte (128-bit) strings instead of 64-byte URLs.

n (millions)	1	2	4	8	16
Average time (s)	3.34 ± 0.02	6.97 ± 0.02	14.64 ± 0.04	31.75 ± 0.49	68.98 ± 0.82
SD	0.03	0.03	0.05	0.73	1.22
n (millions)	32	64	128	512	1024
Average time (s)	142.71 ± 1.44	288.95 ± 2.65	604.70 ± 6.22	2383.08 ± 22.11	4982.97 ± 55.14
SD	2.01	3.70	8.69	28.77	51.12

Table 4.1: EPH algorithm: average time in seconds for constructing a MPHF with confidence level of 95% in a PC using 200 megabytes of internal memory.

Figure 4.7 presents the runtime for each trial. In addition, the solid line corresponds to a linear regression model obtained from the experimental measurements. As we were expecting the runtime for a given n has almost no variation. The percentages of the total time spent in the partitioning step and in the searching are approximately 49% and 51%, respectively.

An intriguing observation is that the runtime of the algorithm is almost deterministic, in spite of the fact that it uses as building block an algorithm with a considerable fluctuation in its runtime. A given bucket i, $0 \leq i < 2^b$, is a small set of keys (at most 256 keys) and, the runtime of the building block algorithm is a random variable X_i with high fluctuation (it follows a geometric distribution with mean $1/Pr_a \approx 3$). However, the runtime Y of the searching step of the EPH algorithm is given by $Y = \sum_{0 \leq i < 2^b} X_i$. Under the hypothesis

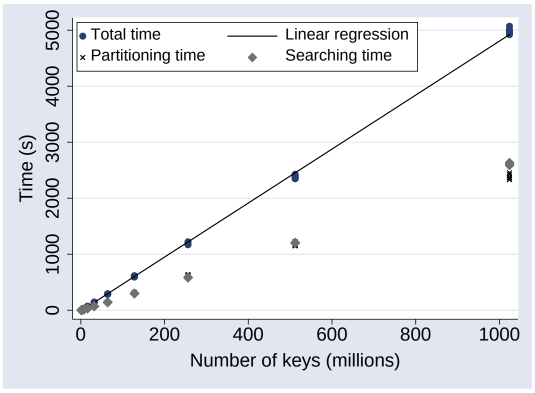


Figure 4.7: Partitioning time and searching time versus number of keys in S for the EPH algorithm. The solid line corresponds to a linear regression model for the total time.

that the X_i are independent and bounded, the *law of large numbers* (see, e.g., [34]) implies that the random variable $Y/2^b$ converges to a constant as $n \to \infty$. This explains why the runtime is almost deterministic.

The next important metric on MPHFs is the space required to store the functions. In order to apply the algorithm used for the buckets to larger sets we randomly choose f_0 and f_1 from the family of universal hash functions proposed by Thorup [54]. In Chapter 3 we have analyzed that algorithm under the uniform hashing assumption so that universal hashing is not enough to guarantee that it works for every key set. But it has been the case for every key set we have applied it to. Then, we refer to this version as *heuristic BPZ algorithm*.

Table 4.2 shows how many bits per key the heuristic BPZ algorithm requires to store the resulting MPHFs. In our setup the heuristic BPZ algorithm requires around 2.1 and 3.3 bits per key to respectively store the resulting PHFs and MPHFs. In a PC with 1 gigabyte of main memory the largest set we are able to generate a MPHF for is a set with 30 millions of keys, because of the sparse graph required to generate the functions is memory demanding.

The EPH algorithm is designed to be used when the key set does not fit in main memory. Table 4.3 shows that it can be used for constructing PHFs and MPHFs that require approximately 2.7 and 3.8 bits per key to be stored, respectively.

The lookup tables used by the hash functions of the EPH algorithm require a fixed storage cost of 1,847,424 bytes. To avoid the space needed for lookup tables we have implemented a version of the EPH algorithm that uses the pseudo random hash function

n	Bits/key	
	PHF	MPHF
10^4	2.13	3.37
10^5	2.09	3.32
10^6	2.09	3.32
10^7	2.09	3.32

Table 4.2: Heuristic BPZ algorithm: space usage to respectively store the resulting PHFs and MPHFs.

n	b	Bits/key	
		PHF	MPHF
10^4	6	2.93	3.71
10^5	9	2.73	3.57
10^6	13	2.65	3.82
10^7	16	2.51	3.70
10^8	20	2.80	4.02
10^9	23	2.65	3.83

Table 4.3: EPH algorithm: space usage to respectively store the resulting PHFs and MPHFs.

proposed by Jenkins [37]. This function was used instead of the linear hash function described in Section 4.1.3, and instead of the two truly random hash function of each bucket, i.e., h_{i1} and h_{i2}, where $0 \leq i < 2^b$. This version is, from now on, referred to as *heuristic EPH algorithm*. 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 now contains 12-byte long strings instead of 16-byte long strings.

The Jenkins function needs just one random seed of 32 bits to be stored instead of quite long lookup tables, a great improvement from the 1,847,424 bytes necessary to implement truly random hash functions. Therefore, there is no fixed cost to store the resulting MPHFs, but two random seeds of 32 bits are required to describe the functions h_{i1} and h_{i2} of each bucket. As a consequence, the MPHFs generation and the MPHFs efficiency at retrieval time are faster (see Table 4.4 and 4.5). The reasons are twofold. First, we are dealing with 12-byte strings computed by the Jenkins function instead of 16-byte strings of the truly random functions presented in Section 4.1.3. Second, there are no large lookup tables to cause *cache misses*. For example, the construction time for a set of 1024 million keys has dropped down to 64.3 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 this paper are required, even being slower. In the implementation available, the hash functions to be used can be chosen by the user.

Table 4.4 presents a comparison of our algorithm with the ones proposed by Botelho, Pagh and Ziviani [9] (BPZ), by Pagh [46] (Hash-displace), by Botelho, Kohayakawa and Ziviani [8] (BKZ), by Czech, Havas and Majewski [18] (CHM), and by Fox, Chen and Heath [28] (FCH), considering construction time and storage space as metrics. Notice that they are the most important practical results on MPHFs known in the literature (see Section 1.3.) Observing the results, the heuristic BPZ algorithm is the best choice for sets that can be handled in main memory and the EPH algorithm is the first one that can be applied to sets that do not fit in main memory and is the fastest one at construction time.

Time in seconds to construct a MPHF for 2×10^6 keys			
Algorithms	Function type	Construction time (seconds)	bits/key
EPH Algorithm	PHF	6.92 ± 0.04	2.64
	MPHF	6.98 ± 0.01	3.85
Heuristic EPH Algorithm	MPHF	4.75 ± 0.02	3.7
Heuristic BPZ Algorithm	PHF	12.99 ± 1.01	2.09
	MPHF	13.94 ± 1.06	3.35
Hash-displace	MPHF	46.18 ± 1.06	64.00
BKZ	MPHF	8.57 ± 0.31	32.00
CHM	MPHF	13.80 ± 0.70	66.88
FCH	MPHF	758.66 ± 126.72	5.84

Table 4.4: Construction time and storage space without considering the fixed cost to store lookup tables.

Finally, we show how efficient is the resulting MPHFs at retrieval time for the methods aforementioned, which is as important as construction time and storage space. Table 4.5 presents the time, in seconds, to evaluate 2×10^6 keys. We group the BKZ and CHM methods together because the resulting MPHFs have the same form. From the results we can conclude that the heuristic BPZ algorithm generates MPHFs that are as fast to be computed as the ones generated by the most practical methods on MPHFs. The MPHFs generated by the EPH algorithm are slower. Nevertheless, the difference is not so expressive (each key can be evaluated in few microseconds) and the EPH algorithm is the first efficient option for sets that do not fit in main memory.

It is important to emphasize that the BPZ, BKZ, CHM and FCH methods were analyzed under the full randomness assumption. Therefore, the EPH algorithm is the first one that has experimentally proven practicality for large key sets and has both space usage for representing the resulting functions and the construction time carefully proven.

Time in seconds to evaluate 2×10^6 keys						
key length (bytes)	Function type	8	16	32	64	128
EPH Algorithm	PHF	2.05	2.31	2.84	3.99	7.22
	MPHF	2.55	2.83	3.38	4.63	8.18
Heuristic EPH Algorithm	MPHF	1.19	1.35	1.59	2.11	3.34
Heuristic BPZ Algorithm	PHF	0.41	0.55	0.79	1.29	2.39
	MPHF	0.85	0.99	1.23	1.73	2.74
Hash-displace	MPHF	0.56	0.69	0.93	1.44	2.54
BKZ/CHM	MPHF	0.61	0.74	0.98	1.48	2.58
FCH	MPHF	0.58	0.72	0.96	1.46	2.56

Table 4.5: Evaluation time.

Additionally, it is the fastest algorithm for constructing the functions and the resulting functions are much simpler than the ones generated by previous theoretical well-founded schemes so that they can be used in practice. Also, it considerably improves the first step given by Pagh with his hash and displace method [46].

4.3.3 Controlling disk accesses

In order to bring down the number of seek operations on disk we benefit from the fact that the EPH algorithm leaves 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 the EPH algorithm. For that we fixed n in 1.024 billion of URLs, set the main memory of the machine used for the experiments to 1 gigabyte and used μ equal to 100, 200, 300, 400 and 500 megabytes.

Table 4.6 presents the number of files N, the buffer size used for all files, the number of seeks in the worst case considering the pessimistic assumption that one seek happens every time buffer j is filled in (see Section 4.2.1 for details), and the time to generate a PHF or a MPHF for 1.024 billion of keys as a function of the amount of internal memory available. Observing Table 4.6 we noticed that the time spent in the construction decreases as the value of μ increases. However, for $\mu > 400$, the variation on the time 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)	100	200	300	400	500
N (files)	245	99	63	46	36
$\mathbb{B}$ (in KB)	418	2,069	4,877	8,905	14,223
$\beta/\mathbb{B}$	151,768	30,662	13,008	7,124	4,461
EPH (time)	94.8	82.2	79.8	79.2	79.2
Heuristic EPH (time)	71.0	63.2	62.9	62.4	62.4

Table 4.6: Influence of the internal memory area size (μ) in the EPH algorithm runtime to construct PHFs or MPHFs for 1.024 billion keys (time in minutes).

4.4 Conclusions

This chapter has presented a novel external memory based algorithm for constructing PHFs and MPHFs. The algorithm can be used with provably good hash functions or with heuristic hash functions that are faster to compute.

The EPH algorithm contains, as a component, a provably good implementation of the BPZ algorithm [9] (see Chapter 3 for details). This means that the two hash functions h_{i1} and h_{i2} (see Eq. (4.3)) used instead of f_0 and f_1 behave as truly random hash functions (see Section 4.2.3). The resulting PHFs and MPHFs require approximately 2.7 and 3.8 bits per key to be stored and are generated faster than the ones generated by all previous methods. The EPH algorithm is the first one that has experimentally proven practicality for sets in the order of billions of keys and has time and space usage carefully analyzed without unrealistic assumptions. As a consequence, the EPH algorithm will work for every key set.

The resulting functions of the EPH algorithm are approximately four times slower than the ones generated by all previous practical methods (see Table 4.5). The reason is that to compute the involved hash functions we need to access lookup tables that do not fit in the cache. To overcome this problem, at the expense of losing the guarantee that it works for every key set, we have proposed a heuristic version of the EPH algorithm that uses a very efficient pseudo random hash function proposed by Jenkins [37]. The resulting functions require the same storage space, are now less than two times slower to be computed and are still faster to be generated.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis proposal we have presented two classes of new 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 an external memory based 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.

The first algorithm we came up with belongs to the class of internal memory based algorithms and assumes uniform hashing to generate MPHFs based on random graphs. The algorithm is presented in Chapter 2. It improves the space requirement of the algorithm by Czech, Havas and Majewski [18], referred to as CHM, at the expense of generating functions in the same form that are not order preserving, but are computed in time $O(1)$. The CHM algorithm uses acyclic random graphs with n edges and cn vertices, where $c > 2$. Our algorithm uses simple random graphs with n edges and cn vertices, but now $c \in [0.93, 1.15]$. A consequence of the smaller number of vertices is that the graph may have cycles. As the space to store the resulting functions in both algorithms is directly related to the number of vertices in the random graph, then we have improved the space required to store a function in our algorithm to 55% of the space required by the CHM algorithm. The algorithm is also linear on n and runs 60% 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.

The second work, presented in Chapter 3, involves a family $\mathcal{F}$ of near space-optimal internal memory based algorithms for generating PHFs and MPHFs. The algorithms in $\mathcal{F}$ also assume uniform hashing and use 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 $\log n$ the well known result by Majewski et al [42]. They generates MPHFs based on r-graphs that are stored in $O(n \log n)$ while the ones generated by our algorithms require $O(n)$ 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 $1.4427n$ bits.

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 factor of 2 from the information theoretical lower bound of around $1.17n$ 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.

The two previous results assume uniform hashing. Therefore, theoretically speaking, it is not guaranteed that the methods work for every key set if universal hash functions are used instead of uniform hash functions. However, in practice, we have never found a key set for which a PHF or MPHF could not be generated using universal hash functions. The methods also require $O(n)$ computer words for the construction process.

In our third work we have designed an external memory based algorithm, referred to as EPH algorithm, that does not assume that uniform hash functions are available for free and requires $O(N)$ computer words, where $N \ll n$, for constructing the functions in linear time. The resulting PHFs and MPHFs require approximately 2.7 and 3.8 bits per key to be stored and are evaluated in constant time. The main technical idea behind the EPH algorithm is that it splits the incoming key set S into small buckets containing at most 256 keys. Then, a MPHF is generated for each bucket and using an *offset* array we obtain a MPHF for S. All together makes the EPH algorithm the first one that demonstrates the capability of generating MPHFs for sets in the order of billions of keys on a commodity PC. Also, the algorithm is theoretically sound because with the help of Rasmus Pagh [7] we have completely analyzed it's time and space usage without unrealistic assumptions.

We believe that the EPH algorithm will be very useful for the information retrieval community. Search engines are nowadays indexing tens of billions of pages and the work with huge collections is becoming a daily task. For instance, the simple assignment of number identifiers to web pages of a collection can be a challenging task. While traditional databases simply cannot handle more traffic once the working set of URLs does not fit in main memory anymore [53], the EPH algorithm we propose here to construct MPHFs can easily scale to billions of entries. Also, algorithms like PageRank [11], which uses the web link structure to derive a measure of popularity for Web pages, operates on the web graph. At construction time of the graph, the URLs must be mapped to integers that will be used to label the vertices. For the same reason, the WebGraph research group [3] would also benefit from a MPHF for sets in the order of billions of URLs to scale and to improve the storage requirements of their algorithms on graph compression.

5.2 Future Work

The future work is as follows:

1. In Chapter 3, the bounds for the algorithms based on r-graphs for $r \geq 3$ have not been completely proved. The problems for $r < 3$ and for $r \geq 3$ have different natures and involve a phase transition, as reported to us by Kohayakawa [39]. Then, we aim to study the problem and to try to obtain a fully proof of the bounds for $r \geq 3$. In the following, the proof presented in Section 4.2.3 is too sketchy. Then we aim to provide an expanded version of that proof.

2. The main technical ingredient of the family of algorithms presented in Chapter 3 is the use of acyclic r-partite random hypergraphs. We believe that we can make use of the fact that our hypergraphs are r-partite to speed up the evaluation of the resulting functions in modern processors with various cores. The r memory probes done in function g (represented by an array) can be done in parallel if the portion of g corresponding to each partition fits in the cache of each core. Pagh [46] have shown that optimal number of memory probes required to evaluate any MPHF is one. Then, we believe that our functions will behave as the optimal ones in modern processors.

3. The EPH algorithm presents intrinsically a high degree of parallelism. Therefore, a distributed implementation of the EPH algorithm will allow the construction and the evaluation of the resulting function in a distributed way. Therefore, the

description of the resulting MPHFs will be distributed in the parallel computer allowing the scalability to sets of hundreds of billions of keys. This will be an important contribution, mainly for applications related to the Web, as mentioned in Section 5.1.

4. We aim to study the application of the EPH algorithm to load balancing for a scalable run-time environment for data mining applications, which is called Anthill [27].

5. We want to investigate the use of the new algorithms for representing very large vocabularies in text compression techniques and web search engines.

6. A problem with all the 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 dynamic minimal perfect hash functions.

Bibliography

[1] N. Alon, M. Dietzfelbinger, P.B. Miltersen, E. Petrank, and G. Tardos. Linear hash functions. *Journal of the ACM*, 46(5):667–683, 1999.

[2] N. Alon and M. Naor. Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16(4-5):434–449, 1996.

[3] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *Proc. of the 13th International World Wide Web Conference (WWW'04)*, pages 595–602, 2004.

[4] B. Bollobás. *Random graphs*, volume 73 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, second edition, 2001.

[5] B. Bollobás and O. Pikhurko. Integer sets with prescribed pairwise differences being distinct. *European Journal of Combinatorics*. To Appear.

[6] F. C. Botelho. Estudo comparativo do uso de hashing perfeito mínimo. Master's thesis, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Novembro 2004.

[7] F. C. Botelho, R. Pagh, and N. Ziviani. Perfect hashing for data management applications. Technical Report TR002/07, Federal University of Minas Gerais, 2007. Available at http://arxiv.org/pdf/cs/0702159.

[8] 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.

[9] 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. To appear.

[10] F.C. Botelho and N. Ziviani. External perfect hashing for very large key sets. In *Submitted to 16th Conference on Information and Knowledge Management*, 2007.

[11] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. of the 7th International World Wide Web Conference (WWW'98)*, pages 107–117, April 1998.

[12] 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.

[13] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[14] Chin-Chen Chang and Chih-Yang Lin. A perfect hashing schemes for mining association rules. *The Computer Journal*, 48(2):168–179, 2005.

[15] Chin-Chen Chang, Chih-Yang Lin, and Henry Chou. Perfect hashing schemes for mining traversal patterns. *Journal of Fundamenta Informaticae*, 70(3):185–202, 2006.

[16] 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)*, 2004.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[18] 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.

[19] Z.J. Czech, G. Havas, and B.S. Majewski. Fundamental study perfect hashing. *Theoretical Computer Science*, 182:1–143, 1997.

[20] E. Demaine, F. Meyer auf der Heide, R. Pagh, and M. Pătraşcu. De dictionariis dynamicis pauco spatio utentibus. In *Proc. of the Latin American Symposium on Theoretical Informatics (LATIN'06)*, pages 349–361, 2006.

[21] M. Dietzfelbinger and T. Hagerup. Simple minimal perfect hashing in less space. In *Proc. of the 9th European Symposium on Algorithms (ESA'01)*, pages 109–120. Springer LNCS vol. 2161, 2001.

[22] M. Dietzfelbinger and C. Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. In *Proc. of 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, pages 166–178, 2005.

[23] J. Ebert. A versatile data structure for edges oriented graph algorithms. *Communication of The ACM*, (30):513–519, 1987.

[24] P. Erdős and A. Rényi. On random graphs I. *Pub. Math. Debrecen*, 6:290–297, 1959.

[25] P. Erdős and A. Rényi. On the evolution of random graphs. *Magyar Tud. Akad. Mat. Kutató Int. Közl.*, 5:17–61, 1960.

[26] P. Erdős and A. Rényi. On the strength of connectedness of a random graph. *Acta Mathematica Scientia Hungary*, 12:261–267, 1961.

[27] R. A. Ferreira, W. Meira Jr., D. Guedes, L. M. A. Drummond, B. Coutinho, G. Teodoro, T. Tavares, R. Araujo, and G. T. Ferreira. Anthill: A scalable run-time environment for data mining applications. In *SBAC-PAD '05: Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing*, pages 159–167, Washington, DC, USA, 2005. IEEE Computer Society.

[28] E.A. Fox, Q.F. Chen, and L.S. Heath. A faster algorithm for constructing minimal perfect hash functions. In *Proc. of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 266–273, 1992.

[29] 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.

[30] M. L. Fredman, J. Komlós, and E. Szemerédi. On the size of separating systems and families of perfect hashing functions. *SIAM Journal on Algebraic and Discrete Methods*, 5:61–68, 1984.

[31] 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.

[32] T. Hagerup and T. Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Proc. of the 18th Symposium on Theoretical Aspects of Computer Science (STACS'01)*, pages 317–326. Springer LNCS vol. 2010, 2001.

[33] G. Havas, B.S. Majewski, N.C. Wormald, and Z.J. Czech. Graphs, hypergraphs and hashing. In *Proc. of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 153–165. Springer LNCS vol. 790, 1993.

[34] R. Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling.* John Wiley, first edition, 1991.

[35] S. Janson. Poisson convergence and poisson processes with applications to random graphs. *Stochastic Processes and their Applications*, 26:1–30, 1987.

[36] S. Janson, T. Łuczak, and A. Ruciński. *Random graphs.* Wiley-Inter., 2000.

[37] B. Jenkins. Algorithm alley: Hash functions. *Dr. Dobb's Journal of Software Tools*, 22(9), september 1997.

[38] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, second edition, 1973.

[39] Yoshihary Kohayakawa. Private communication, 2007.

[40] P. Larson and G. Graefe. Memory management during run generation in external sorting. In *Proc. of the 1998 ACM SIGMOD international conference on Management of data*, pages 472–483. ACM Press, 1998.

[41] S. Lefebvre and H. Hoppe. Perfect spatial hashing. *ACM Transactions on Graphics*, 25(3):579–588, 2006.

[42] 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.

[43] 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.

[44] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* Springer-Verlag, 1984.

[45] 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.

[46] 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.

[47] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

[48] E. M. Palmer. *Graphical Evolution: An Introduction to the Theory of Random Graphs.* John Wiley & Sons, New York, 1985.

[49] B. Pittel and N. C. Wormald. Counting connected graphs inside-out. *J. Combin. Theory Ser. B*, 93(2):127–172, 2005.

[50] B. Prabhakar and F. Bonomi. Perfect hashing for network applications. In *Proc. of the IEEE International Symposium on Information Theory*. IEEE Press, 2006.

[51] J. Radhakrishnan. Improved bounds for covering complete uniform hypergraphs. *Information Processing Letters*, 41:203–207, 1992.

[52] 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.

[53] M. Seltzer. Beyond relational databases. *ACM Queue*, 3(3), April 2005.

[54] M. Thorup. Even strongly universal hashing is pretty fast. In *Proc. of the eleventh annual ACM-SIAM symposium on Discrete algorithms (SODA'00)*, pages 496–497, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[55] P. Woelfel. Maintaining external memory efficient hash tables. In *Proc. of the 10th International Workshop on Randomization and Computation (RANDOM'06)*, pages 508–519. Springer LNCS vol. 4110, 2006.