A New Algorithm for Constructing Minimal Perfect Hash Functions

Fabiano C. Botelho David M. Gomes Nivio Ziviani

Department of Computer Science
Federal University of Minas Gerais
Belo Horizonte, Brazil
{fbotelho,menoti,nivio}@dcc.ufmg.br

Abstract

We present a three-step algorithm for generating minimal perfect hash functions which runs very fast in practice. The first step is probabilistic and involves the generation of random graphs. The second step determines the order in which hash values are assigned to keys. The third step assigns hash values to the keys. We give strong evidences that first step takes linear random time and the second and third steps take deterministic linear time. We improve upon the fastest known method for generating minimal perfect hash functions. The total time to find a minimal perfect hash function in a PC computer took approximately 175 seconds for a collection of 20 million keys. The time to compute a table entry for any key is also fast because it uses only two different hash functions that are computable in time proportional to the size of the key. The amount of space necessary to store the minimal perfect hash function is approximately half the space used by the fastest known algorithm.

1 Introduction

Let S be a set of n distinct keys belonging to a finite universe U of keys. The keys in S are stored so that membership queries asking if key $x \in U$ is in S can be answered. This search problem is called the *dictionary problem*. Various approaches to the dictionary problem have been explored. One of them is to compute a function $h(x)$ to determine the location of a key in a table, leading to a class of very efficient searching methods known as *hashing* methods.

Hashing methods for *non static sets* of keys involve a certain amount of wasted space due to unused locations in a table and wasted time to resolve collisions when two keys are hashed to the same table location. If the set of keys is *static*, then it is possible to compute a function $h(x)$ to find any key in the table in one probe (no collisions in this case). This function is called a *perfect hash function*. A perfect hash function that can preserve an a priori key ordering is called an order preserving function. A perfect hash function that stores a set of records in a table of the size equal to the number of keys is called a *minimal perfect hash function*. A minimal perfect hash function totally avoids the common problem of wasted space and time.

Minimal perfect hash functions are used for memory efficient 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 in Web search engines, or item sets in data mining techniques. Therefore, there are applications for minimal perfect hash functions in information retrieval systems, database systems, hypertext, hypermedia, language translation systems, electronic commerce systems, compilers, operating systems, among others.

Finding perfect hash functions, especially for large sets, may not be easy since these functions are very rare. According to Knuth [12], the total number of possible hash functions from S ($|S| = n$) into $[0, m-1]$ ($m \geq n$) is m^n and only $m(m-1)\ldots(m-n+1)$ are perfect. Thus, the probability that no collisions occur is the ratio $(m(m-1)\ldots(m-n+1))/m^n$ which tends to zero very fast. For $m = 13$ and $n = 10$, the probability that no collisions occur is only 0.0074.

Many methods for generating minimal perfect hash functions use a *mapping*, *ordering* and *searching* (MOS) approach, a description coined by Fox, Chen and Heath [8]. 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.

In this paper we present a practical and efficient algorithm to find minimal perfect hash functions for very large key collections where no a priori key order must be maintained (e.g., for applications where ordered sequential access is not needed). The algorithm is based on the MOS approach and have linear time complexity with a small constant. For a collection of 20 million keys, the total time to find a minimal perfect hash function in a PC computer took approximately 175 seconds. The time to compute the table entry $h(x)$ for key x is also very fast, as it uses only two different universal hash functions, each one computable in time proportional to the size of the key x. The space necessary to store the minimal perfect hash function for a set of n keys is $O(\log n)$ bits per key.

2 Related Work

Czech, Havas and Majewski [2] provide a comprehensive survey and review some of the most important theoretical results on perfect hashing. Mehlhorn [14] shows that the lower bound to store a perfect hash function is $\Omega(n/\log n)$ computer words, where n is the size of the key set. Fox et al. [7] show that the space lower bound to store an order preserving perfect hash function is $\Omega(n \log n)$ bits.

Using the MOS approach, Fox et al. [8, 9] presented algorithms for finding minimal perfect hash functions in which the space to store the functions is close to the lower bound to store a minimal perfect hash function. However, in [2, Section 6.7] it is shown that their algorithms have exponential running times.

The works in [1, 2, 13] present a family of efficient and practical algorithms for generating order preserving minimal perfect hash functions. They present one of the best known random methods for generating minimal perfect hash functions. One of their algorithms involves the generation of acyclic random graphs $G = (V, E)$, $|V| = cn$ and $|E| = n$, with $c \geq 2.09$.

The differences between our algorithm and the algorithms in [1, 2, 13] are as follows. First, we generate cyclic random graphs $G = (V, E)$, $|V| = cn$ and $|E| = n$, with $c \geq 1.15$ while they generate acyclic random graphs with a greater number of vertices ($|V| \geq 2.09n$). Second, their mapping step takes longer time because they must generate an acyclic graph while we do not need to check for this property. Third, the time to compute a table entry is faster in our case because we save a module operation. Fourth, they generate order preserving minimal perfect hash functions while our algorithm does not preserve order.

3 Basic Concepts

This section presents the basic concepts that are used in the next sections.

Definition 1 Let $U = \{0, 1, \ldots, u-1\}$ be the universe for some arbitrary positive integer u.

Definition 2 Let S be a set of n distinct keys belonging to U, i.e., n is the size of S.

Definition 3 Let $h : U \to M$ be a *hashing function* that maps the keys from S into a given interval of integers $[0, m-1]$. Given a key $x \in S$, the hash function computes an integer in $[0, m-1]$ for the storage or retrieval of x in a *hash table*, i.e., m is the size of a hash table.

Definition 4 Let $G = (V, E)$ be a *random undirected graph* without self-loops and multiple edges, where $|E| = n$, $|V| = cn$, generated using a variation of the *uniform model* [6]. In this model, at each step we generate an unordered pair $e = \{u, v\}$, where u and $v \in V$, from $\binom{|V|}{2}$ pairs equally likely. If the undirected edge e is neither a self-loop nor a multiple edge then it is added to G. A self-loop occurs when $u = v$. For an edge $e_1 = \{u', v'\}$ already added to G, a multiple edge occurs when one of the conditions ($u = u'$ and $v = v'$) or ($u = v'$ and $v = u'$) is true.

Definition 5 Let $\mathrm{Adj}(v)$ be the adjacent list of a vertex $v \in V$.

Definition 6 Let $d_{avg} = \frac{2|E|}{|V|}$ be the *average degree* of the vertices V of G.

Definition 7 Let $V_{crit} \subseteq V$ be a set of *critical vertices*. Subset V_{crit} contains all vertices from V that are part of cycles or are in a chain connecting two or more cycles, as the vertex 5 in Figure 4.

Definition 8 Let $E_{crit} \subseteq E$ be a set of *critical edges*. Subset E_{crit} contains all edges from E connecting critical vertices.

Definition 9 Let $V_{ncrit} = V - V_{crit}$ be a set of *non critical vertices*. Subset V_{ncrit} contains all vertices from V that are not part of cycles.

Definition 10 Let $V_{scrit} \subseteq V_{crit}$ be a set of all critical vertices that have at least one non critical vertex as adjacent, as the vertex 5 in Figure 4.

Definition 11 Let $E_{ncrit} = E - E_{crit}$ be a set of *non critical edges*.

Definition 12 Let $G_{crit} = (V_{crit}, E_{crit})$ be a *critical graph* and let $G_{ncrit} = (V_{ncrit} \cup V_{scrit}, E_{ncrit})$ be a *non critical graph*, where the critical subgraph G_{crit} corresponds to the cyclic part of G and the non critical subgraph G_{ncrit} corresponds to the acyclic part of G. Thus, $G = G_{crit} \cup G_{ncrit}$.

Definition 13 Let $P_{|E_{crit}|}$ be the probability that G has at most $|E_{crit}|$ critical edges.

Definition 14 Let $P_{|V_{crit}|}$ be the probability that G has at most $|V_{crit}|$ critical vertices.

4 Minimal Perfect Hashing

In a hashing method, different keys might have the same address computed by the hash function, a situation called *collision*. In this case various schemes for resolving collisions are known. A *perfect hash function* is an injection $h : U \to [0, m-1]$, which means that for all $x, y \in S$ such that $x \neq y$ we have $h(x) \neq h(y)$, which implies that $m \geq n$. For being an injection, a perfect hash function transforms each key of S into a unique address in the hash table, as depicted in Figure 1(a). Since no collisions occur, each key can be retrieved from the table in one probe. If $m = n$ and $h(x)$ is perfect, then $h(x)$ is a *minimal perfect hash function* (MPHF), as depicted in Figure 1(b). The perfect hash function h is said to be *order preserving* if for any pair of keys x_i, $x_j \in S$ then $h(x_i) < h(x_j)$ if and only if $i < j$. In other words, the keys in S are arranged in some order and the function h preserves this order in the hash table.

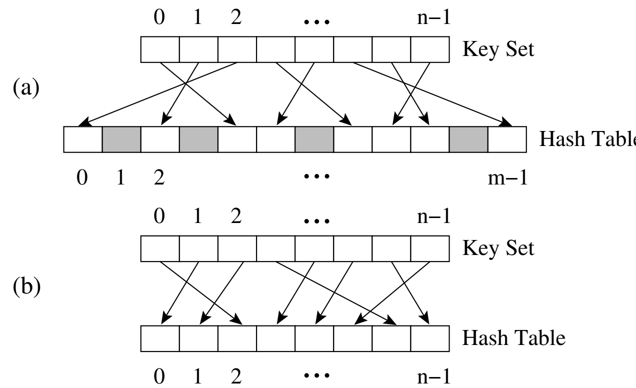


Figure 1: (a) Perfect hash function. (b) Minimal perfect hash function.

In the definitions above the keys to be placed in the hash table are integers in the interval $[0, m-1]$. In practice, it is often the case that keys are sequences of characters over some finite and ordered alphabet Σ, such as the ASCII set. In this case, we convert it to a random number modulo $|V|$ for each key. To obtain a random number for each key, we generate a table of random numbers, one for each possible character of Σ at each position i in the key. The construction of the minimal perfect hash function presumes the existence of two random and independent hash functions h_1 and h_2. For a key x containing $|x|$ characters and two different tables of random numbers $table_1$ and $table_2$, the two hashing functions are:

$$h_1(x) = \left(\sum_{i=1}^{|x|} table_1[i, x[i]] \right) \bmod |V|,$$

$$h_2(x) = \left(\sum_{i=1}^{|x|} table_2[i, x[i]] \right) \bmod |V|.$$

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 $h : E \to \{0, 1, \ldots, n-1\}$, defined as

$$h(e) = (g(a) + g(b)) \bmod n \qquad (1)$$

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 $[0, n-1]$.

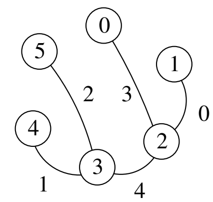
The ordering and searching steps of the MOS approach are a very simple way of solving the perfect assignment problem. Czech, Havas and Majewski [1] showed that the perfect assignment problem can be solved in optimal time if G is acyclic. To generate an acyclic graph two vertices $h_1(x)$ and $h_2(x)$ are computed for each key $x \in S$. Thus, set S has a corresponding graph G, with $V = \{0, 1, \ldots, v\}$ 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 $|V| = cn$, for some constant c, such that acyclic graphs dominate the space of all random graphs. Havas et al. [10] proved that if $|V| = cn$ holds with $c > 2$ the probability that G is acyclic is

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

For $c = 2.09$ the probability of a random graph being acyclic is $p > \frac{1}{3}$. 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. In this paper, the algorithm proposed by Czech, Havas and Majewski [1] will be referred to as CHM from now on.

Given an acyclic graph G, for the ordering step we associate with each edge an unique number $h(e) \in [0, n-1]$ in the order of the keys of S to obtain an order preserving function. Figure 2 illustrates the perfect assignment problem for an acyclic graph with six vertices and with the five table entries 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 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 $h(e)$, set $g(b)$ to $(h(e) - g(a)) \bmod n$. In Figure 2, 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: 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 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.

5 The New Algorithm

In this section we present a new algorithm for constructing minimal perfect hash functions, where the order of the keys in S is not preserved. The algorithm is based on the MOS approach and solves the problem presented in Figure 1(b). The main novelty is that the random graph G might have cycles and even so we are able to find a MPHF.

The new algorithm looks for a function $g : V \to \{-|V|+1, \ldots, 0, 1, \ldots, |V|-1\}$ such that the function $h : E \to \{0, 1, \ldots, m-1\}$ defined as

$$h(e) = g(a) + g(b) \tag{3}$$

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 is a unique integer in the range $[0, m-1]$. Notice that we do not need to take the sum of values associated with endpoints of the edges modulo n.

Figure 3 presents a pseudo code for the new algorithm. The procedure NewAlgorithm (S, g) receives as input the set of keys from S and produces the perfect assignment of vertices represented by the function g. The mapping step generates a random undirected graph G taking S as input. The ordering step determines the order in which hash values are assigned to keys. It partitions the graph G into G_{crit} and G_{ncrit}. The searching step produces the perfect assignment of vertices in G, which is represented by the function g. It starts with G_{crit} and finishes with G_{ncrit}.

```
procedure NewAlgorithm (S, g)
    Mapping (S, G);
    Ordering (G, G_crit, G_ncrit);
    Searching (G, G_crit, G_ncrit, g);
```

Figure 3: Main steps of the new algorithm.

5.1 Mapping Step

The procedure Mapping (S, G) receives as input the set of keys from S and generates a random undirected graph G without self-loops and multiple edges. To generate the MPHF, the number of critical edges in G must be $|E_{crit}| \leq \frac{1}{2}|E|$. The reason is that the maximal value of $h(e)$ assigned to an edge $e \in E$ in this case is $m-1$. In Section 5.3.1 we show that the condition $|E_{crit}| \leq \frac{1}{2}|E|$ is necessary and sufficient to generate a MPHF.

The random graph G is generated using two hash functions h_1 and h_2. The functions h_1 and h_2 transform the keys from S to integers in $[0, |V|-1]$, so the set of vertices V has $|V|$ vertices and each one of them is labelled with a distinct value from $[0, |V|-1]$. For each key x from S the edge $\{h_1(x), h_2(x)\}$ is added to E.

A self-loop occurs when $h_1(x) = h_2(x)$. To avoid self-loops we modify $h_2(x)$ by adding a random number in the range $[1, |V|-1]$. When a multiple edge occurs we abort and start again a new iteration.

We now show that the expected number of iterations to obtain G is constant. Let p be the probability of generating a random graph G without self-loops and multiple edges. Let X be a random variable counting the number of iterations to generate G. Variable X is said to have the 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 $|\xi| = \binom{|V|}{2}$. 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 $p(|V|)$ is:

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

As $|V| = cn$ 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 [15], for two functions $f_1 : \Re \to \Re$ and $f_2 : \Re \to \Re$ defined by $f_1(k) = 1-k$ and $f_2(k) = e^{-k}$, the inequality $f_1(k) \leq f_2(k)$ is true $\forall\ k \in \Re$. Considering $k = \frac{2j}{c^2n^2-cn}$, we have

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

Thus,

$$\lim_{n\to\infty} p(n) \simeq e^{-\frac{1}{c^2}}. \tag{4}$$

As $N_i(X) = 1/p$ then $N_i(X) \simeq e^{\frac{1}{c^2}}$. After that, we empirically determine the c value to obtain a random graph G with $|E_{crit}| \leq \frac{1}{2}|E|$. For this we built 10,000 graphs for each c value and number of keys presented in Table 1. The two collections used in the experiments (TodoBR and TREC-VLC2) are described in Table 4 (see Section 7 for more details).

We show in Table 1 the probability $P_{|E_{crit}|}$ that $|E_{crit}| \leq \frac{1}{2}|E|$, $|E| = n$, tends to 0 when $c < 1.15$ and n increases. However, it tends to 1 when $c \geq 1.15$ and n increases. Thus, $|V| = 1.15n$ is considered a *threshold function* (a definition coined by Erdös and Rényi [3, 5]) for generating a random graph G where $|E_{crit}| \leq \frac{1}{2}|E|$ with probability tending to 1 when n increases. Therefore, we use $c = 1.15$ in the new algorithm.

The MPHF generated by the new algorithm needs $1.15n$ integer numbers to be stored, since $|V| = 1.15n$. Thus, the generated function is stored in 55% — $1.15n/2.09n$ — of the space necessary to store the one generated by the CHM algorithm.

As $P_{|E_{crit}|}$ tends to 1 when n increases, we consider that the expected number of iterations to generate G is $N_i(X) \simeq e^{\frac{1}{c^2}}$. For $c = 1.15$, $N_i(X) \simeq 2.13$ on average, which is constant. So, the mapping step takes $O(n)$ time.

	VLC2 (n)					TodoBR (n)				
c	$1,000$	$10,000$	$100,000$	$1,000,000$	$3,000,000$	$1,000$	$10,000$	$100,000$	$1,000,000$	$3,000,000$
1.10	0.01	0.00	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00
1.11	0.04	0.00	0.00	0.00	0.00	0.05	0.00	0.00	0.00	0.00
1.12	0.12	0.00	0.00	0.00	0.00	0.13	0.00	0.00	0.00	0.00
1.13	0.19	0.03	0.00	0.00	0.00	0.20	0.02	0.00	0.00	0.00
1.14	0.30	0.09	0.00	0.00	0.00	0.31	0.11	0.00	0.00	0.00
1.15	0.50	0.56	0.65	0.89	1.00	0.51	0.57	0.65	0.88	1.00
1.16	0.68	0.70	0.88	0.95	1.00	0.70	0.83	0.95	0.95	1.00
1.17	0.77	0.82	0.90	1.00	1.00	0.78	0.99	0.98	1.00	1.00
1.18	0.91	0.97	0.98	1.00	1.00	0.91	1.00	1.00	1.00	1.00
1.19	0.94	1.00	1.00	1.00	1.00	0.95	1.00	1.00	1.00	1.00
1.20	0.98	1.00	1.00	1.00	1.00	0.99	1.00	1.00	1.00	1.00

Table 1: Probability $P_{|E_{crit}|}$ that $|E_{crit}| \leq \frac{1}{2}n$ for different c values and different number of keys for the collections VLC2 and TodoBR.

The new algorithm does not verify if G really has at most $0.5n$ critical edges in the mapping step. The rationale is that $P_{|E_{crit}|}$ tends to 1 when n increases. However, if some addition $g(u)+g(w)$ is greater than m in the searching step for $\{u, w\} \in E$ then the mapping step is restarted, as shown in line 17 of Figure 7.

5.2 Ordering Step

The procedure Ordering (G, G_{crit}, G_{ncrit}) receives as input the graph G and partitions G into two subgraphs G_{crit} and G_{ncrit}. To partition the graph G into G_{crit} and G_{ncrit} we use an optimal time algorithm, as follows. Figure 4 presents a sample graph with 16 vertices and 14 edges, where the degree of a vertex is shown besides each vertex. Initially, all vertices with degree 1 are added to a queue Q. For the example shown in Figure 4(a), $Q = \{14, 15, 9, 10, 0, 1, 11, 12\}$ after the initialization step. This initialization takes $O(|V|)$ time, because we need to check the degree of each vertex from V.

Next, we remove one vertex v from the queue, decrement its degree and the degree of vertices with degree greater than 0 in the adjacent list of v, as depicted in Figure 4(b) for $v = 14$. At this point, the adjacencies of v with degree 1 are inserted into the queue, such as vertex 13 in Figure 4(c). This process is repeated until the queue becomes empty. All vertices with degree 0 are non critical vertices and the others are critical vertices, as depicted in Figure 4(d). This process takes $O(|V_{ncrit}|)$, because each non critical vertex is removed from the queue only once.

Finally, to determine the vertices in V_{scrit} we collect all vertices $v \in V_{crit}$ with at least one vertex u that is in Adj(v) and in V_{ncrit}, as the vertex 5 in Figure 4(d). This process takes $O(|V_{crit}|)$. Considering that $|V_{crit}| \leq |V|$, $|V_{ncrit}| \leq |V|$ and $|V| = n$, the ordering step takes $O(n)$ time.

5.3 Searching Step

The procedure Searching (G, G_{crit}, G_{ncrit}, g) receives as input G, G_{crit}, G_{ncrit} and finds a $\log_2 |V| + 1$ bit value for each vertex $v \in V$, stored in the array g. A pseudo code for the searching step is presented in Figure 5. The searching step is first performed for the vertices in G_{crit} and second for the vertices in G_{ncrit}.

5.3.1 Assignment of Values to Critical Vertices

The procedure CriticalVerticesAssignment (G, G_{crit}, g, AssignedEdges) receives G and G_{crit} as input and produces as output a g value for each vertex in G_{crit} and the AssignedEdges array. Such array has m entries and indicates the edges for which a value $h(e) \in [0, m-1]$, $e \in E_{crit}$, has already been assigned. We use a breadth-first search algorithm to assign values to each vertex in G_{crit}. The reason we start the assignment of values to vertices in G_{crit} is to resolve reassignments as earlier as possible. The reassignment problem is illustrated in the next paragraph.

Considering the subgraph G_{crit} in Figure 4(d), a step by step example of the assignment of values to vertices

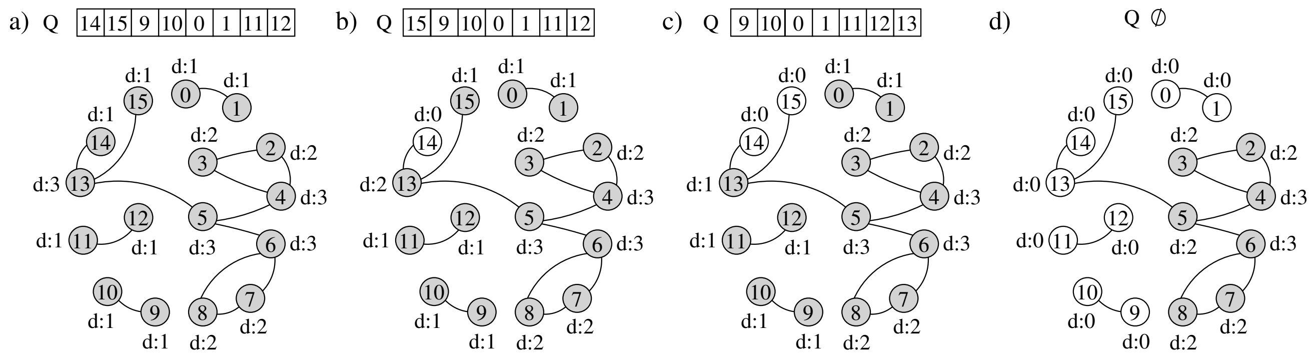


Figure 4: Ordering step for a graph with 16 vertices and 14 edges.

```
procedure Searching (G, G_crit, G_ncrit, g)
    CriticalVerticesAssignment (G, G_crit, g, AssignedEdges);
    NonCriticalVerticesAssignment (G, G_ncrit, AssignedEdges, g);
```

Figure 5: Pseudo code for the searching algorithm.

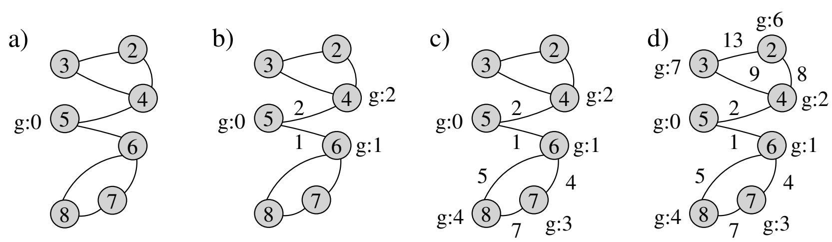


Figure 6: Example of the critical vertices assignment.

in G_{crit} is presented in Figure 6. Initially, a vertex v is chosen chosen and the assignment $g(v) = 0$ is made. For example, suppose that vertex 5 in Figure 6(a) is chosen and the assignment $g(5) = 0$ is made. In Figure 6(b), following the adjacent list of vertex 5, $g(6)$ is set to 1 and $g(4)$ is set to 2, implying that addresses 1 and 2 must be assigned to edges $\{5, 6\}$ and $\{5, 4\}$, respectively. At the same time, addresses 1 and 2 are added to the list of AssignedEdges. In Figure 6(c), following the adjacent list of vertex 6, $g(7)$ is set to 3 and $g(8)$ is set to 4, implying that addresses 4, 5 and 7 must be assigned to edges $\{6, 7\}$, $\{6, 8\}$ and $\{7, 8\}$, respectively. Finally, in Figure 6(d), following the adjacent list of vertex 4, $g(2)$ cannot be assigned to 5 because the sum $g(2) + g(4)$ would cause a reassignment with the already assigned address 7 to edge $\{7, 8\}$, so the next g value 6 is used instead, and the assignments $g(2) = 6$ and $g(3) = 7$ are made, implying that addresses 8, 9 and 13 must be assigned to edges $\{4, 2\}$, $\{4, 3\}$ and $\{2, 3\}$, respectively. This finishes the algorithm with AssignedEdges $= \{1, 2, 4, 5, 7, 8, 9, 13\}$.

A pseudo code for the assignment of values to critical vertices is presented in Figure 7. For all edges $e = \{u, w\} \in E$, $g(u) + g(w)$ must be unique. If this constraint is not forced then two different keys from S will be mapped in the same hash table location. Thus, the AssignedEdges array is used to force that $g(u) + g(w)$ will be distinct for all edges in E, as shown in line 18 of Figure 7. The variable Nextg represents $g(u)$.

Now we define certain complexity measures used hereinafter:

1. Let $I(u)$ be the number of iterations occurred in the repeat-until loop from line 13 until line 19, when vertex u is assigned.

```
procedure CriticalVerticesAssignment(G, G_crit, g, AssignedEdges)
    for v ∈ V_crit do g(v) := −∞;
    for i := 0 to m − 1 do AssignedEdges[i] = false;
    for v ∈ V_crit do
        if g(v) = −∞ then traverseBfs (G, v, G_crit, g, AssignedEdges);

procedure traverseBfs (G, v, G_crit, g, AssignedEdges)
     Nextg := 0;
     g(v) := Nextg;
     EnQueue (v, Q);
     while Q ≠ ∅ do
        v := DeQueue (Q);
        for u ∈ Adj (v) and g(u) = −∞ do
           AssignedVertices := ∅;
           for w ∈ Adj (u) and g(w) ≠ −∞ do AssignedVertices := AssignedVertices ∪ {w};
           repeat
              NoAssignedEdges := true;
              Nextg := Nextg + 1;
              for w ∈ AssignedVertices and NoAssignedEdges = true do
                 if (Nextg + g(w)) ≥ m then restart mapping step;
                 if AssignedEdges[Nextg + g(w)] = true then NoAssignedEdges := false;
           until NoAssignedEdges = true;
           g(u) := Nextg; {set the g value to vextex u and change g(u) from −∞ to Nextg}
           for w ∈ AssignedVertices do AssignedEdges[Nextg + g(w)] := true;
           EnQueue (u, Q);
```

Figure 7: The critical vertices assignment algorithm.

2. Let N_t be the number of times that AssignedEdges$[g(u) + g(w)]$ is true in the procedure CriticalVerticesAssignment. Thus,

$$N_t = \sum_{u=1}^{|V_{crit}|} (I(u) - 1) \qquad (5)$$

Maximal Value Assigned to An Edge

In this section we present the following conjecture.

Conjecture 1 For a random graph G with $|E_{crit}| = 0.5n$ and $|V| = 1.15n$, it is always possible to generate a MPHF because the maximal value A_{max} assigned to an edge $e \in E_{crit}$ is at most $m - 1$ (A_{max} corresponds to the maximal value generated by the assignment of values to critical vertices in Eq. (3).)

Next, we present two auxiliary theorems that will help us in the discussion of Conjecture 1.

Theorem 1 The number of back edges N_{bedges} of a random graph $G = G_{crit} \cup G_{ncrit}$ is given by: $N_{bedges} = |E_{crit}| - |V_{crit}| + 1$.

Proof: In an undirected graph G, every edge of G is either a tree edge or a back edge. In the subgraph G_{ncrit} there are no back edges because it is an acyclic graph. As shown by Erdös and A. Rényi [4, 5], when n tends to infinity the random graph G forms, with probability tending to 1, a giant component containing all cycles of G. So considering that G_{crit} is connected, the number of tree edges is $|V_{crit}| - 1$. It happens because we have only one tree connecting all vertices in V_{crit}. As the total number of edges in G_{crit} is $|E_{crit}|$ then $N_{bedges} + (|V_{crit}| - 1) = |E_{crit}|$. Thus,

$$N_{bedges} = |E_{crit}| - |V_{crit}| + 1. \;\square$$

Theorem 2 The maximal value A_{max} assigned to an edge $e \in E_{crit}$ in the assignment of values to critical vertices is: $A_{max} \leq 2|V_{crit}| - 3 + 2N_t$.

Proof: We start the assignment of values to critical vertices using the sequence $\{0, 1, \ldots \text{Nextg}\}$ so that each edge receives the sum of the values associated with its endpoints. The g value for each vertex u in V_{crit} is assigned only once. It happens because a g value is assigned to a vertex u if and only if $g(u) = -\infty$. Thus, after $g(u)$ change from $-\infty$ to the value stored in Nextg, the condition $g(u) = -\infty$ becomes false and $g(u)$ will not be assigned again. Consider now two possibilities: (i) If $N_t = 0$ then the g values will be assigned to vertices in V_{crit} sequentially. Therefore, the greatest and the second greatest values assigned to u and $w \in V_{crit}$ are $g(u) = |V_{crit}| - 1$ and $g(w) = |V_{crit}| - 2$, respectively. Thus, $A_{max} \leq (|V_{crit}|-1)+(|V_{crit}|-2)$ since the edge $\{u, w\}$ may be in E_{crit}, in the worst case. (ii) If $N_t > 0$ then Nextg is incremented by one for each time the condition AssignedEdges[Nextg $+ g(w)$] is true, as shown in line 15 of Figure 7. Thus, in the worst case,

$$\begin{aligned} A_{max} &\leq (|V_{crit}| - 1 + N_t) + (|V_{crit}| - 2 + N_t) \\ A_{max} &\leq 2|V_{crit}| - 3 + 2N_t. \square \end{aligned}$$

Let us now resume the discussion of Conjecture 1. Let us consider that $N_t \leq N_{bedges}$ when the average degree of vertices (d_{avg}) in G_{crit} is a constant. Substituting $N_t \leq N_{bedges}$ in Theorem 2 gives:

$$A_{max} \leq 2|V_{crit}| - 3 + 2N_{bedges}$$

Substituting the value of N_{bedges} from Theorem 1 gives:

$$A_{max} \leq 2|V_{crit}| - 3 + 2(|E_{crit}| - |V_{crit}| + 1)$$

Applying Definition 6 in G_{crit} we obtain $d_{avg} = 2|E_{crit}|/|V_{crit}|$. This implies that $|E_{crit}| = \frac{d_{avg}}{2}|V_{crit}|$. Thus,

$$\begin{aligned} A_{max} &\leq 2|V_{crit}| - 3 + 2\left(\frac{d_{avg}}{2}|V_{crit}| - |V_{crit}| + 1\right) \\ &\leq 2|V_{crit}| - 3 + (d_{avg} - 2)|V_{crit}| + 2 \\ &\leq d_{avg}|V_{crit}| - 1 \\ &\leq \frac{2|E_{crit}|}{|V_{crit}|}|V_{crit}| - 1 \\ &\leq 2|E_{crit}| - 1 \end{aligned}$$

As $|E_{crit}| = 0.5n$ and $n = m$ then $A_{max} \leq n - 1 \leq m - 1$.

We now show evidences that $N_t \leq N_{bedges}$ when d_{avg} is a constant. As shown in Section 5.1, $|E_{crit}| \leq 0.5n$ with probability tending to 1 when n increases. So, in order to obtain the average degree d_{avg} of vertices in G_{crit} we empirically determined that $|V_{crit}| \leq 0.35|V|$. As $|V| = 1.15n$ then $|V_{crit}| \leq 0.403n$. Table 2 presents the probability $P_{|V_{crit}|}$ that $|V_{crit}| \leq 0.403n$. As $P_{|V_{crit}|}$ tends to 1 when n increases then, $d_{avg} = 2 \times 0.5n/0.403n = 2.48$ is a constant value. We built 10,000 graphs for each number of keys.

n	$P_{\|V_{crit}\|}$	
	VLC2	TodoBR
$1,000$	0.51	0.52
$10,000$	0.76	0.77
$100,000$	0.98	0.98
$1,000,000$	1.00	1.00

Table 2: Probability $P_{|V_{crit}|}$ that $|V_{crit}| \leq 0.403n$ for different number of keys for the collections VLC2 and TodoBR.

Finally, we show experimental evidences that $N_t \leq N_{bedges}$. The expected values for $|V_{crit}|$ and $|E_{crit}|$ are $0.403n$ and $0.5n$, respectively. Then, by Theorem 1, $N_{bedges} = 0.5n - 0.403n + 1 = 0.097n + 1$. In Table 3 we show the maximal value of N_t obtained during 10,000 executions of the new algorithm for different sizes of S. As shown in Table 3, the maximal value of N_t is smaller than $N_{bedges} = 0.097n + 1$. So, Conjecture 1 is correct for $c = 1.15$.

n	Maximal value of N_t	
	VLC2	TodoBR
$1,000$	$0.085n$	$0.093n$
$10,000$	$0.067n$	$0.069n$
$100,000$	$0.061n$	$0.061n$
$1,000,000$	$0.059n$	$0.059n$

Table 3: The maximal value of N_t for different sizes of S for the collections VLC2 and TodoBR.

Complexity Analysis

We now show that the time complexity of the pseudo code presented in Figure 7 is $O(|V_{crit}|)$. For each unassigned vertex u, Adj(u) must be scanned with complexity |Adj(u)| in order to obtain (in AssignedVertices) the adjacencies of u that have already been assigned, as shown in lines 11 and 12. For each iteration of the repeat-until loop, |AssignedVertices| vertices must be scanned, as shown from lines 13 to 19. As each critical vertex is assigned only once and |AssignedVertices| vertices must be scanned to update the AssignedEdges array (as shown in line 21), the time complexity is given by

$$C(|V_{crit}|) = \sum_{u=1}^{|V_{crit}|} [\, |\mathrm{Adj}(u)| + (I(u) \times |\mathrm{AssignedVertices}|) + |\mathrm{AssignedVertices}| \,]$$

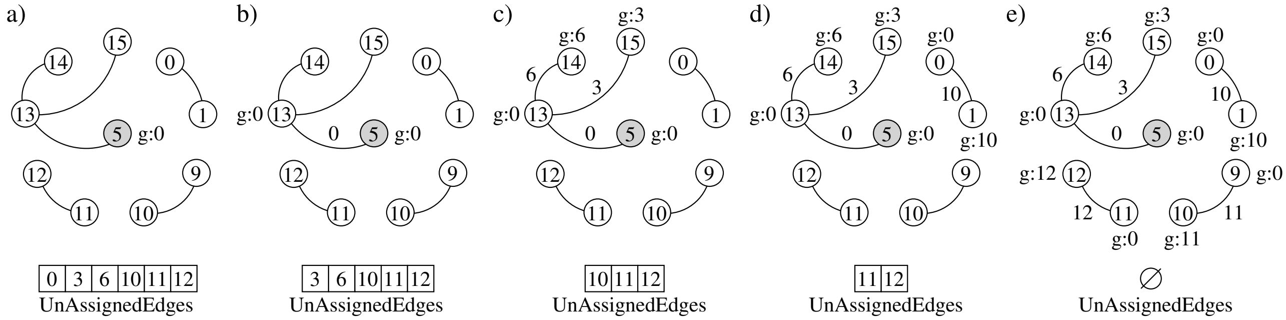


Figure 8: Example of the non critical vertices assignment.

```
procedure NonCriticalVerticesAssignment (G, G_ncrit, AssignedEdges, g)
    for i := 0 to m − 1 do
      if AssignedEdges[i] = false then UnAssignedEdges := UnAssignedEdges ∪ {i};
    for v ∈ V_scrit do traverseDfs (G, v, G_ncrit, g(v), g, UnAssignedEdges);
    for v ∈ V_ncrit and g(v) = −∞ do traverseDfs (G, v, G_ncrit, 0, g, UnAssignedEdges);

procedure traverseDfs (G, v, G_ncrit, gValue, g, unAssignedEdges)
    g(v) := gValue;
    for u ∈ Adj (v) and g(u) = −∞ do
       gValue := NextUnusedAddress(UnAssignedEdges) − g(v);
       traverseDfs (G, u, G_ncrit, gValue, g, UnAssignedEdges);
```

Figure 9: The algorithm to assign values to non critical vertices.

Considering that $|\mathrm{Adj}(u)| = d_{avg}$ on average and that $|\mathrm{AssignedVertices}| \leq |\mathrm{Adj}(u)|$, then $|\mathrm{AssignedVertices}| \leq d_{avg}$. Thus,

$$C(|V_{crit}|) \leq \sum_{u=1}^{|V_{crit}|} (d_{avg} + (I(u) \times d_{avg}) + d_{avg}) \qquad (6)$$

As presented before, d_{avg} is a constant and $N_t \leq N_{bedges}$. Therefore, applying $d_{avg} = 2|E_{crit}|/|V_{crit}|$ in Theorem 1 gives:

$$\begin{aligned} N_t &\leq |E_{crit}| - |V_{crit}| + 1 \\ &\leq \frac{d_{avg}}{2}|V_{crit}| - |V_{crit}| + 1 \\ &\leq \left(\frac{d_{avg}}{2} - 1\right)|V_{crit}| + 1 \end{aligned}$$

Since d_{avg} is a constant then $N_t = O(|V_{crit}|)$. The number of times that $\mathrm{AssignedEdges}[\mathrm{Nextg} + g(w)] = \mathit{true}$ is given by Eq. (5). Thus, $I(u)$ must be a constant because

$$\sum_{u=1}^{|V_{crit}|} (I(u) - 1) = N_t = O(|V_{crit}|).$$

Since $I(u)$ and d_avg in Eq. (6) are constants, we have that $C(|V_{crit}|) = O(|V_{crit}|)$. As $|V_{crit}| \leq |V|$ and $|V| = cn$, the time complexity of the assignment of values to critical vertices is $O(n)$.

5.3.2 Assignment of Values to Non Critical Vertices

The procedure NonCriticalVerticesAssignment (G, G_{ncrit}, AssignedEdges, g) receives G, G_{ncrit} and AssignedEdges as input and produces the assignment of values to vertices in G_{ncrit}, represented by the array g. This finishes the perfect assignment of values to vertices of G. We use a depth-first search algorithm to assign values to vertices in G_{ncrit}.

As G_{ncrit} is acyclic, we can impose the order in which addresses are associated with edges in G_{ncrit}. Therefore, in the assignment of values to vertices in G_{ncrit} we place the unused addresses in the gaps left by the assignment of values to vertices in G_{crit}. For that, we start the depht-first search from the vertices in V_{scrit} because these critical vertices were already assigned, so their g values can not be changed.

Considering the subgraph G_{ncrit} in Figure 4(d), a step by step example of the assignment of values to vertices in G_{ncrit} is presented in Figure 8. Figure 8(a) presents the initial state of the algorithm. The critical vertex 5 is the only one that has non critical vertices as adjacent. In the example presented in Figure 6, the addresses $\{0, 3, 6, 10, 11, 12\}$ were not used. So, taking the first unused address 0 and the vertex 13, which is reached from the vertex 5, the g value of vertex 13 is set to $0 - g(5) = 0$, as shown in Figure 8(b). In Figure 8(c), using the unused addresses 3 and 6, the g values for vertices 15 and 14 are set to $3 - g(13) = 3$ and to $6 - g(13) = 6$, respectively. Vertices 0, 1, 9, 10, 11 and 12 were not assigned yet, so we continue the assignment of values to non critical vertices from vertex 0. In Figure 8(d), we set $g(0)$ to 0. The only vertex that is reached from vertex 0 is vertex 1, so taking the unused address 10 we set $g(1)$ to $10 - g(0) = 10$. This process is repeated until the UnAssignedEdges list becomes empty. The final result is shown in Figure 8(e).

A pseudo code for the assignment of values to non critical vertices is presented in Figure 9.

Complexity Analysis

The assignment of values to vertices in G_{ncrit} is a depth-first search algorithm. Then, its time complexity is $O(|V_{scrit}| + |V_{ncrit}| + |E_{ncrit}|)$. Considering that $|V_{ncrit}| \leq |V|$, $|V_{scrit}| \leq |V|$, $|V| = cn$ and $|E_{ncrit}| \leq n$, the complexity of the assignment of values to non critical vertices is $O(n)$.

6 MPHF Evaluation

Figure 10 presents a pseudo code to evaluate the MPHF generated by the new algorithm. The procedure h (x, g, h_1, h_2) receives as input a key $x \in S$, the g function, the tables used by h_1 and h_2 and returns the hash table address assigned to x.

```
procedure h (x, g, h1, h2)
    u := h1(x);
    v := h2(x);
    return (g(u) + g(v));
```

Figure 10: Evaluating the MPHF.

7 Experimental Results

In this section we present experimental results to show the efficiency of the new algorithm. Also, a comparison with algorithm CHM (proposed by Czech, Havas and Majewski [1]) is made.

The two algorithms were implemented in the C language. All experiments were carried out on a computer running the Linux operating system, version 2.6.7, with a 2.2 gigahertz Athlon processor and 1 gigabyte of main memory.

Collection	n	Key Size (Avg)
TodoBR	3,541,615	8.3
Random	10,000,000	20.0
VLC2	10,935,900	8.6
URLs	20,000,000	57.4

Table 4: Collections used in the experiments.

We used four collections in the experiments: (i) the vocabulary of the TodoBR search engine (http://www.todobr.com.br); (ii) a collection of keys generated randomly (Random); (iii) the vocabulary extracted from the TREC-VLC2 (Very Large Collection 2) collection [11]; (iv) a set of URLs crawled from the Web. Table 4 presents some details about the collections.

Table 5 presents the main characteristics of the two algorithms. The number of edges of graph $G = (V, E)$ is equal to the size n of the set S of keys for the two algorithms. The number of vertices of G is equal to $1.15n$ and $2.09n$ for the new algorithm and the CHM algorithm, respectively. This measure is related to the amount of space to store the array g. The number of critical edges is $0.5|E|$ and 0, for the new algorithm and the CHM algorithm, respectively.

Characteristics	Algorithms	
	New algorithm	CHM
$\|E\|$	n	n
$\|V\|$	cn	cn
c	1.15	2.09
$\|g\|$	$1.15n$	$2.09n$
$\|E_{crit}\|$	$0.5\|E\|$	0
G	cyclic	acyclic
Order preserving	no	yes

Table 5: Main characteristics of the algorithms.

Table 6 presents time results for constructing MPHFs using the two algorithms. The table entries

	New algorithm, $c = 1.15$					CHM, $c = 2.09$			
Collection	N_i	Mapping	Ordering	Searching	Total	N_i	Mapping + Ordering	Searching	Total
TodoBR	1.92	11.33	1.93	0.97	14.23	2.63	19.51	3.03	22.54
Random	1.77	41.90	7.17	3.70	52.77	2.96	59.92	10.31	70.23
VLC2	2.24	44.69	7.00	3.59	55.28	2.94	78.77	11.09	89.86
URLs	2.18	153.23	14.62	7.52	175.37	-	-	-	-

Table 6: Time to generate the MPHFs for the new algorithms.

represent averages over 50 trials. The column labelled as N_i represents the number of iterations to generate the random graph G in the mapping step of the algorithms. The other columns represent the run times for each step of the algorithms. All times are in seconds.

The CHM algorithm performs the ordering step together the mapping step. In the CHM algorithm the ordering step is just the assignment of hash values to the edges of G.

The mapping step of the new algorithm is faster because the number of iterations to generate G is lower, since G has $1.15n$ vertices and must not be acyclic. This result fully backs the theoretical considerations. Using Eq. (4), the expected number of iterations to generate G for the new algorithm is 2.13 and using Eq. (2), the same measure is 2.92 for the CHM algorithm. The CHM algorithm also needs to verify if G is acyclic during the mapping step, which has the same complexity of the ordering step of the new algorithm.

The random graph G generated in the mapping step of the new algorithm has $1.15n$ vertices and the one generated in the mapping step of the CHM algorithm has $2.09n$ vertices. That is why the searching step of new algorithm is faster, since the time complexity of the searching step of the algorithms depends on the number of vertices in G.

We were not able to generate a MPHF for the CHM algorithm using the URLs collection. The reason was that its random graph G has more vertices ($|V| = 2.09n$) and could not be stored in the main memory of the machine used for the experiments.

The MPHF generated by the new algorithm is slightly faster than the one generated by the CHM algorithm. It happens because we save a module operation, as shown in Eq. (3). Table 7 presents the evaluation times, which are averages over 50 trials. Each entry in Table 7 represents the time to evaluate all keys of each collection.

Finally, Figure 11 presents the time to generate the MPHF by the new algorithm for different number of keys of the TREC-VLC2 collection. As claimed, the time to generate a MPHF using the new algorithm grows linearly with n.

Collection	Algorithms	
	New algorithm	CHM
TodoBR	3.33	3.59
Random	12.29	13.70
VLC2	12.41	13.81
URLs	60.03	-

Table 7: Time to compute a hash table entry for the algorithms considered. All times are in seconds.

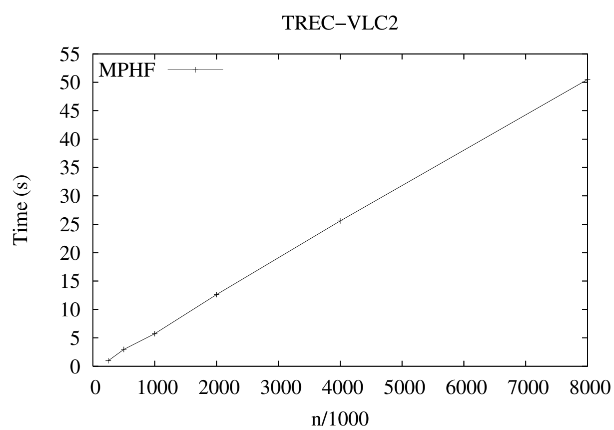


Figure 11: Verification of the $O(n)$ complexity to generate a MPHF by the new algorithm.

8 Conclusions

A new algorithm for generating MPHFs has been proposed. Its expected time complexity is $O(n)$, so that the new algorithm is time optimal. The time to evaluate the generated function is very fast and the space needed to store it is $O(n \log n)$ bits. Experimental results show that the times to both generate the MPHF and compute a hash table entry by the new algorithm are better than the times obtained by the CHM algorithm, one of the fastest known algorithm.

References

[1] 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.

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

[3] P. Erdos and A. Rényi. On random graphs. *Pubicationes Mathematicae*, 6:290–297, 1959.

[4] P. Erdös and A. Rényi. On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 56:17–61, 1960.

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

[6] P. Flajolet, D. E. Knuth, and B. Pittel. The first cycles in an evolving graph. *Discrete Math*, 75:167–215, 1989.

[7] E. A. Fox, Q. F. Chen, A. M. Daoud, and L. S. Heath. Order preserving minimal perfect hash functions and information retrieval. *ACM Trans. Inform. Systems*, 9(3):281–308, July 1991.

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

[9] 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.

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

[11] D. Hawking. Overview of trec-7 very large collection track (draft for notebook), 1998.

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

[13] 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.

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

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