A Practical Minimal Perfect Hashing Method

Fabiano C. Botelho,[1] Yoshiharu Kohayakawa,[2] and Nivio Ziviani[1]

[1] Dept. of Computer Science, Federal Univ. of Minas Gerais, Belo Horizonte, Brazil
{fbotelho,nivio}@dcc.ufmg.br
[2] Dept. of Computer Science, Univ. of São Paulo, São Paulo, Brazil
yoshi@ime.usp.br

Abstract. We propose a novel algorithm based on random graphs to construct minimal perfect hash functions h. For a set of n keys, our algorithm outputs h in expected time $O(n)$. The evaluation of $h(x)$ requires two memory accesses for any key x and the description of h takes up $1.15n$ words. This improves the space requirement to 55% of a previous minimal perfect hashing scheme due to Czech, Havas and Majewski. A simple heuristic further reduces the space requirement to $0.93n$ words, at the expense of a slightly worse constant in the time complexity. Large scale experimental results are presented. For a collection of 100 million keys, each key 63 bytes long on average, our algorithm finds a minimal perfect hash function in 811 seconds on average.

1 Introduction

Suppose U is a universe of *keys*. Let $h : U \to M$ be a *hash function* that maps the keys from U to a given interval of integers $M = [0, m-1] = \{0, 1, \ldots, m-1\}$. Let $S \subseteq U$ be a set of n keys from U. Given a key $x \in S$, the hash function h computes an integer in $[0, m-1]$ for the storage or retrieval of x in a *hash table*. Hashing methods for *non-static sets* of keys can be used to construct data structures storing S and supporting membership queries "$x \in S$?" in expected time $O(1)$. However, they involve a certain amount of wasted space owing to unused locations in the table and waisted time to resolve collisions when two keys are hashed to the same table location.

For *static sets* of keys it is possible to compute a function to find any key in a table in one probe; such hash functions are called *perfect*. More precisely, given a set of keys S, we shall say that a hash function $h : U \to M$ is a *perfect hash function* for S if h is an injection on S, that is, there are no *collisions* among the keys in S: if x and y are in S and $x \neq y$, then $h(x) \neq h(y)$. Figure 1(a) illustrates a perfect hash function. Since no collisions occur, each key can be retrieved from the table with a single probe. If $m = n$, that is, the table has the same size as S, then we say that h is a *minimal perfect hash function* for S. Figure 1(b) illustrates a minimal perfect hash function. Minimal perfect hash functions totally avoid the problem of wasted space and time.

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.

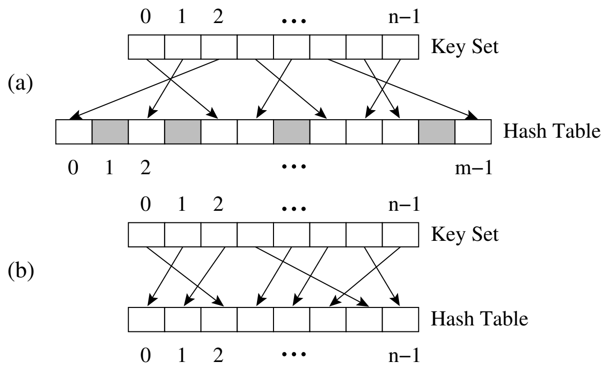

Fig. 1. (a) Perfect hash function (b) Minimal perfect hash function

The aim of this paper is to describe a new way of constructing minimal perfect hash functions. Our algorithm shares several features with the one due to Czech, Havas and Majewski [4]. 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 $c \geq 1.15$, and hence G necessarily contains cycles, while they generate *acyclic* random graphs $G = (V, E)$ with $|V| = cn$ and $|E| = |S| = n$, with a greater number of vertices: $|V| \geq 2.09n$; (ii) they generate order preserving minimal perfect hash functions while our algorithm does not preserve order (a perfect hash function h is *order preserving* if the keys in S are arranged in some given order and h preserves this order in the hash table). 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 function h with a very economical description. As the algorithm in [4], our algorithm produces h in $O(n)$ expected time for a set of n keys. The description of h requires $1.15n$ computer words, and evaluating $h(x)$ 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.

2 Related Work

Czech, Havas and Majewski [5] provide a comprehensive survey of the most important theoretical results on perfect hashing. In the following, we review some of those results.

Fredman, Komlós and Szemerédi [10] showed that it is possible to construct space efficient perfect hash functions that can be evaluated in constant time with table sizes that are linear in the number of keys: $m = O(n)$. In their model of computation, an element of the universe U fits into one machine word, and arithmetic operations and memory accesses have unit cost. Randomized algorithms in the FKS model can construct a perfect hash function in expected time $O(n)$: this is the case of our algorithm and the works in [4, 14].

Many methods for generating minimal perfect hash functions use a *mapping*, *ordering* and *searching* (MOS) approach, a description coined by Fox, Chen and Heath [9]. 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. Our algorithm and the algorithm presented in [4] use the MOS approach.

Pagh [14] proposed a family of randomized algorithms for constructing minimal perfect hash functions. The form of the resulting function is $h(x) = (f(x) + d_{g(x)}) \bmod n$, where f and g are universal hash functions 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 $(2+\epsilon)n$ computer words. Dietzfelbinger and Hagerup [6] improved [14], reducing from $(2+\epsilon)n$ to $(1+\epsilon)n$ the number of computer words 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. Differently of the works in [14, 6], our algorithm uses two universal hash functions h_1 and h_2 randomly selected from a class of universal hash functions that do not need to meet any additional requirements.

The work in [4] presents an efficient and practical algorithm for generating order preserving minimal perfect hash functions. Their method involves the generation of acyclic random graphs $G = (V, E)$ with $|V| = cn$ and $|E| = n$, with $c \geq 2.09$. They showed that an order preserving minimal perfect hash function can be found 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, each set S has a corresponding graph $G = (V, E)$, where $V = \{0, 1, \ldots, t\}$ and $E = \big\{\{h_1(x), h_2(x)\} : x \in S\big\}$. In order to guarantee the acyclicity of G, the algorithm repeatedly selects h_1 and h_2 from a family of universal hash functions until the corresponding graph is acyclic. Havas et al. [11] proved that if $|V(G)| = cn$ and $c > 2$, then the probability that G is acyclic is $p = e^{1/c}\sqrt{(c-2)/c}$. For $c = 2.09$, this probability is $p \simeq 0.342$, and the expected number of iterations to obtain an acyclic graph is $1/p \simeq 2.92$.

3 The Algorithm

Let us show how the minimal perfect hash function h will be constructed. We make use of two auxiliary random functions h_1 and $h_2 : U \to V$, where $V = [0, t-1]$ for some suitably chosen integer $t = cn$, where $n = |S|$. We build a random graph $G = G(h_1, h_2)$ on V, whose edge set is $\big\{\{h_1(x), h_2(x)\} : x \in S\big\}$. There is an edge in G for each key in the set of keys S.

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., [1, 12]). 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 $h(x) = g(h_1(x)) + g(h_2(x))$ ($x \in S$) is a minimal perfect hash function 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 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_crit, G_ncrit);
    Searching (G, G_crit, G_ncrit, g);
```

Fig. 2. Main steps of the algorithm for constructing a minimal perfect hash function

3.1 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, $h_2 : U \to [0, t-1]$.

The functions h_1 and h_2 are constructed as follows. We impose some upper bound L on the lengths of the keys in S. To define h_j ($j = 1,2$), we generate an $L \times \Sigma$ table of random integers table_j. For a key $x \in S$ of length $|x| \leq L$ and $j \in \{1, 2\}$, we let

$$h_j(x) = \Big(\sum_{i=1}^{|x|} \text{table}_j[i, x[i]] \Big) \bmod t.$$

The random graph $G = G(h_1, h_2)$ has vertex set $V = [0, t-1]$ 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 $h_2(x) = (2h_1(x) + 1) \bmod t$ 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 $|V| = t$ and $|E| = n$ be a random graph in the uniform model $\mathcal{G}(t, n)$, the model in which all the $\binom{\binom{t}{2}}{n}$ graphs on V with n edges are equiprobable. The study of $\mathcal{G}(t, n)$ goes back to the classical work of

Erdös and Rényi [7, 8, 13] (for a modern treatment, see [1, 12]). Let $d = 2n/t$ be the average degree of G. It is well known that, if $d > 1$, or, equivalently, if $c < 2$ (recall that we have $t = cn$), then, almost every G contains[3] a "giant" component of order $(1+o(1))bt$, 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 $O(\log t)$ 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 $o(t)$ almost surely.

Pittel and Wormald [15] 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. Experiments show that this random graph, conditioned on being simple, strongly resembles the random graph from the uniform model $\mathcal{G}(t, n)$. In what follows, we work under the hypothesis that $G = G(h_1, h_2)$ is drawn from $\mathcal{G}(t, n)$. Thus, following [15], the number of vertices of G_{crit} is

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

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

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

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 $|E(G_{\text{crit}})| \le \frac{1}{2}|E(G)| = \frac{1}{2}|S| = n/2$. The crucial step now is to determine the value of c (in $t = cn$) to obtain a random graph $G = G_{\text{crit}} \cup G_{\text{ncrit}}$ with $|E(G_{\text{crit}})| \le \frac{1}{2}|E(G)|$.

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

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

[3] 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 $t \to \infty$'.

c	URLs (n)						
	1,000	10,000	100,000	1,000,000	2,000,000	3,000,000	4,000,000
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. Probability $P_{|E_{\mathrm{crit}}|}$ that $|E(G_{\mathrm{crit}})| \leq n/2$ for different c values and different number of keys for a collections of URLs

To estimate p, we estimate the probability of obtaining n *distinct* objects when we independently draw n objects from a universe of cardinality $\binom{t}{2} = \binom{cn}{2} \sim c^2n^2/2$, with replacement. This latter probability is about $e^{-\binom{n}{2}/\binom{t}{2}}$ for large n. As $e^{-\binom{n}{2}/\binom{t}{2}} \to e^{-1/c^2} > 0$ as $n \to \infty$, the expected number of iterations is $e^{1/c^2} = 2.13$ (recall $c = 1.15$). As the expected number of iterations is $O(1)$ then, the mapping step takes $O(n)$ time.

3.2 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 $G = G_{\mathrm{crit}} \cup G_{\mathrm{ncrit}}$.

Figure 3 presents a sample graph with 9 vertices and 8 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 3(a), $Q = \{2\}$ after the initialization step.

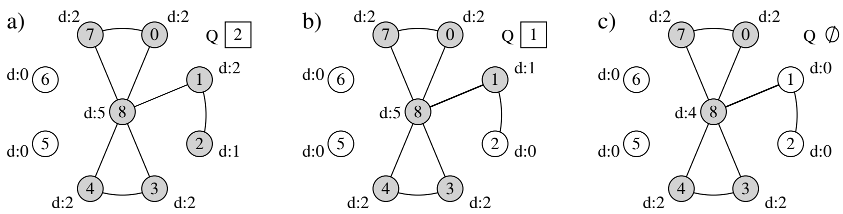

Fig. 3. Ordering step for a graph with 9 vertices and 8 edges

Next, we remove one vertex v from the queue, decrement its degree and the degree of the vertices with degree greater than 0 in the adjacent list of v, as depicted in Figure 3(b) for $v = 2$. At this point, the adjacencies of v with degree 1 are inserted into the queue, such as vertex 1. 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 3(c). Finally, to determine the vertices in V_{scrit} we collect all vertices $v \in V(G_{\mathrm{crit}})$ with at least one vertex u that is in Adj(v) and in $V(G_{\mathrm{ncrit}})$, as the vertex 8 in Figure 3(c).

Analysis of the Ordering Step. The initialization takes $O(|V(G)|)$ time, because we need to check the degree of each vertex from $V(G)$. Next, the non-critical vertices are obtained. This process takes $O(|V(G_{\mathrm{ncrit}})|)$, because each non-critical vertex is removed from the queue only once. Finally, to determine the vertices in V_{scrit} takes $O(|V(G_{\mathrm{crit}})|)$. Considering that $|V(G_{\mathrm{crit}})| \leq |V(G)|$, $|V(G_{\mathrm{ncrit}})| \leq |V(G)|$ and $|V(G)| = t = cn$, the ordering step takes $O(n)$ time.

3.3 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 $h : E(G) \to \mathbb{Z}$ defined by

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

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 $|E(G)| - 1 = n - 1$, and thus we have a bijection between S and $[0, n - 1]$.

The procedure Searching $(G, G_{\text{crit}}, G_{\text{ncrit}}, g)$ receives as input G, G_{crit}, G_{ncrit} and finds a suitable $\log_2 |V(G)| + 1$ 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 3(c), a step by step example of the assignment of values to vertices in G_{crit} is presented in Figure 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 4(a) is chosen, the assignment $g(8) = 0$ is made and x is set to 1.

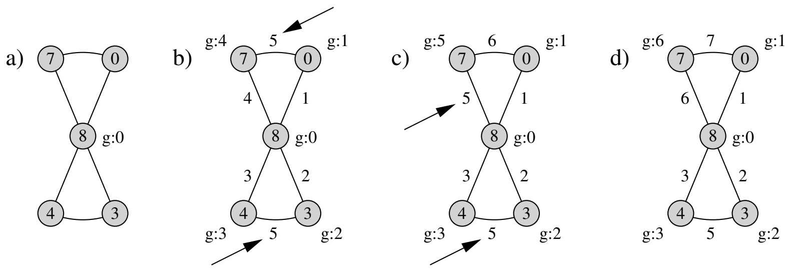

Fig. 4. Example of the assignment of values to critical vertices

In Figure 4(b), following the adjacent 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 4(c). Since $g(8)+5 = 5 \in A_E$, x is again incremented by 1 and set to 6, as depicted in Figure 4(d). These two reassignments are indicated by the arrows in Figure 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.

A pseudo code for the assignment of values to critical vertices is presented in the Appendix.

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 were already assigned and cannot be changed.

Considering the subgraph G_{ncrit} in Figure 3(c), a step by step example of the assignment of values to vertices in G_{ncrit} is presented in Figure 5. Figure 5(a) presents the initial state of the algorithm. The critical vertex 8 is the only one that has non-critical vertices as adjacent. In the example presented in Figure 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 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 5(c). This process is repeated until the UnAssignedAddresses list becomes empty.

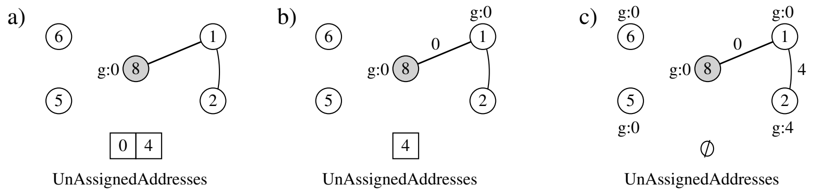

Fig. 5. Example of the assignment of values to non-critical vertices

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

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 $v \in V(G_{\mathrm{crit}})$.

Combining the fact that G_{crit}, the 2-core of G, is almost surely connected for $c = 1.15$ (see [1, 12, 15]) and the fact that every edge is either a tree edge or a back edge (see, e.g., [3]), we see that the following holds.

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

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

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

Proof. (Sketch) The assignment of g values to critical vertices starts from 0, and each edge e receives the label $h(e)$ as given by Eq. (3). The g value for each vertex v in $V(G_{\mathrm{crit}})$ is assigned only once. A little thought shows that $\max_v g(v) \leq |V(G_{\mathrm{crit}})| - 1 + N_t$, where the maximum is taken over all vertices v in $V(G_{\mathrm{crit}})$. Moreover, two distinct vertices get distinct g values. Hence, $A_{\max} \leq (|V(G_{\mathrm{crit}})| - 1 + N_t) + (|V(G_{\mathrm{crit}})| - 2 + N_t) \leq 2|V(G_{\mathrm{crit}})| - 3 + 2N_t$, as required. □

Maximal Value Assigned to An Edge. In this section we present the following conjecture.

Conjecture 1. *For a random graph G with $|E(G_{\mathrm{crit}})| \leq n/2$ 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 $e \in E(G_{\mathrm{crit}})$ is at most $n - 1$.*

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

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

We now show experimental evidence that $N_t \leq N_{\mathrm{bedges}}$. Considering Eqs (1) and (2), the expected values for $|V(G_{\mathrm{crit}})|$ and $|E(G_{\mathrm{crit}})|$ for $c = 1.15$ are $0.401n$ and $0.501n$, respectively. From Theorem 1, $N_{\mathrm{bedges}} = 0.501n - 0.401n + 1 = 0.1n + 1$. Table 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 $N_{\mathrm{bedges}} = 0.1n + 1$ and tends to $0.059n$ for $n \geq$ 1,000,000.

Time Complexity. We now show that the time complexity of determining $g(v)$ for all critical vertices $x \in V(G_{\mathrm{crit}})$ is $O(|V(G_{\mathrm{crit}})|) = O(n)$. For each unassigned vertex v, the adjacency list of v, which we call $\mathrm{Adj}(v)$, 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,

[4] Bollobás and Pikhurko [2] 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.

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

Table 3. The maximal value of N_t for different number of URLs

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 $d_{\mathrm{crit}} = 2|E(G_{\mathrm{crit}})|/|V(G_{\mathrm{crit}})|$ be the average degree of G_{crit}, note that $|Y| \leq |\mathrm{Adj}(v)|$, and suppose for simplicity that $|\mathrm{Adj}(v)| = O(d_{\mathrm{crit}})$. Then, putting all these together, we see that the time complexity of this procedure is

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

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

4 Experimental Results

We now present some experimental results. The same experiments were run with our algorithm and the algorithm due to Czech, Havas and Majewski [4], referred to as CHM algorithm. The two algorithms were implemented in the C language and are available at `https://sourceforge.net/projects/cmph/`. 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 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 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 necessarily contains cycles and the CHM algorithm generates acyclic random graphs. Finally, the CHM algorithm generates order preserving functions while our algorithm does not preserve order.

Table 5 presents time measurements. All times are in seconds. The table entries represent averages over 50 trials. The column labelled as N_i represents

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

Table 4. Main characteristics of the algorithms

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 5. Time measurements for our algorithm and the CHM algorithm

the number of iterations to generate the random graph G in the mapping step of the algorithms. The next columns represent the run times for the mapping plus ordering steps together and the searching step for each algorithm. The last column represents the percent gain of our algorithm over 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 59% 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 3.3). 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

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 6. Time measurements for our tuned algorithm with $c = 1.00$ and $c = 0.93$

expected number of iterations are 2.72 and 3.17, respectively (for $n = 12{,}500{,}000$, the number of iterations are 2.78 for $c = 1$ and 3.04 for $c = 0.93$). Table 6 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 5) to 101.74 seconds for $c = 1$ and to 102.19 seconds for $c = 0.93$.

Finally, we run 1,000,000 evaluations of the function generated by our algorithm for the set of 100 million URLs and it took 3.06 seconds. An equivalent time was obtained for the CHM algorithm.

5 Conclusion

We have presented a practical method to construct minimal perfect hash functions for static sets which is efficient and may be tuned to yield a function with a very economical description.

References

1. B. Bollobás. *Random graphs*, volume 73 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, second edition, 2001.
2. B. Bollobás and O. Pikhurko. Integer sets with prescribed pairwise differences being distinct. *European Journal of Combinatorics*. To Appear.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.
4. 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.
5. Z.J. Czech, G. Havas, and B.S. Majewski. Fundamental study perfect hashing. *Theoretical Computer Science*, 182:1–143, 1997.
6. M. Dietzfelbinger and T. Hagerup. Simple minimal perfect hashing in less space. In *The 9th European Symposium on Algorithms (ESA), volume 2161 of Lecture Notes in Computer Science*, pages 109–120, 2001.
7. P. Erdős and A. Rényi. On random graphs I. *Pub. Math. Debrecen*, 6:290–297, 1959.
8. P. Erdős and A. Rényi. On the evolution of random graphs. *Magyar Tud. Akad. Mat. Kutató Int. Közl.*, 5:17–61, 1960.
9. 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.
10. M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with O(1) worst case access time. *J. ACM*, 31(3):538–544, July 1984.
11. 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.
12. S. Janson, T. Łuczak, and A. Ruciński. *Random graphs*. Wiley-Inter., 2000.
13. P. Erdős and A. Rényi. On the strength of connectedness of a random graph. *Acta Mathematica Scientia Hungary*, 12:261–267, 1961.
14. R. Pagh. Hash and displace: Efficient evaluation of minimal perfect hash functions. In *Workshop on Algorithms and Data Structures*, pages 49–54, 1999.

15. B. Pittel and N. C. Wormald. Counting connected graphs inside-out. *Journal of Combinatorial Theory.* To Appear.

6 Appendix

```
procedure CriticalVerticesAssignment (G, G_crit, g, A_E)
  for v ∈ V_crit do g(v) := −∞;
  A_E := ∅;
  for v ∈ V_crit do if g(v) = −∞ then traverseBfs (G, v, G_crit, g, A_E);

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

Fig. 6. The critical vertices assignment algorithm

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

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

Fig. 7. The algorithm to assign values to non-critical vertices