Simple and Space-Efficient Minimal Perfect Hash Functions *

Fabiano C. Botelho[1], Rasmus Pagh[2] and Nivio Ziviani[1]

[1] Dept. of Computer Science, Federal Univ. of Minas Gerais, Belo Horizonte, Brazil
{fbotelho,nivio}@dcc.ufmg.br
[2] Computational Logic and Algorithms Group, IT Univ. of Copenhagen, Denmark
pagh@itu.dk

Abstract. A *perfect hash function* (PHF) $h : U \rightarrow [0, m-1]$ for a key set S is a function that maps the keys of S to unique values. The minimum amount of space to represent a PHF for a given set S is known to be approximately $1.44n^2/m$ bits, where $n = |S|$. In this paper we present new algorithms for construction and evaluation of PHFs of a given set (for $m = n$ and $m = 1.23n$), with the following properties:

1. Evaluation of a PHF requires constant time.
2. The algorithms are simple to describe and implement, and run in linear time.
3. The amount of space needed to represent the PHFs is around a factor 2 from the information theoretical minimum.

No previously known algorithm has these properties. To our knowledge, any algorithm in the literature with the third property either:

- Requires exponential time for construction and evaluation, or
- Uses near-optimal space only asymptotically, for extremely large n.

Thus, our main contribution is a scheme that gives low space usage for realistic values of n. The main technical ingredient is a new way of basing PHFs on random hypergraphs. Previously, this approach has been used to design simple PHFs with superlinear space usage[3].

* This work was supported in part by GERINDO Project–grant MCT/CNPq/CT-INFO 552.087/02-5, and CNPq Grants 30.5237/02-0 (Nivio Ziviani) and 142786/2006-3 (Fabiano C. Botelho)

[3] This version of the paper is identical to the one published in the WADS 2007 proceedings. Unfortunately, it does not give reference and credit to the paper *The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables*, by Chazelle et al., Proceedings of SODA 2004. They 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 paper 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.

1 Introduction

Perfect hashing is a space-efficient way of associating unique identifiers with the elements of a static set S. We will refer to the elements of S as *keys*. A *perfect hash function* (PHF) maps $S \subseteq U$ to unique values in the range $[0, m-1]$. We let $n = |S|$ and $u = |U|$ — note that we must have $m \geq n$. A *minimal perfect hash function* (MPHF) is a PHF with $m = n$. For simplicity of exposition, we consider in this paper the case $\log u \ll n$. This allows us to ignore terms in the space usage that depend on u.

In this paper we present a simple, efficient, near space-optimal, and practical family $\mathcal{F}$ of algorithms for generating PHFs and MPHFs. 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-uniform hypergraph is the generalization of a standard undirected graph where each edge connects $r \geq 2$ vertices. The idea of basing perfect hashing on random hypergraphs is not new, see e.g. [14], but we will proceed differently to achieve a space usage of $O(n)$ bits rather than $O(n \log n)$ bits. (As in previous constructions based on hypergraphs we assume that the hash functions used are uniformly random and have independent function values. However, we argue that our scheme can also be realized using explicitly defined hash functions using small space.) Evaluation time for all schemes considered is constant. For $r = 2$ we obtain a space usage of $(3 + \epsilon)n$ bits for a MPHF, for any constant $\epsilon > 0$. 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 $0.89n$ bits. The bounds for $r = 3$ assume a conjecture about the emergence of a 2-core in a random 3-partite hypergraph, whereas the bounds for $r = 2$ are fully proved. Choosing $r > 3$ does not give any improvement of these results.

We will argue that our method is far more practical than previous methods with proven space complexity, both because of its simplicity, and because the constant factor of the space complexity is more than 6 times lower than its closest competitor, for plausible problem sizes. We verify the practicality experimentally, using heuristic hash functions, and slightly more space than in the mentioned theoretical bounds.

2 Related Work

In this section we review some of the most important theoretical and practical results on perfect hashing. Czech, Havas and Majewski [4] provide a more comprehensive survey.

2.1 Theoretical Results

Fredman and Komlós [9] proved that at least $n \log e + \log\log u - O(\log n)$ bits are required to represent a MPHF (in the worst case over all sets of size n), provided that $u \geq n^\alpha$ for some $\alpha > 2$. Logarithms are in base 2. Note that the two last terms are negligible under the assumption $\log u \ll n$. 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 [18].

Mehlhorn [15] showed that the Fredman-Komlós bound is 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 from practice because its construction and evaluation time is exponential in n.

Schmidt and Siegel [19] 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 [10]. 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 [19] 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 [11] 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 [11] 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. Thus, 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.

2.2 Practical Results

We now describe some of the main "practical" results that our work is based on. They are characterized by simplicity and (provably) low constant factors.

The first two results assume uniform random hash functions to be available for free. Czech et al [14] proposed a family of algorithms to construct MPHFs based on r-uniform hypergraphs (i.e., with edges of size r). The resulting functions can be evaluated in $O(1)$ time and stored in $O(n \log n)$ bits. Botelho, Kohayakawa and Ziviani [3] improved the space requirement of one instance of the family considering $r = 2$, but the space requirement is still $O(n \log n)$ bits. In both cases, the MPHF can be generated in expected $O(n)$ time. It was found experimentally in [3] that their construction procedure works well in practice.

Pagh [16] proposed an algorithm for constructing MPHFs of the form $h(x) = (f(x) + d[g(x)]) \bmod n$, where f and g are randomly chosen from a family of universal hash functions, and d is a vector of "displacement values" that are used to resolve collisions that are caused by the function f. The scheme is simple and evaluation of the functions very fast, but the space usage is $(2 + \epsilon)n \log n$ bits, which is suboptimal. Dietzfelbinger and Hagerup [5] improved [16], reducing from the space usage to $(1 + \epsilon)n \log n$ bits, still using simple hash functions. Woelfel [20] 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.

2.3 Heuristics

Fox et al. [7, 8] presented several algorithms for constructing MPHFs that in experiments require between 2 and 8 bits per key to be stored. However, it is shown in [4, Section 6.7] that their algorithms have exponential running times in expectation. Also, there is no warranty that the number of bits per key to store the function will be fixed as n increases. The work by Lefebvre and Hoppe [13] has the same problem. They have designed a PHF method to specifically represent sparse spatial data and the resulting functions requires more than 3 bits per key to be stored.

3 A Family of Minimal Perfect Hashing Methods

In this section we present our family $\mathcal{F}$ of algorithms for constructing near space-optimal MPHFs. The basic idea is as follows. 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 $r \geq 2$. Note that each key in S is associated with an edge in $E(G_r)$. Also in the Mapping Step, we order the edges of G_r into a list L such that each edge e_i contains a vertex that is not incident to any edge that comes after e_i in L. The next 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 $V(G_r)$. If we desire a PHF with a smaller range ($n < |V(G_r)|$), we subsequently map the assigned vertices of $V(G_r)$ to $[0, n - 1]$. This mapping is produced by the

Ranking Step, which creates a data structure that allows us to compute the *rank* of any assigned vertex of $V(G_r)$ 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 6 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. The PHF $p : S \to V(G_r)$ produced by the Assigning Step has the form

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

The function $g : V(G_r) \to \{0, 1, \ldots, r\}$ is a labeling of the vertices of $V(G_r)$. We will show how to choose the labeling such that p is 1-1 on S, given that G_r is acyclic. In addition, $g(y) \neq r$ if and only if y is an assigned vertex, i.e., exactly when $y \in p(S)$. This means that we get a MPHF for S as follows:

$$h(x) = \text{rank}(p(x)) \quad (2)$$

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

$$\text{rank}(u) = |\{v \in V(G_r) \ \mid \ v < u \wedge g(v) \neq r\}|. \quad (3)$$

The Ranking Step produces a data structure that allows us to compute the rank function in constant time.

Figure 1 presents a pseudo code for our family of minimal perfect hashing algorithms. If we omit the third step we will build PHFs with $m = |V(G_r)|$ instead. We now describe each step in detail.

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

Fig. 1. Main steps of the family of algorithms

3.1 Mapping Step

The Mapping Step takes the key set S as input, where $|S| = n$, and creates an acyclic random hypergraph G_r and a list of edges L. We say that a hypergraph

is *acyclic* if it is the empty graph, or if we can remove an edge with a node of degree 1 such that (inductively) the resulting graph is acyclic. This means that we can order the edges of G_r into a list $L = e_1, \ldots, e_n$ such that any edge e_i contains a vertex that is not incident to any edge e_j, for $j > i$. The list L is obtained during a test which determines whether G_r is acyclic, which runs in time $O(n)$ (see e.g. [14]).

Let Pr_a denote the probability that G_r is acyclic. We want to ensure that this is the case with constant probability, i.e., $Pr_a = \Omega(1)$. Define c by $m = cn$. For $r = 2$, we can use the techniques presented in [12] 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 [4, Theorem 6.5] also holds for our $r-$partite random hypergraphs. Their argument suggests that if $c = c(r)$ is given by

$$c(r) = \begin{cases} 2 + \varepsilon, \varepsilon > 0 & \text{for } r = 2 \\ r \left(\min_{x>0} \left\{ \frac{x}{(1-e^{-x})^{r-1}} \right\} \right)^{-1} & \text{for } r > 2, \end{cases} \tag{4}$$

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. (4). 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 [14] for general hypergraphs.

3.2 Assigning Step

The Assigning Step constructs the labeling $g : V(G_r) \rightarrow \{0, 1, \ldots, r\}$ of the vertices of G_r. To assign values to the vertices of G_r we traverse the edges in the reverse order $e_n, \ldots, e_1$ to ensure that each edge has at least one vertex that is traversed for the first time. 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 $Visited[i] = false$, $0 \leq i \leq m - 1$. Then, for each edge $e \in L$ from tail to head, we look for the first vertex u belonging to e not yet visited. Let j, $0 \leq j \leq r - 1$ be the index of u in e. Then, we set $g[u] = (j - \sum_{v \in e \wedge Visited[v] = true} g[v]) \bmod r$. Whenever we pass through a vertex u from e, if it has not yet been visited, we set $Visited[u] = true$. As each edge is handled once, the Assigning Step also runs in linear time.

3.3 Ranking Step

The Ranking Step obtains MPHFs from the PHFs presented in Section 3.2. It receives the labeling g as input and produces the 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) by using $o(m)$ additional bits of space. This is a well-studied primitive in succinct data structures (see e.g. [17]).

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.

Evaluation. To compute rank(u), where u is given by Eq. (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 that allows us to count the number of assigned vertices in $\Omega(\log m)$ bits in constant time. Such a lookup table takes $m^{\Omega(1)}$ bits of space.

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). 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 the following sections.

4 The 2-Uniform Hypergraph Instance

The use of 2-graphs allows us to generate the PHFs of Eq.(1) that give values in the range $[0, m-1]$, where $m = (2+\varepsilon)n$ for $\varepsilon > 0$ (see Section 3.1). 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. 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. (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. Then, the resulting MPHFs require $(2+\epsilon)m$ bits to be stored (remember that the ranking

information requires ϵm bits), which corresponds to $(2+\epsilon)(2+\varepsilon)n$ bits for any $\epsilon > 0$ and $\varepsilon > 0$. For $\epsilon = 0.125$ and $\varepsilon = 0.09$ the resulting functions are stored in approximately $4.44n$ bits.

4.1 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 $(\log(3)+\epsilon)(2+\varepsilon)n$ 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 $3^5 = 243 < 256$. Thus, if $\epsilon = 0.125$ and $\varepsilon = 0.09$, then the resulting functions are stored in approximately $3.6n$ bits.

We now sketch another way of improving the space to just over 3 bits per key, adding a little complication to the scheme. Use $m = (2+\epsilon/2)n$ for $\epsilon > 0$. Now store separately the set of assigned vertices, such that rank operations are efficiently supported using $(\epsilon/2)n$ bits of extra space. Finally, store for each assigned vertex v the bit $g(v)$ (must be 0 or 1). The correct bit can be found using rank on the set of assigned vertices. Thus, we need n bits to store $g(v)$. Therefore, the total space is $(3+\epsilon)n$.

5 The 3-Uniform Hypergraph 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). Therefore, we will be able to generate the PHFs of Eq. (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 2 bits to be represented. Thus, the resulting PHFs require $2(1.23+\varepsilon)n$ bits to be stored, which corresponds to $2.46n$ bits for $\varepsilon = 0$.

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

5.1 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. (1) are $\{0,1,2\}$. Then, we can apply the packing technique presented in Section 4.1 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.

6 The Full Randomness Assumption

The full randomness 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 PHFs. 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 [6]. 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 truly 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 $O(n^{1-\delta})$.

Implementation. In practice, limited randomness is often as good as total randomness [19]. For our experiments we choose h_i from a family $\mathcal{H}$ of universal hash functions proposed by Alon, Dietzfelbinger, Miltersen and Petrank [1], and we verify experimentally that the schemes behave well (see Section 7). We use a function h' from $\mathcal{H}$ so that the functions h_i are computed in parallel. For that, we impose some upper bound L on the lengths of the keys in S. The function h' has the following form: $h'(x) = Ax$, where $x \in S \subseteq \{0,1\}^L$ 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. Each hash function h_i is computed by $h_i(x) = h'(x)[a,b] \bmod (\frac{m}{r}) + i(\frac{m}{r})$, where $a = \beta i$, $b = a + \beta - 1$ and β is the number of bits used from h' for computing each h_i. In [2] it is shown a tabulation idea that can be used to efficiently implement h' and, consequently, the functions h_i. The storage space required for the hash functions h_i corresponds to the one required for h', which is $\gamma \times L$ bits.

7 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 [3] (referred to as BKZ), Fox, Chen and Heath [7] (referred to as FCH), Majewski, Wormald, Havas and Czech [14] (referred to as MWHC), and Pagh [16] (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 [3] that the BKZ algorithm outperforms it. We used the linear hash functions presented in Section 6 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 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 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 2 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 [16] (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 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. In a conversely situation where the functions do not fit in the cache, the MPHFs generated by the PAGH algorithm are the most efficient (because of lack of space we will not show this experiment).

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 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 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). 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.

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. 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 4.1 and 5.1

8 Conclusions

We have presented an efficient family of algorithms to generate near space-optimal PHPs 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 purpose algorithms found in the literature considering generation time and storage space as metrics.

Acknowledgment. We thank Djamal Belazzougui for suggesting arithmetic coding to generate more compact functions and Yoshiharu Kohayakawa for helping us with acyclic r-partite random r-graphs.

References

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. F.C. Botelho, Y. Kohayakawa, and N. Ziviani. A practical minimal perfect hashing method. In *Proc. of the 4th International Workshop on Efficient and Experimental Algorithms (WEA'05)*, pages 488–500. Springer LNCS vol. 3503, 2005.
4. Z.J. Czech, G. Havas, and B.S. Majewski. Fundamental study perfect hashing. *Theoretical Computer Science*, 182:1–143, 1997.
5. 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.
6. 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.
7. 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.
8. 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.
9. M. L. Fredman and J. Komlós. On the size of separating systems and families of perfect hashing functions. *SIAM Journal on Algebraic and Discrete Methods*, 5:61–68, 1984.
10. 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.
11. 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.
12. S. Janson. Poisson convergence and poisson processes with applications to random graphs. *Stochastic Processes and their Applications*, 26:1–30, 1987.
13. S. Lefebvre and H. Hoppe. Perfect spatial hashing. *ACM Transactions on Graphics*, 25(3):579–588, 2006.

14. 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.
15. K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, 1984.
16. 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.
17. R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2001.
18. J. Radhakrishnan. Improved bounds for covering complete uniform hypergraphs. *Information Processing Letters*, 41:203–207, 1992.
19. 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.
20. Philipp 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.