A Family of Perfect Hashing Methods

Bohdan S. Majewski* Nicholas C. Wormald† George Havas‡
Zbigniew J. Czech§

Abstract

Minimal perfect hash functions are used for memory efficient storage and fast retrieval of items from static sets. We present an infinite family of efficient and practical algorithms for generating order preserving minimal perfect hash functions. We show that almost all members of the family construct space and time optimal order preserving minimal perfect hash functions, and we identify the one with minimum constants. Members of the family generate a hash function in two steps. First a special kind of function into an $r-$graph is computed probabilistically. Then this function is refined deterministically to a minimal perfect hash function. We give strong theoretical evidence that the first step uses linear random time. The second step runs in linear deterministic time. The family not only has theoretical importance, but also offers the fastest known method for generating perfect hash functions.

Key words: Data structures, probabilistic algorithms, analysis of algorithms, hashing, random generalized graphs

1 Introduction

Consider a set S of n keys, where S is a subset of some universe $U = \{0, 1, \ldots, u-1\}$. We assume that the keys in S are either integers or strings of characters. In the latter case the keys can either be treated as numbers base $|\Sigma|$ where Σ is the alphabet in which the keys were encoded, or as sequences of characters over Σ. For convenience we assume that u is a prime.

A *hash function* is a function $h : U \to M$ that maps the set of keys S into some given interval of integers M, say $[0, m-1]$, $m > 0$. The hash function, given a key, computes an address (an integer from M) for the storage or retrieval of that item. The storage area used to store items is known as a *hash table*. A *perfect hash function* is an injection $h : S \to M$, where S and M are sets as defined above, $m \geq n$. If $m = n$, then we say that h is a *minimal perfect hash function*. If for any two keys, x_i and x_j, from S we have that $i < j$ implies $h(x_i) < h(x_j)$ then the hash function is *order preserving*.

*Department of Computer Science and Software Engineering, University of Newcastle, Callaghan, NSW 2308, Australia, **email**: bohdan@cs.newcastle.edu.au

†Department of Mathematics, University of Melbourne, Parkville, Victoria 3052, Australia, **email**: nick@maths.mu.oz.au

‡Department of Computer Science, The University of Queensland, Queensland 4072, Australia, **email**: havas@cs.uq.edu.au

§Institutes of Computer Science, Silesia University of Technology and Polish Academy of Sciences, Gliwice, Poland 44–100, **email**: zjc@star.iinf.gliwice.edu.pl

Perfect hashing offers one of the best implementations of a dictionary, a basic data structure used in many areas of computer science. Overviews of perfect hashing are given by Gonnet & Baeza-Yates (1991); Lewis & Cook (1988); Havas & Majewski (1992) and some recent independent results are presented by Fox, Heath, Chen & Daoud (1992); Fox, Chen & Heath (1992).

Various algorithms with different time complexities have been presented for constructing perfect or minimal perfect hash functions. These fall into four broad categories: (i) number theoretical methods; (ii) segmentation techniques; (iii) algorithms based on reducing the search space; and (iv) algorithms based on sparse matrix packing. The algorithms in each of the categories provide distinct solutions to the problem of finding perfect hash functions, using similar ideas but different methods to approach them.

In recent years we have witnessed vital development in the field of probabilistic methods. A number of powerful techniques have been developed, see Gupta, Bhaskar & Smolka (1994) for an overview. One method that has proved its usefulness in fields ranging from Number Theory to the Theory of Data Structures is *random search*. Often we are presented with tasks where some vast search space $\mathcal{S}$, which lacks regularity, is to be probed for an element with a particular property $\mathcal{P}$. Provided $\mathcal{P}$ is easily verified and $\mathcal{S}$ is abundant in elements satisfying $\mathcal{P}$, random search is an efficient and simple way of locating a desired element. The technique relies on picking a random member of the space and testing whether it has property $\mathcal{P}$.

Denote by p the probability that an x, randomly selected from $\mathcal{S}$, has the property $\mathcal{P}$; $p = \Pr(x \in \mathcal{S} \text{ has } \mathcal{P}) = |\mathcal{P}|/|\mathcal{S}|$. Let X be a random variable that counts the number of attempts executed before an $x \in \mathcal{S}$ having $\mathcal{P}$ is found. X has geometric distribution with

$$\Pr(X = i) = (1-p)^{i-1}p.$$

The probability distribution function of X, $F_X(i)$, is defined as

$$F_X(i) = \sum_{j=1}^{i} \Pr(X = j) = 1 - (1-p)^i.$$

The expected value of X, which is the expected number of probes executed by the random search algorithm, is

$$E(X) = \sum_{j=1}^{\infty} j \Pr(X = j) = \frac{1}{p}.$$

The probability that the random search algorithm executes no more than i iterations is

$$\Pr(X \leq i) = F_X(i).$$

For ϵ, $0 < \epsilon < 1$, the smallest i for which the random search algorithm terminates in i or less iterations with probability at least $1 - \epsilon$ is

$$Q_X(\epsilon) = \min_i \{F_X(i) \geq 1 - \epsilon\} = \left\lceil \frac{\ln \epsilon}{\ln(1-p)} \right\rceil.$$

Observe that if p is a constant then random search requires an average of $O(1)$ attempts and, with high probability (that is probability exceeding $1 - O(n^{-k})$ for some positive k), takes no more than $O(\log n)$ attempts to find an $x \in \mathcal{S}$ having $\mathcal{P}$. If $p = 1 - O(n^{-\delta})$, for $\delta > 0$, then with high probability only 1 attempt is necessary.

We present a parameterized method for generating minimal perfect hash functions, which allows arbitrary distribution of keys in the hash table. The method is probabilistic and uses random search to complete the first of two steps. It uses the concept of generalized graphs, called $r-$graphs, where an $r-$graph is a graph in which each edge is a subset of a vertex set V, containing precisely r elements, $r \geq 1$. The hash function is of the form

$$h : x \mapsto g(f_1(x)) \diamond g(f_2(x)) \diamond \cdots \diamond g(f_r(x))$$

where binary operator $\diamond$ is a mapping from $M \times M$ to M, the functions f_i map U into $V = \{0, 1, \ldots, \nu - 1\}$ for some integer $\nu > 0$, and function g maps V into M. As the generated function is to be minimal, we set $|M|$ $(= m)$ to n. For simplicity we choose $\diamond$ to be addition modulo n. Actually, any binary operator $\diamond$ that forms a group together with M may be used. For example we could choose exclusive or, giving benefits in speed and avoiding overflow for large n. We show that each member of the family, for $r \geq 2$ and a suitable choice of parameters, constructs a minimal perfect hash function for S in $O(n)$ expected time and requires $O(n \log n)$ space. (The theoretical derivation is based on a reasonable assumption about uniformity of the graphs involved in our algorithms.) An announcement of some of the results presented here is given by Havas, Majewski, Wormald & Czech (1994).

2 The family

In order to generate a minimal perfect hash function we first compute a special kind of function from the n keys into an $r-$graph with $\mu = n$ edges and ν vertices, where ν (which depends on n and r) is determined in Sections 3 and 4. The special feature is that the resulting $r-$graph must be acyclic (which we define formally below). We achieve acyclicity probabilistically. Then, deterministically, we refine this function (from the keys into an $r-$graph) to a minimal perfect hash function. The expected time for finding the hash function is $O(r\mu + \nu)$. This type of approach works for any $r > 0$. As the family of $r-$graphs for $r > 0$ is infinite, we have an infinite family of algorithms for generating minimal perfect hash functions.

Consider the following assignment problem. For a given $r-$graph $G = (V, E)$, $|E| = \mu$, $|V| = \nu$, where each $e \in E$ is an $r-$subset of V, find a function $g : V \to \{0, 1, \ldots, \mu - 1\}$ such that the function $h : E \to \{0, 1, \ldots, \mu - 1\}$ defined for $e \in E$, $e = \{v_1, v_2, \ldots, v_r\}$, as

$$h : e \mapsto \Big(g(v_1) + g(v_2) + \cdots + g(v_r)\Big) \bmod \mu$$

is a bijection. In other words we are looking for an assignment of values to vertices so that for each edge the sum of values associated with its vertices, modulo the number of edges, is a unique integer in the range $[0, \mu - 1]$.

Acyclic $r-$graphs play a prominent role in our algorithm. However, because acyclicity of $r-$graphs, especially for $r \geq 3$, may be defined in many ways (Duke, 1985) we make the notion precise by providing the explicit definition applicable to our case. We define a cycle in an $r-$graph to be a subgraph with all vertices of degree at least 2. Thus we say that an $r-$graph is acyclic if it does not contain a subgraph with minimum degree 2. An equivalent definition of acyclicity of $r-$graphs is:

Definition 2.1 *An $r-$graph is acyclic if and only if some sequence of repeated deletions of edges containing vertices of degree 1 yields a graph with no edges.*

As has been shown by Majewski (1992) and Havas & Majewski (1993), acyclicity of an $r-$graph is a sufficient (although not a necessary) condition for the assignment problem to have a solution

Unfortunately, the acyclicity test suggested directly by the definition for $r-$graphs with $r > 1$ may take $O(\mu^2)$ time. A solution that runs in optimal time has the following form (Havas, Majewski, Wormald & Czech, 1994). Initially mark all the edges of the $r-$graph as not removed. Then scan all vertices, each vertex only once. If vertex v has degree 1 then remove the edge e, such that $v \in e$, from the $r-$graph. When edge e is removed check if any other vertex in e now has degree 1. If so, then for each such vertex remove the unique edge to which the vertex belongs. Repeat this recursively until no further deletions are possible. After all vertices have been scanned, check if the $r-$graph contains edges. If so, the $r-$graph has failed the acyclicity test. If not, the $r-$graph is acyclic and the reverse order to that in which the edges were removed is one we are looking for. (A stack can be used to arrange the edges of G in an appropriate order for the second step.)

Theorem 2.2 *The recursive acyclicity test runs in $O(r\mu + \nu)$ time.*

Proof. Consider an $r-$graph $G = (V, E)$, represented by a bipartite $2-$graph $H = (V_1 \cup V_2, E')$ such that $V_1 \cap V_2 = \emptyset$, where $V_1 = V$ and $V_2 = E$ and $E' = \{\{v, e\} : v \in V, e \in E, v \in e\}$. In this representation both edges and vertices of the $r-$graph G become vertices of the bipartite graph H. An edge in the bipartite graph connects two vertices $v \in V_1$ and $e \in V_2$ if and only if v, a vertex in the $r-$graph, belongs to e, an edge, in the $r-$graph. During the acyclicity test each vertex in V_1 is tested at least once by the loop which looks for vertices of degree 1. Once some edge e (which is a vertex in V_2) is deleted, we test each other vertex of e to see if its degree is now 1. This corresponds to traveling in graph H through some edge $\{e, v\}$, where v belongs to e in G. Regardless of the result of the test, we never use that edge (of H) again. Consequently, the number of tests performed on vertices in V_1 is at most $\sum_{v \in V_1} \mathrm{dg}(v)$. As we access vertices in V_2 only once (when we delete them) and all operations mentioned here take constant time (for details see Section 3), the whole process takes at most $|V_1| + |V_2| + \sum_{v \in V_1} \mathrm{dg}(v) = \nu + \mu(r + 1)$ constant-time steps. □

To obtain a solution to the generalized assignment problem we proceed as follows. Associate with each edge a unique number $h(e) \in \{0, 1, \ldots, \mu - 1\}$ in any order. Consider the edges in reverse order to the order of deletion in an acyclicity test, and assign values to each as yet unassigned vertex in that edge. Each edge, at the time it is considered, will have one (or more) vertices unique to it, to which no value has yet been assigned. Let the set of unassigned vertices for edge e be $\{v_1, v_2, \ldots, v_j\}$. For edge e assign 0 to $g(v_2), \ldots, g(v_j)$ and set $g(v_1) = (h(e) - \sum_{i=2}^{r} g(v_i)) \bmod \mu$.

To prove the correctness of the method it is sufficient to show that the values of the function g are computed exactly once for each vertex, and for each edge we have at least one unassigned vertex by the time it is considered. This property is clearly fulfilled if G is acyclic and the edges of G are processed in the reverse order to that imposed by the acyclicity proof.

A complete algorithm comprises two steps: mapping and assignment. In the mapping step the input set is mapped into an acyclic $r-$graph $G = (V, E)$, where $V = \{0, 1, \ldots, \nu-1\}$, $E = \{\{f_1(x), f_2(x), \ldots, f_r(x)\} : x \in S\}$, and $f_i : U \to V$. An acyclic mapping is found using random search. Then the assignment step is executed. Generating a minimal perfect hash function is reduced to the assignment problem as follows. As each edge $e = \{v_1, v_2, \ldots, v_r\} \in E$ corresponds uniquely to some key x (such that $f_i(x) = v_i$, $1 \le i \le r$) we simply set,

for $e = \{f_1(x), f_2(x), \ldots, f_r(x)\}$, $h(e) = i - 1$ if x is the ith key of S, yielding the order preserving property. Then values of function g for each $v \in V$ are computed by the assignment step (which solves the assignment problem for G). The function h is an order preserving minimal perfect hash function for S.

To complete the description of the algorithm we need to define the mapping functions f_i. Let us begin with the case when keys in the input set are integers. Ideally the f_i functions should map any key $x \in S$ randomly into the range $[0, \nu - 1]$. There is no efficient algorithm for realizing a random function in the ideal sense, however limited randomness is often as good as total randomness (Carter & Wegman, 1979; Schmidt & Siegel, 1990). A suitable solution comes from the field originated by Carter & Wegman (1977) and called universal hashing. A class of universal hash functions $\mathcal{H}$ is a collection of generally good quality hash functions, from which we can easily select one at random. Carter & Wegman (1979) suggested that polynomials of fixed degree be used while Dietzfelbinger & Meyer auf der Heide (1990) proposed a class of universal hash functions which has many properties of truly random functions. Another class was suggested by Siegel (1989). The last two classes require $O(n^\epsilon)$ extra space, $0 < \epsilon < 1$. Finally Dietzfelbinger, Gil, Matias & Pippenger (1992) proved that polynomials of degree $d \geq 3$ perform well with high probability. An advantage that this class offers is a compact representation of functions, as each requires only $O(d \log u)$ bits of space. Any of these can be used for our purposes. Our experiments indicate that polynomials of degree 3 or the class defined by Dietzfelbinger & Meyer auf der Heide (1990) are the most reliable choices. For applications where speedy evaluation of the hash function is critical, polynomials of degree 1 or 2 may be used, at the risk of longer than expected generation time.

Character keys are more naturally treated as sequences of characters. Therefore we define one more class of universal hash functions, $\mathcal{C}_\nu$, designed specially for character keys. (This class has been used by others, including Fox, Heath, Chen & Daoud (1992).) We denote the length of the key x by $|x|$ and its ith character by $x_{[i]}$. A member of this class, a function $f : \Sigma^* \to \{0, 1, \ldots, \nu - 1\}$, is defined as

$$f : x \mapsto \left(\sum_{i=1}^{|x|} T\left[i, x_{[i]}\right] \right) \bmod \nu$$

where T is a table of random integers modulo ν for each character and for each position of a character in a key. Selecting a member of the class is done by selecting (at random) the mapping table T. The performance of this class is satisfactory for alphabets with more than about 8 symbols.

Class $\mathcal{C}_\nu$ allows us to treat character keys in the most natural way, as sequences of characters from a finite alphabet Σ. However, we must remember that, for any fixed maximum key length L, the total number of keys cannot exceed $\sum_{i=1}^{L} |\Sigma|^i \sim |\Sigma|^L$ keys. Thus either L cannot be treated as a constant and $L \geq \log_{|\Sigma|} n$ or, for a fixed L, there is an upper limit on the number of keys. In the former case, strictly speaking, processing a key character-by-character takes nonconstant time. Nevertheless, in practice it is often faster and more convenient to use the character-by-character approach than it is to treat a character key as a binary string. Other hashing schemes (Sager, 1985; Pearson, 1990; Fox, Heath, Chen & Daoud, 1992) use this approach, asserting that the maximum key length is bounded. This is an abuse of the RAM model (Aho, Hopcroft & Ullman, 1974, pp. 5–14), however it is a pragmatic abuse. We make this assumption, keeping in mind that it is a convenience that works in practice. It can be avoided by using for character keys our integer key approach,

giving a theoretical validation of our claims. In practice, the schemes designed specially for character keys have superior performance.

3 Implementation

In this section we explain two aspects of our implementation of the algorithm which are of interest. We outline a graph representation that allows us to carry out the acyclicity test in optimal time. We also show how to generate a random set of r vertices in such a way that no two vertices are identical. We start with the former.

We use a modification of the edge-oriented graph representation suggested by Ebert (1987). Ebert's graph representation is for directed graphs, and the direction of edges plays a significant role. The lists of edges incident with all vertices of a graph are stored in two arrays: `FIRST` and `NEXT`. The element `FIRST`$[v]$ defines the entry point to the list of edges incident with the vertex v, while `NEXT`[`FIRST`$[v]$], `NEXT`[`NEXT`[`FIRST`$[v]$]], and so on, define the subsequent edges containing v. The list ends with a special null edge, denoted by 0. Given any edge e with two endpoints v_1 and v_2, we need a mechanism that allows us to store e on both lists, without getting those lists interconnected. By introducing the notion of direction and storing e as $-|e|$ on one list and as $+|e|$ on the other we avoid the problem of duplicate entries in the two lists.

In the case of r graphs we need a more sophisticated mechanism. Using signs we have only 2 choices, but each edge e is present on r (generally greater than 2) different lists. Observing that the sign of an integer is simply one extra bit reserved in a computer word and extending that concept by allocating $\lceil \log_2(r) \rceil$ extra bits, we achieve the same effect that Ebert had for 2−graphs. We choose to reserve those extra bits at the least significant part of the word. Then, for notational convenience, we denote $e \times 2^{\lceil \log_2(r) \rceil} + i$ by $e \oplus i$.

In order to facilitate fast deletion of edges we add one more array, called `PREV`. For an edge e, `PREV`$[e \oplus i]$ (for $i = 1, \ldots, r$) stores the edge preceding edge $e \oplus i$. Thus, if `NEXT`$[e \oplus i] = b \oplus j$ then `PREV`$[b \oplus j] = e \oplus i$. For those edges e for which `NEXT`$[e \oplus i] = 0$, no entry in `PREV` is equal to $e \oplus i$. For all edges e for which `FIRST`$[v] = e \oplus i$, `PREV`$[e \oplus i] = 0$.

```
procedure insert(v_1, v_2, ..., v_r)
    G.μ := G.μ + 1;
    for i ∈ [1, 2, .., r] do
        NODE[G.μ][i] := v_i;
        if FIRST[v_i] ≠ 0 then
            PREV[FIRST[v_i]] := G.μ ⊕ i;
        end if;
        NEXT[G.μ ⊕ i] := FIRST[v_i];
        PREV[G.μ ⊕ i] := 0;
        FIRST[v_i] := G.μ ⊕ i;
    end for;
end insert;
```

Figure 1: Edge insertion

```
procedure delete(e)
    for i ∈ [1, 2, .., r] do
        if PREV[e ⊕ i] = 0 then {initial edge}
            FIRST[v_i] := NEXT[e ⊕ i];
        else                    {not initial}
            NEXT[PREV[e ⊕ i]] := NEXT[e ⊕ i];
        end if;
        if NEXT[e ⊕ i] ≠ 0 then {not at the end}
            PREV[NEXT[e ⊕ i]] := PREV[e ⊕ i];
        end if;
    end for;
end delete;
```

Figure 2: Edge deletion

For the complete representation, and also to facilitate retrieval of vertices given an edge, for each e we store its r endpoints in $\texttt{NODE}[e][1], \ldots, \texttt{NODE}[e][r]$.

In this environment, building the representation of a graph is achieved by initializing $\texttt{FIRST}[v \in V] := 0$ and then executing the procedure *insert* (Fig. 1) μ times. Testing whether a vertex v has degree 1 is done by verifying that $\texttt{FIRST}[v] \neq 0 \wedge \texttt{NEXT}[\texttt{FIRST}[v]] = 0$. Removing an edge requires $O(r)$ constant time operations, as illustrated by the procedure *delete* (Fig. 2). These and a few other simple operations allow us to implement the acyclicity test in $O(\nu + r\mu)$ time.

The second issue that we clarify is how, using exactly r calls to a random number generator, we construct r numbers $v_1, v_2, \ldots, v_r$ in the range $[0, \nu - 1]$, so that $v_i \neq v_j$ for $i \neq j$. Clearly simply generating r numbers between 0 and $\nu - 1$ may create two or more equal numbers, which is unsatisfactory, so we need a suitable alternative. We present solutions that work for $r = 2$ and $r = 3$. As indicated later, these are the two most useful cases and thus suffice for all practical purposes. An algorithm due to Floyd (Bentley, 1987) can be used for general r.

In the following we assume that the function $f_i(x)$ returns a random number between 0 and $\nu - i$, for $i = 1, 2, \ldots, r$. For $r = 2$ we use the following device to generate the endpoints of an edge:

$v_1 := f_1(x);$
$v_2 := (v_1 + f_2(x) + 1) \bmod \nu;$

This generates any pair of distinct numbers between 0 and $\nu - 1$ with equal likelihood. For $r = 3$ we use the following algorithm:

```
v1 := f1(x);
v2 := v1 + f2(x) + 1;
v3 := v1 + f3(x) + 1;
if v3 ≥ v2 then
    v3 := v3 + 1;
end if;
```

$v_2 := v_2 \bmod \nu$;
$v_3 := v_3 \bmod \nu$;

The second vertex is placed randomly anywhere except the first vertex; the third vertex is placed randomly anywhere except the first or second vertex. This solution maintains an even probability for any triple of values to be selected. As long as the functions f_i take constant time to compute, the time necessary to generate all vertices of an edge is $O(r)$.

4 Complexity analysis

In this section we give strong theoretical evidence that the expected time complexity of our algorithm is $O(r\mu+\nu)$, with $\mu = n$. For $r \geq 2$, $\nu = O(\mu)$ and thus the method stops in $O(n)$ time.

The second step of the algorithm, assignment as described above, runs in $O(r\mu+\nu)$ time. We now show that each iteration of the mapping step takes $O(r\mu+\nu)$ time, and that we can choose ν suitably so that the probability of generating an acyclic mapping tends to a nonzero constant. By the argument presented in Section 1, the expected number of iterations in such a case is $O(1)$.

In each iteration of the mapping step, the following operations are executed: (i) selection of a set of r hash functions from some class of universal hash functions; (ii) computation of values of auxiliary functions for each key in a set; (iii) testing whether the generated graph G is acyclic. Operation (i) for any of the described classes takes no more than $O(\mu+\nu)$ time. Operations (ii) and (iii) need $O(r\mu)$ and $O(r\mu+\nu)$ time, respectively. Hence, the complexity of a single iteration is $O(r\mu+\nu)$.

To obtain a nonzero probability of generating an acyclic $r-$graph we use very sparse graphs. We choose $\nu = c\mu$, for some $c > 1$. We present three results which estimate c's for every $r > 0$, such that, as μ goes to infinity the associated probability, p, is a nonzero constant. For $r \geq 3$, $p = 1$.

4.1 Case 1; 1−graphs

Theorem 4.1 *The probability of a random* $1-$*graph with* $\nu = c\mu$ *vertices and* μ *edges being acyclic is a nonzero constant if and only if* $c = \Omega(\mu)$.

Proof. This follows from the solution to the occupancy problem (Feller, 1968). To prove it in the case of limited randomness we may use Fredman, Komlós & Szemerédi (1984, Corollary 2) or Dietzfelbinger, Gil, Matias & Pippenger (1992, Fact 3.2). □

Use of 1−graphs is not efficient. It requires $O(n^2)$ space and time to build the hash function.

4.2 Case 2; 2−graphs

This case is described in detail by Czech, Havas & Majewski (1992), including an example and code for the algorithm.

Theorem 4.2 *Let G be a random graph with ν vertices and μ edges. Then if $\nu = c\mu$ holds, with $c > 2$, the probability that G is acyclic, for $\mu \to \infty$, is*

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

Proof. The probability that a random graph has no cycles, as μ tends to infinity, is $\exp(1/c + 1/c^2)\sqrt{\frac{c-2}{c}}$ (Erdős & Rényi, 1960). As our graphs may have multiple edges, but no loops, the probability that the graph generated in the mapping step is acyclic is equal to the probability that there are no cycles times the probability that there are no multiple edges. The jth edge is unique with probability $(\binom{\nu}{2} - j + 1)/\binom{\nu}{2}$. Thus the probability that all μ edges are unique is $\prod_{j=0}^{\mu-1} \left(\binom{\nu}{2} - j\right) / \binom{\nu}{2}^{\mu} \sim \exp(-1/c^2 + o(1))$ (Palmer, 1985, p. 129). Multiplying the probabilities proves the theorem.

In the case of limited randomness we may use Fredman, Komlós & Szemerédi (1984, Corollary 4) to prove that the probability of having no multiple edges tends to a nonzero constant, and differs only slightly from the result obtained for unlimited randomness. For longer cycles we must rely on the uniformity assumption. □

Thus, if $c = 2+\epsilon$ the algorithm constructs a minimal perfect hash function in $\Theta(n)$ random time. For c as small as 2.09 the probability of a random graph being acyclic exceeds $p > \frac{1}{3}$. Consequently, for such c, the expected number of iterations in the mapping step is $E(X) \leq 3$. The probability that the algorithm executes more than j iterations is $F_X(j) \leq 1 - (2/3)^j$ and with probability exceeding 0.999 the algorithm does not execute more than $Q_X(0.001) = 18$ iterations.

4.3 Case 3; $r-$graphs for $r \geq 3$

Recent theoretical work done by Pittel, Spencer & Wormald (1996) on 2−graphs allows us to identify the ratio of ν to μ for which $r-$graphs tend to be acyclic. The following claim characterizes the minimum c_r such that if $\nu > c_r\mu$ a random $r-$graph is acyclic with high probability. (We choose to give a claim and justification rather than a theorem and proof because of the length of a complete proof, witness Pittel, Spencer & Wormald (1996) which runs to about 40 pages.)

Claim 4.3 *The threshold for the appearance of a subgraph of minimum degree 2 in a random $r-$graph, $r \geq 3$, is $c_r = r/\gamma$ where*

$$\gamma = \min_{x>0} \left\{ \frac{x}{(1-e^{-x})^{r-1}} \right\}.$$

Justification. The argument given by Pittel, Spencer & Wormald (1996) is primarily devoted to graphs (that is, 2-graphs) having subgraphs of minimum degree at least k. Here is a brief summary, suitably modified so as to apply to $r-$graphs in general having subgraphs of minimum degree at least 2.

Let H be a randomly chosen $r-$graph. Define $V_0 = V(H)$ (the vertex set of H) and, for $j \geq 0$, define by induction V_{j+1} to be the set of vertices of V_j that have degree at least 2 in H_j, which we define as the restriction of H to V_j; that is, we derive H_{j+1} from H_j by peeling off those vertices of degree less than 2. This process eventually terminates when

$V_{j+1} = V_j = V_\infty$, say, for some j. H has a subgraph of minimum degree at least 2 if and only if $V_\infty \neq \emptyset$, since V_∞ will be the vertex set of the maximal subgraph of H of minimum degree at least 2. Suppose H is selected by choosing each edge independently with probability $\gamma(r-1)!/\nu^{r-1}$, where γ is some constant. Define

$$p_{i+1} = (1 - e^{-\gamma p_i})^{r-1}, \qquad q_{i+1} = (1 - e^{-\gamma p_i})^r.$$

Arguments similar to those of Pittel, Spencer and Wormald show that $|V_j| \sim q_j \nu$ almost surely as $\nu \to \infty$ with j fixed. Thus, if $\lim_{j\to\infty} q_j = 0$ then for all $\epsilon > 0$ we get $|V_j| < \epsilon\nu$ almost surely for j sufficiently large. A separate argument then shows that in this case there is almost surely no subgraph of minimum degree 2. For higher edge probabilities, we have $\lim_{j\to\infty} q_j \neq 0$, and in this case it follows that a subgraph of minimum degree 2 exists almost surely.

To see how these claims imply the stated value for the threshold, consider that in the limit $p_i = p_{i+1} = p$ where

$$p = (1 - e^{-\gamma p})^{r-1}.$$

Then

$$\gamma = \frac{\gamma p}{(1 - e^{-\gamma p})^{r-1}},$$

and so

$$\gamma = \frac{x}{(1 - e^{-x})^{r-1}}$$

for some $x > 0$. Thus the minimum positive value of $\frac{x}{(1-e^{-x})^{r-1}}$ is the minimum γ for which $\lim_{j\to\infty} q_j \neq 0$. This gives the claimed threshold. □

Since, for the critical c_r, there is almost surely no subgraph of minimum degree 2 we deduce that a random $r-$graph with μ edges and at least $c_r\mu$ vertices is acyclic with high probability. This corresponds well with the preliminary analysis and observations made by Havas, Majewski, Wormald & Czech (1994). There it was noticed that, for an increasing number of keys, the probability of generating an acyclic $r-$graph, for $r \geq 3$, approaches 1 polynomially fast in n. This distinguishes the case of $r \geq 3$ from the case when $r = 2$, as for $2-$graphs we have only a nonzero constant probability, but less than one, of completing the mapping step to yield an acyclic graph

Plots of $x(1 - e^{-x})^{1-r}$, for $r \in \{3,\ 4,\ 5\}$ are presented in Figure 3. It is possible to find the exact formula for the point $x^* > 0$ that minimizes $f(x, r) = x(1-e^{-x})^{1-r}$. By computing the first derivative of $f(x, r)$ we get

$$\frac{df(x,r)}{dx} = (1 - e^{-x})^{-r} \left[((1 - r)x - 1)e^{-x} + 1\right].$$

As the first term, $(1 - e^{-x})^{-r}$, for $x > 0$, is never 0 we may disregard it entirely. Thus we are left with the equation

$$e^x = 1 + (r - 1)x.$$

To solve it analytically we use Lambert's W function, which satisfies $W(z)e^{W(z)} = z$ (Corless, Gonnet, Hare & Jeffrey, 1993). Solving the equation gives

$$x_r^* = -W\left(\frac{-1}{r-1}e^{-1/(r-1)}\right) - \frac{1}{r-1}.$$

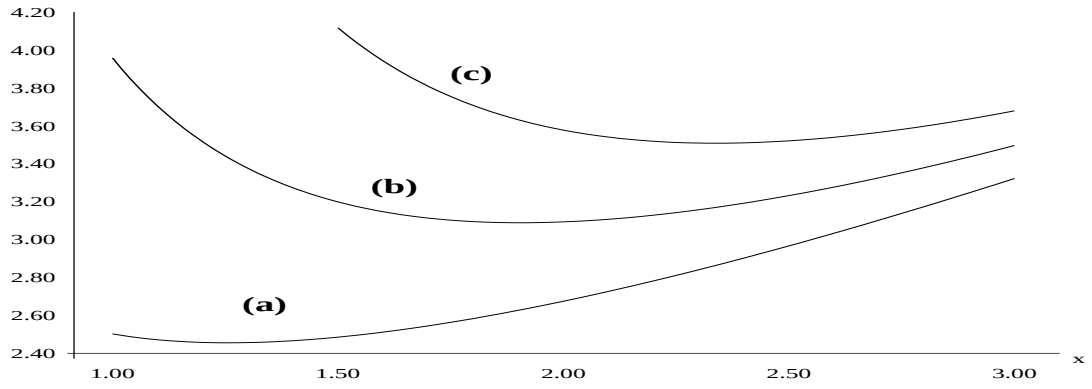

Figure 3: Plots of $x(1-e^{-x})^{1-r}$ for (a): $r=3$, (b) $r=4$ and (c) $r=5$

For real z, with $-1/e \leq z \leq 0$ (which is appropriate for $r \geq 2$), there are two real branches of $W(z)$, namely $W_0(z)$ and $W_{-1}(z)$. For the principal branch, which satisfies $-1 \leq W_0(z)$, we have $W_0(ze^z) = z$. It follows that this branch gives us $x_r^* = 0$ as a solution. The other branch, satisfying $W_{-1}(z) \leq -1$, provides us with an alternative nonzero solution. As we are not interested in the trivial (zero) solution we obtain

$$x_r^* = -W_{-1}\left(\frac{-1}{r-1}e^{-1/(r-1)}\right) - \frac{1}{r-1}.$$

For small values of r, $r=3$, 4, 5, we have the following points (all approximations obtained by Maple$^{\mathrm{TM}}$ V, Release 2):

$$\begin{aligned}
\gamma_3 &= \frac{x_3^*}{(1-e^{-x_3^*})^2} \approx 2.45541, \\
\gamma_4 &= \frac{x_4^*}{(1-e^{-x_4^*})^3} \approx 3.08912, \\
\gamma_5 &= \frac{x_5^*}{(1-e^{-x_5^*})^4} \approx 3.50890.
\end{aligned}$$

The above points give us the following thresholds: $c_3 \approx 1.22179$, $c_4 \approx 1.29487$ and $c_5 \approx 1.42495$. Notice that these values correspond very nicely with the experimental results reported by Havas, Majewski, Wormald & Czech (1994), where the found constants c_r were: $c_3 = 1.23$, $c_4 = 1.29$ and $c_5 = 1.41$. The theoretical threshold points for $r-$graphs, for $2 \leq r \leq 22$, are plotted in Figure 4.

For $r \geq 5$, using the approximation developed by Corless, Gonnet, Hare & Jeffrey (1993), we may characterize the asymptotic behavior of x_r^*, which is

$$x_r^* \simeq \ln(r) + \ln(\ln(r)) + \frac{\ln(\ln(r))}{\ln(r)} + \frac{2\ln(\ln(r)) - \ln(\ln(r))^2}{2\ln(r)^2} - \frac{1}{r-1} + O\left(\frac{1}{r^2}\right).$$

5 Discussion

The described methods represent special cases of solving a set of n linearly independent integer congruences with a larger number of unknowns. These unknowns are the entries of

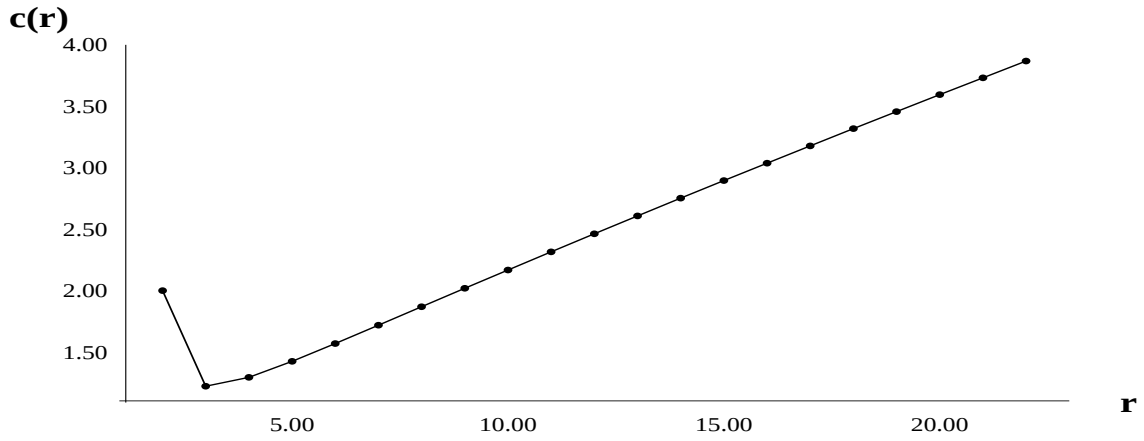

Figure 4: Threshold points for random $r-$graphs, for $r \in \{2, 3, \ldots, 22\}$

array g. We generate the set of congruences probabilistically in $O(n)$ time. We require that the congruences are consistent and that there exists a sequence of them such that 'solving' $i-1$ congruences by assignment of values to unknowns leaves at least one unassigned unknown in the ith congruence. We find the congruences in our mapping step and such a solution sequence in our independence test. It is conceivable that there are other ways to generate a suitable set of congruences, with at least n unknowns, possibly deterministically. It may be that memory requirements for such a method would be smaller than for the given method. However, any space saving can only be by a constant factor, since $\Omega(n \log n + \log \log u)$ space is required for order preserving minimal perfect hash functions, as informally proved by Fox, Chen, Daoud & Heath (1991); Havas & Majewski (1992). Further, it remains to be seen whether the solution (such values for array g that the resulting function is minimal and perfect) can then be found in linear time.

6 Conclusions

A new family of algorithms for generating minimal perfect hash functions has been developed. The time complexity of the members of the family is shown to be $O(r\mu + \nu)$. For $r > 1$ this is linear in the number of keys, which is optimal. The space complexity of constructed hash functions is $cn \log n + O(r) \log \nu$ bits (or $cn + O(r)$ words, as long as ν fits into a word). This is also optimal. For $r = 3$, minimum space is required, with $c \approx 1.23$.

For a large enough number of keys, $r = 3$ also provides the fastest member of the family. Observe that the generated hash function allows arbitrary arrangement of the keys in an input, which may be useful in some applications. The generated function is quickly computable (with $r = 2$ providing the function evaluated most quickly). The model used in theoretical considerations proved to be adequate, and we were able to give very sharp estimates. Those theoretical estimates agree very well with experimental results reported by Havas, Majewski, Wormald & Czech (1994). As indicated there, the time requirements of the new algorithm are very low, even for very large sets.

Acknowledgments

The first three authors were supported in part by the Australian Research Council.

References

Aho, A.V., Hopcroft, J.E., and Ullman, J.D. (1974) *The Design and Analysis of Computer Algorithms.* Addison-Wesley Pub. Co., Reading, Mass.

Bentley, J. (1987) Programming Pearls: A sample of brilliance. *Communications of the ACM*, **30**(9), 754–757.

Carter, J.L., and Wegman, M.N. (1977) Universal Classes of Hash Functions. *9th Annual ACM Symposium on Theory of Computing – STOC'77*, 106–112.

Carter, J.L., and Wegman, M.N. (1979) New classes and applications of hash functions. *20th Annual Symposium on Foundations of Computer Science – FOCS'79*, 175–182.

Corless, R.M., Gonnet, G.H., Hare, D.E.G., Jeffrey, D.J. (1993) On Lambert's W Function. Research Report **Cs-93-03**, Department of Computer Science, University of Waterloo, Canada.

Czech, Z.J., Havas, G., and Majewski, B.S. (1992) An Optimal Algorithm for Generating Minimal Perfect Hash Functions. *Information Processing Letters*, **43**(5), 257–264.

Dietzfelbinger, M., Gil, J., Matias, Y., and Pippenger, N. (1992) Polynomial hash functions are reliable. *19th International Colloquium on Automata, Languages and Programming – ICALP'92*, 235–246. Vienna, Austria.

Dietzfelbinger, M., and Meyer auf der Heide, F. (1990) A New Universal Class of Hash Functions, and Dynamic Hashing in Real Time. *17th International Colloquium on Automata, Languages and Programming – ICALP'90*, 6–19. Warwick University, England.

Duke, R. (1985) Types of cycles in hypergraphs. *Annals of Discrete Mathematics*, **27**, 399–418.

Ebert, J. (1987) A Versatile Data Structure for Edge-oriented Graph Algorithms. *Communications of the ACM*, **30**(6), 513–519.

Erdős, P., and Rényi, A. (1960) On The Evolution of Random Graphs. *Publ. Math. Inst. Hung. Acad. Sci.*, **5**, 17–61.

Feller, W. (1968) *An Introduction to Probability Theory and Its Applications* (3rd edition). John Wiley & Sons, Inc., New York, London, Sydney.

Fox, E.A., Chen, Q.F., Daoud, A.M., and Heath, L.S. (1991) Order Preserving Minimal Perfect Hash Functions and Information Retrieval. *ACM Transactions on Information Systems*, **9**(3), 281–308.

Fox, E.A., Chen, Q.F., and Heath, L.S. (1992) A Faster Algorithm for Constructing Minimal Perfect Hash Functions. *15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval – SIGIR'92*, 266–273. Copenhagen, Denmark.

Fox, E.A., Heath, L.S., Chen, Q.F., and Daoud, A.M. (1992) Practical Minimal Perfect Hash Functions for Large Databases. *Communications of the ACM*, **35**(1), 105–121.

Fredman, M.L., Komlós, J., and Szemerédi, E. (1984) Storing a Sparse Table with $O(1)$ Worst Case Access Time. *Journal of the ACM*, **31**(3), 538–544.

Gonnet, G.H., and Baeza-Yates, R. (1991) *Handbook of Algorithms and Data Structures*. Addison-Wesley, Reading, Mass.

Gupta, R., Bhaskar, S., and Smolka, S.A. (1994) On Randomization in Sequential and Distributed Algorithms. *ACM Computing Surveys*, **26**(1), 7–86.

Havas, G., and Majewski, B.S. (1992) Optimal Algorithms for Minimal Perfect Hashing. Research Report **234**, Department of Computer Science, The University of Queensland.

Havas, G., and Majewski, B.S. (1993) Graph Theoretic Obstacles to Perfect Hashing. *Congressus Numerantium*, **98**, 81–93.

Havas, G., Majewski, B.S., Wormald, N.C., and Czech, Z.J. (1994) Graphs, Hypergraphs and Hashing. *19th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'93)*, Lecture Notes in Computer Science, **790**, (Utrecht, 1993), 153–165. Springer Verlag, Berlin.

Lewis, T.G., and Cook, C.R. (1988) Hashing for Dynamic and Static Internal Tables. *Computer*, **21**, 45–56.

Majewski, B.S. (1992) *Minimal Perfect Hash Functions*, PhD thesis. Department of Computer Science, The University of Queensland.

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

Pearson, P.K. (1990) Fast Hashing of Variable-Length Text Strings. *Communications of the ACM*, **33**(6), 677–680.

Pittel, B., Spencer, J., and Wormald, N.C. (1996) Sudden emergence of a giant k-core in a random graph. *J. Combinatorial Theory, Series B*, **67**, 111–151.

Sager, T.J. (1985) A Polynomial Time Generator for Minimal Perfect Hash Functions. *Communications of the ACM*, **28**(5), 523–532.

Schmidt, J.P., and Siegel, A. (1990) The Analysis of Closed Hashing Under Limited Randomness. *22st Annual ACM Symposium on Theory of Computing – STOC'90*, 224–234. Baltimore, MD.

Siegel, A. (1989) On Universal Classes of Fast High Performance Hash Functions, Their Time-Space Trade-off, and Their Applications. *30th Annual Symposium on Foundations of Computer Science – FOCS'89*, 20–25.