Graphs, Hypergraphs and Hashing

George Havas * Bohdan S. Majewski† Nicholas C. Wormald‡
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 minimal perfect hash functions which allow an arbitrary order to be specified for the keys. We show that almost all members of the family are space and time optimal, and we identify the one with minimum constants. Members of the family generate a minimal perfect 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 practical and 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, generalized random graphs

1 Introduction

Consider a set W of m keys, where W is a subset of some universe $U = \{0, \ldots, u-1\}$. For simplicity we assume that the keys in W 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 $h : W \to I$ that maps the set of keys W into some given interval of integers I, say $[0, k-1]$, where $k \geq m$. The hash function, given a key, computes an address (an integer from I) for the storage or retrieval of that item. The storage area used to store items is known as a *hash table*. Keys for which the same address is computed are called *synonyms*. Due to the existence of synonyms a situation called *collision* may arise in which two keys are mapped into the same address. Several schemes for resolving

*Key Centre for Software Technology, Department of Computer Science, University of Queensland, Queensland 4072, Australia, **email**: havas@cs.uq.oz.au

†Key Centre for Software Technology, Department of Computer Science, University of Queensland, Queensland 4072, Australia, **email**: bohdan@cs.uq.oz.au

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

§Institutes of Computer Science, Silesia University of Technology and Polish Academy of Sciences, 44–100 Gliwice, Poland, **email**: zjc@gleto2.gliwice.edu.pl

collisions are known. A perfect hash function is an injection $h : W \to I$, where W and I are sets as defined above, $k \geq m$. If $k = m$, then we say that h is minimal perfect hash function. As the definition implies, a perfect hash function transforms each key of W into a unique address in the hash table. Since no collisions occur each item can be retrieved from the table in a single probe.

Minimal perfect hash functions are used for memory efficient storage and fast retrieval of items from a static set, such as reserved words in programming languages, command names in operating systems, commonly used words in natural languages, etc. An overview of perfect hashing is given in [GBY91, §3.3.16], the area is surveyed in [LC88, HM92], and some recent independent developments appear in [FCDH91, FHCD92].

Various other algorithms with different time complexities have been presented for constructing perfect or minimal perfect hash functions. They fall into four broad categories:

(i) Number theoretical methods. These solutions involve the determination of a small number of numeric parameters, using methods based on results in number theory. These include [Spr77, Jae81, Cha84, CL86, CC88, Win90b].

(ii) Perfect hash functions with segmentation. Keys are initially distributed into smaller sets, kept in buckets, by an ordinary, first-stage hash function. For all keys in a bucket a perfect hash function is computed. Algorithms falling into this category are [FKS84, SvEB84, JvEB86, SS90b, DM90, DGMP92, CHK85, DHJS83, YD85, Win90a].

(iii) Algorithms based on restricting the search space. These methods use usually some kind of backtracking procedures to search through the space of all possible functions, in order to find a perfect hash function. To limit the search space, and in effect to speed up the search, an ordering heuristic is applied to keys before the search begins. Solutions belonging to this category include [Cic80, CO82, CBK85, Sag85, HK86, BT89, GS89, FHCD92, FCH92, CM92].

(iv) Algorithms based on sparse matrix packing. The main idea behind these solutions is to map m keys uniformly into a matrix $m \times m$. Then, using a matrix packing algorithm [Meh84, p. 108–118], compress the two-dimensional array into linear space. This type of approach is adopted in [TY79], [BT90] and [CW91]. A modification of this approach, which leads to a more compact hash function, is presented in [CCJ91].

The algorithms in each of the categories provide distinct solutions, using similar ideas but different methods to approach them. Significant advances are made in [FHCD92, FCH92, CHM92, HM92] where algorithms using linear space and time are described.

We present a related family of algorithms based on generalized random graphs for finding order preserving minimal perfect hash functions of the form

$$h(w) = g(f_1(w)) \diamond g(f_2(w)) \diamond \cdots \diamond g(f_r(w))$$

where $\diamond$ is a binary operation. For simplicity we choose $\diamond$ to be addition modulo m. (Alternatively we could choose exclusive or, giving benefits in speed and avoiding overflow for large m.) We show that each member of the family, for a suitable choice of parameters, constructs a minimal perfect hash function for W in $O(m)$ expected time and requires

$O(m \log m)$ space. (The theoretical derivation is based on a reasonable assumption about uniformity of the graphs involved in our algorithms.) These algorithms are both efficient and practical.

Throughout this paper $\log(n)$ means $\log_2(n)$, $\ln(n)$ stands for $\log_e(n)$ and $y^{\underline{i}}$ denotes the falling factorial y, $y^{\underline{i}} = y(y-1)\cdots(y-i+1)$. A generalized graph, called an r–graph, is a graph in which each edge is a subset of V containing precisely r elements, $r \geq 1$.

2 Graphs and hashing

The first method which used graphs (implicitly) was published by Fredman, Komlós and Szemerédi [FKS84]. Although the authors did not cast their algorithm in graph theoretical terms it is easy to do so. Thus we can regard their first step as mapping a set of m keys into m primary vertices. For each group of keys that have been mapped into the same primary vertex, a second mapping takes each key in the group into a unique vertex. The result is an graph which is a union of star shaped trees. Fredman, Komlós and Szemerédi provide a structure that efficiently encodes the resulting graph and prove that the number of the vertices, which determines the size of the structure, does not exceed $11m$. Since the m primary vertices also need to store parameters for the second mapping, the total size of the structure is $13m$. (Fredman, Komlós and Szemerédi mention a complicated refinement that allows some reduction in the size of the encoding.)

The graph created in this scheme is a union of stars. Such graphs are not very common, which is why the number of vertices must be much greater than the number of edges (which correspond to the keys). This reduces the practicality of the method, requiring more space than desirable. Our techniques rely upon mapping the keys into any acyclic graph. This leads to two advantages: much smaller constants and the ability to generate perfect hash functions that allow arbitrary arrangement of keys in the hash table. The space requirements for our major data structures can be as little as $1.23m$.

3 The family

In order to generate a minimal perfect hash function we first compute a special kind of function from the m keys into an r–graph with m edges and n vertices, where n, depending on m and r, is determined in section 5. The special feature is that the edges of the resulting r–graph must be independent, a notion we define later. We achieve edge independence 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 $O(rm + n)$. This type of approach is suitable 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 $G = (V, E)$, $|E| = m$, $|V| = n$, where each $e \in E$ is an r–subset of V, find a function $g : V \to [0, m-1]$ such that the function $h : E \to [0, m-1]$ defined as

$$h(e = \{v_1, v_2, \ldots, v_r\} \in E) = \Big(g(v_1) + g(v_2) + \cdots + g(v_r)\Big) \bmod m$$

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 $[0, m-1]$.

This problem does not always have a solution if arbitrary graphs are considered. However, if the graph G fulfills an edge independence criterion, a simple procedure can be used to find values for each vertex. We define the edge independence criterion as follows:

Definition *Edges of an r–graph $G = (V, E)$ are independent if and only if repeated deletion of edges containing vertices of degree 1 results in a graph with no edges in it.*

Observe that the definition is equivalent to the requirement that the r–graph does not contain a subgraph with minimum degree 2. Such subgraphs, discussed in [Duk85, p. 407], are a natural generalization of cycles in 2–graphs.

To solve the assignment problem we proceed as follows. Associate with each edge a unique number $h(e) \in [0, m-1]$ in any order. Consider the edges in reverse order to the order of deletion during a test of independence, and assign values to each as yet unassigned vertex in that edge. As the definition implies, 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 $g(v_1)$, $g(v_2)$, $\ldots$, $g(v_{j-1})$ and set $g(v_j) = \left(h(e) - \sum_{v \in e, v \neq v_j} g(v)\right) \bmod m$.

To prove the correctness of the method it is sufficient to show that the value of the function g is 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 the edges of G are independent and they are processed in the reverse order to that imposed by the test for independence.

Unfortunately, the independence test suggested directly by the definition is not fast enough. For r–graphs, with $r > 1$ it is easy to find examples for which it requires $O(m^2)$ time. Consequently we must find a better method to test and arrange edges of any r–graph, for $r > 0$.

One solution has the following form. 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 (that is, belongs to only one edge) then remove the edge $e \ni v$ from the r–graph. As soon as edge e is removed check if any other of its vertices has now degree 1. If yes, then for each such a vertex remove the unique edge to which this vertex belongs. Repeat this recursively until no further deletions are possible. After all vertices has been scanned, check if the r–graph contains edges. If so, the r–graph has failed the independence test. If not, the edges are independent and the reverse order to that in which they were removed is one we are looking for. This method can be implemented so that the running time is $O(rm + n)$. A stack can be used to arrange the edges of G in an appropriate order.

The solution to this assignment problem becomes the second part of our algorithms for generating minimal perfect hash functions.

Now we are ready to present an algorithm for generating a minimal perfect hash function. The algorithm comprises two steps: mapping and assignment. In the mapping step the input set in mapped into an r–graph $G = (V, E)$, where $V = \{0, \ldots, n-1\}$, $E = \{\{f_1(w), f_2(w), \ldots, f_r(w)\} : w \in W\}$, and $f_i : U \to \{0, \ldots, n-1\}$. The step is repeated until graph G passes the test for edge independence. Once this has been achieved

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 w, such that $f_i(w) = v_i$, $1 \leq i \leq r$, the search for the desired function is straightforward. We simply set $h(e = \{f_1(w), f_2(w), \ldots f_r(w)\}) = i - 1$ if w is the ith word of W, 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 W.

To complete the description of the algorithm we need to define the mapping functions f_i. Ideally the f_i functions should map any key $w \in W$ randomly into the range $[0, n-1]$. Total randomness is not efficiently computable, however the situation is far from hopeless. Limited randomness is often as good as total randomness [CW79a, CW79b, KU86, SS89, SS90a]. A suitable solution comes from the field originated by Carter and Wegman [CW77] and called universal hashing. A class of universal hash functions $\mathcal{H}$ is a collection of generally good hash functions from which we can easily select one at random. A class is called k universal if any member of it maps k or less keys randomly and independent of each other. Carter and Wegman [CW79a] suggested polynomials to be used as hash functions. They prove that polynomials of degree d constitute a class of $(d+1)$ universal hash functions. Dietzfelbinger and Mayer auf der Heide proposed another class of universal hash functions [DM90]. Their class, based on polynomials of a constant degree d and tables of random numbers of size m^δ, $0 < \delta < 1$, is $(d+1)$ universal but shares many properties of truly random functions (see [DM90, Mey90] for details). Another class was suggested by Siegel [Sie89]. Unfortunately the last class requires a considerable amount of space. Finally, in 1992 Dietzfelbinger, Gil, Matias and Pippenger [DGMP92] proved that polynomials of degree $d \geq 3$ are reliable, meaning that they 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 the above specified classes can be used for our purposes. Our experimental results indicate that polynomials of degree 3 or the class defined by Dietzfelbinger and Mayer auf der Heide [DM90] are the best choices.

The above suggested classes perform quite well for integer keys. Character keys however are more naturally treated as sequences of characters. For that reason we define one more class of universal hash functions, $\mathcal{C}_n$, designed specially for character keys. (This class has been used by others including Fox, Heath, Chen and Daoud [FHCD92].) We denote the length of the key w by $|w|$ and its i-th character by $w[i]$. A member of this class, a function $f_i : \Sigma^* \to \{0, \ldots, n-1\}$ is defined as:

$$f_i(w) = \left(\sum_{j=1}^{|w|} T_i(j, w[j]) \right) \bmod n$$

where T_i is a table of random integers modulo n for each character and for each position of a character in a word. Selecting a member of the class is done by selecting (at random) the mapping table T_i. We can prove the following theorem:

Theorem 1 *The expected number of keys mapped independently by a member of class $\mathcal{C}_n$ is* $\left\lceil \dfrac{\log L}{\log |\Sigma| - \log(|\Sigma| - 1)} \right\rceil = O(|\Sigma| \log L)$.

Outline of proof. Consider the following painting problem. We are given an urn containing b white balls. At each step we take one ball at random, paint it in red and

return it to the urn. What is the expected number of white balls in the urn after k steps? The urn corresponds to one column of the mapping table, and $b = |\Sigma|$. Choosing a white ball represents taking an unused entry of the column, hence the key for which such an entry is selected will be mapped independently of other keys. Solving the painting problem for L independent urns proves the above theorem. □

The above defined class allows us to treat character keys in the most natural way, as sequences of characters from a finite alphabet Σ. However this approach has an unpleasant theoretical consequence. For any fixed maximum key length L, the total number of keys cannot exceed $\sum_{i=1}^{L} |\Sigma|^i = |\Sigma|(|\Sigma|^L - 1)/(|\Sigma| - 1) \sim |\Sigma|^L$ keys. Thus either L cannot be treated as a constant and $L \geq \log_{|\Sigma|} m = \Omega(\log m)$ 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 to treat a character key as a binary string. Other hashing schemes use this approach, asserting that the maximum key length is bounded (for example [Sag85, HK86, Pea90, FHCD92]). This is an abuse of the RAM model [AHU74, pp. 5–14], however it is a purely *practical* abuse. (In the RAM model with uniform cost measure it is assumed that each operation on a numeric key takes constant time, which corresponds to being able to process a character key character by character in constant time.) We make this assumption, keeping in mind that it is a convenience that works in practice. It can be avoided by use for character keys of the approach that we proposed earlier in this section. This gives a theoretical validation of the claims we make. In practice the schemes designed specially for character keys have superior performance.

4 Some benefits of arbitrary key arrangements

Unlike traditional hashing functions, our method allows the keys to be arranged in any specified order in the hash table. Furthermore, we can modify the perfect assignment problem so that for any predefined function h into the cardinal numbers, not necessarily a bijection, we are able to find a suitable function g in linear time. This can be easily extended to functions into integers, rational numbers or character strings in natural ways. It can then provide an effective method for evaluation of various kinds of discrete functions.

This property offers some advantages. One example is a dictionary for character keys. In a standard application the hash value is an index into an array of pointers that point to the beginning of the hashed strings. This means that on top of the space required for the hashing function we need m pointers. With an arbitrary order perfect hash function we can make each value point directly to the beginning of each string, saving the space required for the pointers. Another simple example is when keys form disjoint classes. Instead of storing with each key the name of a class to which it belongs, we simply assign to all keys from a given class the same hash value. Our hashing scheme also facilitates implementing a total ordering not otherwise directly computable from the data elements. These are just a few examples where hashing with arbitrary hash value selection can help.

5 Complexity analysis

In this section we give strong theoretical evidence that the expected time complexity of the algorithm is $O(rm+n)$. For $r > 1$, n is $O(m)$ and thus the method runs in $O(m)$ time for suitably chosen n.

The second step of the algorithm, assignment as described above, runs in $O(rm + n)$ time. We now show that each iteration of the mapping step takes $O(rm + n)$ time, and that we can choose n suitably so that the expected number of iterations is bounded above by a constant as m increases. For this we use the assumption that the edges in our graphs appear at random independently of each other. We call this the "uniformity assumption" since it implies that all graphs have the same probability of occurring.

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 word in a set; (iii) testing if edges of the generated graph G are independent. Operation (i) takes no more than $O(m+n)$ time (for class $\mathcal{C}_n$ the time depends on the maximum length of a word in the set W times size of alphabet Σ times r. For a particular set and predefined alphabet this may be considered to be $O(r)$). Operations (ii) and (iii) need $O(rm)$ and $O(rm + n)$ time, respectively. Hence, the complexity of a single iteration is $O(rm + n)$.

The expected number of iterations in the mapping step can be made constant by a suitable choice of n. Let p denote the probability of generating in one mapping step an r–graph with m independent edges and n vertices. Then the expected number of iterations is $\sum_{i>0} ip(1-p)^{i-1} = 1/p$.

To obtain a high probability of generating an r–graph with independent edges we use very sparse graphs. We choose $n = cm$, for some c. In the following subsections we present three theorems which estimate c's for each $r > 0$, such that as m goes to infinity the associated probability p^∞ is a nonzero constant. For $r > 2$, $p^\infty = 1$. (For detailed proofs and models see [CHM92, HM92, MWCH92]).

5.1 Case 1; 1–graphs

Theorem 2 *The probability that a random 1–graph with $n = cm$ vertices and m edges has independent edges is a non-zero constant iff $c = \Omega(m)$.*

Proof. The result follows easily from the solution to the occupancy problem (cf. [Fel68]). To prove the above result in the case of limited randomness we may use [FKS84, Corollary 2] or [DGMP92, Fact 3.2] □

The solution for 1–graphs is not acceptable, primarily because of its space requirements. It also requires $O(m^2)$ time to build the hash function.

5.2 Case 2; 2–graphs

This case is described in detail in [CHM92], including pseudocode for the algorithms.

Theorem 3 *Let G be a random graph with n vertices and m edges obtained by choosing m random edges with repetitions. Then if $n = cm$ holds with $c > 2$ the probability that G*

has independent edges, for $n \to \infty$, is

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

Proof. For 2−graphs the edge independence is equivalent to acyclicity. By the well known result of Erdős and Rényi [ER60] the probability that a random graph has no cycles, as m tends to infinity, is $\exp(1/c + 1/c^2)\sqrt{\frac{c-2}{c}}$. 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 multiple edges times the probability that there are no cycles, conditional upon there being no multiple edges. The j-th edge is unique with the probability $(\binom{n}{2} - j + 1)/\binom{n}{2}$ conditional on the earlier edges being distinct. Thus the probability that all m edges are unique is $\binom{n}{2}^{\underline{m}}\binom{n}{2}^{-m} \sim \exp(-1/c^2 + o(1))$. Multiplying the probabilities proves the theorem. □

In the case of limited randomness we may use [FKS84, Corollary 4] to prove that the probability of having no multiple edges tends to a nonzero constant, and differs only slightly form the result obtained for unlimited randomness. For longer cycles we must rely on the uniformity assumption.

As a consequence of the above theorem, if $c = 2+\epsilon$ the algorithm constructs a minimal perfect hash function in $\Theta(m)$ random time. For 2–graphs we can speed up the detection of cycles. One method is to use a set union algorithm [TVL84].

5.3 Case 3; r–graphs for $r > 2$

Edge independence is equivalent to the requirement that the r–graph does not contain a subgraph with minimum degree at least 2. For $r > 2$ the analysis is much more complicated than that for $r \leq 2$. We only show that there exists a constant, c_{inv}, such that if $m \leq c_{\text{inv}}n$ then the expected number of edge-minimal subgraphs on i edges, $E(Y_i)$, of minimum degree at least 2 tends to 0 for all $i \leq m$ as m goes to infinity. We determine the minimum possible $c \leq 1/c_{\text{inv}}$ experimentally.

Theorem 4 *For any r–graph there exists a constant c_{inv} depending only on r such that if $m \leq c_{\text{inv}}n$ the probability that a random r–graph has independent edges tends to* 1.

Outline of proof. Here also we use the uniformity assumption. To prove the above theorem we need to estimate the number of subgraphs of minimum degree at least 2 in a random $r-$graph with n vertices and m edges. This number can be shown to be

$$E(X_i) = O(m^{\underline{i}})\left(\frac{i^{r-1}r^r}{e^{r-1}n^r}\right)^i \sum_{k=r}^{ir/2}\left(\frac{\alpha^\alpha}{\rho^{1-\alpha}}\right)^{ir}\binom{n}{k}$$

where $\alpha = k/(ir)$ and $\rho = \rho(\alpha)$ is defined by $\dfrac{e^\rho - 1 - \rho}{\rho} = \alpha$. Then, using the fact that for edge-minimal subgraphs removing any edge must reduce the degree of at least one vertex in the subgraph to 1, we deduce that any such subgraph with i edges must have at least $i/2$ vertices. Now, exploiting different approximation techniques, we can show that for each r there exists a constant c_{inv} depending only on r such that if $m \leq c_{\text{inv}}n$ the expected number of edge-minimal subgraphs tends to 0. □

6 Experimental results

Four algorithms for $r \in \{2, 3, 4, 5\}$, without any specific improvements, were implemented in the C language. All experiments were carried out on a Sun SPARC station 2, running under the SunOS^{tm} operating system. For $r \in \{2, 3\}$ and character keys the results are summarized in Table 1. The values of *reps*, *mapping*, *assgn* and *total* are the average number of iterations in the mapping step, time for the mapping step, time for the assignment step and total time for the algorithm, respectively. All times are in seconds.

	$r = 2$, $c = 2.1$				$r = 3$, $c = 1.23$			
m	reps	mapping	assgn	total	reps	mapping	assgn	total
1024	2.248	0.068	0.016	0.085	2.188	0.105	0.007	0.112
2048	2.540	0.134	0.030	0.164	1.928	0.166	0.013	0.179
4096	2.536	0.246	0.056	0.302	1.664	0.271	0.027	0.298
8192	2.828	0.526	0.123	0.650	1.332	0.444	0.053	0.497
16384	2.620	0.972	0.255	1.227	1.108	0.783	0.111	0.895
24692	2.880	1.565	0.392	1.958	1.061	1.099	0.144	1.243
32768	2.660	2.109	0.529	2.638	1.010	1.584	0.236	1.820
65536	2.700	4.189	1.067	5.256	1.000	3.169	0.495	3.664
131072	2.824	8.582	2.148	10.730	1.000	6.368	1.025	7.393
262144	2.868	18.022	4.620	22.642	1.000	12.176	2.086	14.262
524288	2.756	33.448	8.563	42.011	1.000	24.855	4.201	29.056

Table 1: Experimental results

For integer keys and $r = 3$ the results are presented in Table 2. The mapping functions f_i, $1 \leq i \leq 3$ were selected from class $\mathcal{H}_n^3$ and keys were chosen from the universe $U = \{0, \ldots, 2^{31} - 2\}$. Each row in the table represents the average taken over 200 experiments. Notice that computing three polynomials of degree 3 takes about twice as much as evaluating three functions from class $\mathcal{C}_n$. Similarly, as for character keys, for $m > 64000$ the number of iterations stabilized at 1. Hence the average is also the worst case behavior for sufficiently large m.

For $r \in \{3, 4, 5\}$ we experimentally determined constants c_r, such that if $n = c_r m$ then the expected number of iterations in the mapping step is a nonincreasing function of m. These values are: $c_3 = 1.23$, $c_4 = 1.29$, $c_5 = 1.41$. For these constants and for increasing m, the observed average number of iterations in the mapping step approached 1. Thus $r = 3$ outperforms $r = 2$ as the number of keys goes up, because the expected number of iterations for the mapping step for $r = 3$ goes down, while the expected number is constant for $r = 2$.

The experimental results fully back the theoretical considerations. Also, the time requirements of the new algorithm are very low. Likewise the mapping, assignment and total times grow approximately linearly with m.

7 Discussion

The method presented is a special case of solving a set of m linearly independent integer congruences with a larger number of unknowns. These unknowns are the entries of array

$r = 3$, $c_3 = 1.23$				
m	reps	mapping	assgn	total
1000	2.290	0.222	0.007	0.229
2000	1.655	0.328	0.014	0.343
4000	1.325	0.534	0.026	0.560
8000	1.215	0.978	0.053	1.031
16000	1.100	1.799	0.109	1.908
32000	1.010	3.357	0.231	3.588
64000	1.005	6.693	0.486	7.179
128000	1.000	13.277	1.003	14.280
256000	1.000	26.447	2.034	28.481
512000	1.000	52.676	4.102	56.778

Table 2: Experimental results for integer keys

g. We generate the set of congruences probabilistically in $O(m)$ 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 m 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 $O(m \log m)$ space is required for order preserving minimal perfect hash functions (see [HM92]). Further, it remains to be seen if the solution (such values for array g that the resulting function is minimal and perfect) can then be found in linear time.

Acknowledgements

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

References

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Pub. Co., Reading, Mass., 1974.

[BT89] M.D. Brain and A.L. Tharp. Near-perfect hashing of large word sets. *Software — Practice and Experience*, 19:967–978, 1989.

[BT90] M.D. Brain and A.L. Tharp. Perfect hashing using sparse matrix packing. *Information Systems*, 15(3):281–290, 1990.

[CBK85] N. Cercone, J. Boates, and M. Krause. An interactive system for finding perfect hash functions. *IEEE Software*, 2(6):38–53, 1985.

[CC88] C.C. Chang and C.H. Chang. An ordered minimal perfect hashing scheme with single parameter. *Information Processing Letters*, 27(2):79–83, February 1988.

[CCJ91] C.C. Chang, C.Y. Chen, and J.K. Jan. On the design of a machine independent perfect hashing. *The Computer Journal*, 34(5):469–474, 1991.

[Cha84] C.C. Chang. The study of an ordered minimal perfect hashing scheme. *Communications of the ACM*, 27(4):384–387, April 1984.

[CHK85] G.V. Cormack, R.N.S. Horspool, and M. Kaiserwerth. Practical perfect hashing. *The Computer Journal*, 28(1):54–55, February 1985.

[CHM92] 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, October 1992.

[Cic80] R.J. Cichelli. Minimal perfect hash functions made simple. *Communications of the ACM*, 23(1):17–19, January 1980.

[CL86] C.C. Chang and R.C.T. Lee. A letter-oriented minimal perfect hashing scheme. *The Computer Journal*, 29(3):277–281, June 1986.

[CM92] Z.J. Czech and B.S. Majewski. Generating a minimal perfect hashing function in $O(m^2)$ time. *Archiwum Informatyki*, 4:3–20, 1992.

[CO82] C.R. Cook and R.R. Oldehoeft. A letter oriented minimal perfect hashing function. *SIGPLAN Notices*, 17(9):18–27, September 1982.

[CW77] J.L. Carter and M.N. Wegman. Universal classes of hash functions. In *9th Annual ACM Symposium on Theory of Computing – STOC'77*, pages 106–112, 1977.

[CW79a] J.L. Carter and M.N. Wegman. New classes and applications of hash functions. In *20th Annual Symposium on Foundations of Computer Science — FOCS'79*, pages 175–182, 1979.

[CW79b] J.L. Carter and M.N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[CW91] C-C. Chang and T-C. Wu. Letter oriented perfect hashing scheme based upon sparse table compression. *Software — Practice and Experience*, 21(1):35–49, January 1991.

[DGMP92] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In *19th International Colloquium on Automata, Languages and Programming – ICALP'92*, pages 235–246, Vienna, Austria, July 1992. LNCS 623.

[DHJS83] M.W. Du, T.M. Hsieh, K.F. Jea, and D.W Shieh. The study of a new perfect hash scheme. *IEEE Transactions on Software Engineering*, SE–9(3):305–313, March 1983.

[DM90] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions, and dynamic hashing in real time. In *17th International Colloquium on Automata, Languages and Programming – ICALP'90*, pages 6–19, Warwick University, England, July 1990. LNCS 443.

[Duk85] R. Duke. Types of cycles in hypergraphs. *Ann. Discrete Math.*, 27:399–418, 1985.

[ER60] P. Erdős and A. Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci.*, 5:17–61, 1960. Reprinted in J.H. Spencer, editor, *The Art of Counting: Selected Writings*, Mathematicians of Our Time, pages 574–617. Cambridge, Mass.: MIT Press, 1973.

[FCDH91] E. Fox, Q.F. Chen, A. Daoud, and L. Heath. Order preserving minimal perfect hash functions and information retrieval. *ACM Transactions on Information Systems*, 9(2):281–308, July 1991.

[FCH92] E. Fox, Q.F. Chen, and L. Heath. LEND and faster algorithms for constructing minimal perfect hash functions. Technical Report TR-92-2, Virginia Polytechnic Institute and State University, February 1992.

[Fel68] W. Feller. *An Introduction to Probability Theory and Its Applications*. John Wiley & Sons, Inc., New York, London, Sydney, third edition, 1968.

[FHCD92] 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, January 1992.

[FKS84] 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.

[GBY91] G.H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, Reading, Mass., 1991.

[GS89] M. Gori and G. Soda. An algebraic approach to Cichelli's perfect hashing. *BIT*, 29(1):209–214, 1989.

[HK86] G. Haggard and K. Karplus. Finding minimal perfect hash functions. *ACM SIGCSE Bulletin*, 18:191–193, 1986.

[HM92] G. Havas and B.S. Majewski. Optimal algorithms for minimal perfect hashing. Technical Report 234, The University of Queensland, Key Centre for Software Technology, Queensland, July 1992.

[Jae81] G. Jaeschke. Reciprocal hashing: A method for generating minimal perfect hashing functions. *Communications of the ACM*, 24(12):829–833, December 1981.

[JvEB86] C.T.M. Jackobs and P. van Emde Boas. Two results on tables. *Information Processing Letters*, 22:43–48, 1986.

[KU86] A. Karlin and E. Upfal. Parallel hashing - an efficient implementation of shared memory. In *18th Annual ACM Symposium on Theory of Computing – STOC'86*, pages 160–168, May 1986.

[LC88] T.G. Lewis and C.R. Cook. Hashing for dynamic and static internal tables. *Computer*, 21:45–56, 1988.

[Meh84] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1. Springer-Verlag, Berlin Heidelberg, New York, Tokyo, 1984.

[Mey90] F. Meyer auf der Heide. Dynamic hashing strategies. In *15th Symposium on Mathematical Foundations of Computer Science – MFCS'90*, pages 76–87, Banska Bystrica, Czechoslovakia, August 1990.

[MWCH92] B.S. Majewski, N.C. Wormald, Z.J. Czech, and G. Havas. A family of generators of minimal perfect hash functions. Technical Report 16, DIMACS, Rutgers University, New Jersey, USA, April 1992.

[Pea90] P.K. Pearson. Fast hashing of variable-length text strings. *Communications of the ACM*, 33(6):677–680, June 1990.

[Sag85] T.J. Sager. A polynomial time generator for minimal perfect hash functions. *Communications of the ACM*, 28(5):523–532, May 1985.

[Sie89] A. Siegel. On universal classes of fast high performance hash functions, their time-space trade-off, and their applications. In *30th Annual Symposium on Foundations of Computer Science – FOCS'89*, pages 20–25, October 1989.

[Spr77] R. Sprugnoli. Perfect hashing functions: A single probe retrieving method for static sets. *Communications of the ACM*, 20(11):841–850, November 1977.

[SS89] J.P. Schmidt and A. Siegel. On aspects of universality and performance for closed hashing. In *21st Annual ACM Symposium on Theory of Computing – STOC'89*, pages 355–366, Seattle, Washington, May 1989.

[SS90a] J.P. Schmidt and A. Siegel. The analysis of closed hashing under limited randomness. In *22st Annual ACM Symposium on Theory of Computing – STOC'90*, pages 224–234, Baltimore, MD, May 1990.

[SS90b] 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.

[SvEB84] C. Slot and P. van Emde Boas. On tape versus core: An application of space efficient hash functions to the invariance of space. In *16th Annual ACM Symposium on Theory of Computing – STOC'84*, pages 391–400, Washington, DC, May 1984.

[TVL84] R.E. Tarjan and J. Van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, April 1984.

[TY79] R.E. Tarjan and A.C-C Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, November 1979.

[Win90a] V.G. Winters. Minimal perfect hashing for large sets of data. In *International Conference on Computing and Information – ICCI'90*, pages 275–284, Niagara Falls, Canada, May 1990.

[Win90b] V.G. Winters. Minimal perfect hashing in polynomial time. *BIT*, 30(2):235–244, 1990.

[YD85] W.P. Yang and M.W. Du. A backtracking method for constructing perfect hash functions from a set of mapping functions. *BIT*, 25(1):148–164, 1985.