Optimal algorithms for minimal perfect hashing

George Havas and Bohdan S. Majewski
Key Centre for Software Technology
Department of Computer Science
University of Queensland
Queensland 4072
Australia

Abstract

Minimal perfect hash functions are used for memory efficient storage and fast retrieval of items from static sets. We present an overview of previous solutions and analyze a new algorithm based on random graphs for generating order preserving minimal perfect hash functions. We show that the new algorithm is both time and space optimal in addition to being practical. The algorithm generates a minimal perfect hash function in two steps. First a special kind of function into a random graph is computed probabilistically. Then this function is refined deterministically to a minimal perfect hash function. The first step uses linear random time, while the second runs in linear deterministic time.

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

1 Introduction

Consider a set W of m words each of which is a finite string of symbols over an ordered alphabet Σ. A *hash function* is a function $h : W \to I$ that maps the set of words W into some given interval of integers I, say $[0, k-1]$, where k is an integer, and usually $k \geq m$. The hash function, given a word, 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*. Words 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 items w_1 and w_2 have the same address. Several schemes for resolving 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 a minimal perfect hash function. As the definition implies, a perfect hash function transforms each word 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. A hash function is *order preserving* if it puts entries into the hash table in a prespecified order.

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. Aspects of perfect hashing are discussed in detail in [Meh84, §III.2.3], [GBY91, §3.3.16], [LC88] and [FHCD92]. It is clear that generating a perfect hash function takes at least linear time in the number of keys (all keys must be read). The generated program is space optimal if it can be stored in $O(m + \log\log u)$ bits (see [Meh82, Meh84], [Mai83] or [GBY91]), where u is the size of universe U from which keys in W have been selected. If the function is also to be order preserving, extra space is required: $O(m \log m + \log\log u)$ bits are necessary and sufficient. This result can be derived by observing that only 1 out of $m!$ mappings is order preserving. Since at most $(u/k)^m\binom{k}{m}$ subsets are mapped perfectly [Meh82, Meh84] therefore a perfect hash function is order preserving for just $(u/k)^m\binom{k}{m}/m!$ subsets of universe U. Consequently the number of distinct order preserving perfect hash functions must be at least $H \geq m!\binom{u}{m}/((u/k)^m\binom{k}{m})$, and the length of a program that evaluates any order preserving perfect hash function is $\Omega(\log|H| + \log\log u) = \Omega(m\log m + \log\log u)$, where the $\log\log u$ term is an initial cost due to the size of the universe (cf. [Meh84, p. 129]). The upper bound for the size of a program can be proved by explicitly constructing an order preserving perfect hash function of size $O(m\log m + \log\log u)$. This has been done in [Cha84, CC88] and in this paper (a similar bound is given in [FCDH91] using a different argument). The generated program is time optimal if the evaluation of the perfect hash function takes $O(1)$ time. Fast, space efficient algorithms for generating minimal perfect hash functions are desired. The resulting hash functions should also be fast and space efficient.

Various algorithms with different time complexities have been presented for constructing perfect or minimal perfect hash functions. Early methods had exponential time complexity, after which algorithms with apparently polynomial time performance were developed. We analyze an algorithm based on random graphs for finding order preserving minimal perfect hash functions of the form:

$$h(w) = \Big(g(f_1(w)) + g(f_2(w))\Big) \bmod m$$

where f_1 and f_2 are functions that map keys into integers, and g is a function that maps integers into $[0, m-1]$. We show that the expected time required by the generation algorithm is $O(m)$, which is optimal. The generated program evaluates the hash function in constant time, also optimal. It requires $O(m\log m + \log\log u)$ bits space for both generation and evaluation, which is optimal for an order preserving function (for character keys, by virtue of the mapping used, the $\log\log u$ term is bounded by $\log m$ and thus can be omitted). In some ways, the $\log m$ component of the space requirements is little more than a theoretical measure. If we assume 32 bit unsigned integers for storage, then the $O(m\log m)$ space requirements can be readily implemented using about $2m\lceil\log_{2^{32}} m\rceil$ words. This means that the $\log m$ component has no effect till m exceeds a trillion, so is unlikely to have any impact in practice.

Fox *et al.* concentrate on minimizing the space required for generating minimal perfect hashing functions. In a recent paper Fox, Chen and Heath [FCH92] present an algorithm capable of generating a minimal perfect hash function which requires about 2.7 bits per key and exhibits apparently linear time. However no mathematical proof of the time complexity is provided. We emphasize fast algorithms with proofs of linear expected time. Our algorithm can generate an order preserving minimal perfect hash function for one million keys (character strings of length 18) in about 82 seconds on a Sun Sparc station 2, using about $2.1m$ words space. An initial description of our method has appeared in [CHM92] and a generalization is in [MWCH92].

2 Previous solutions

2.1 The searching problem

Hashing belongs to a wide class of searching algorithms. Straightforward solutions to the searching problem are linear and binary search. In linear search, keys are stored in a sequence. To locate a key we compare it with stored keys until we find the matching key or all m keys have been examined. Any key can be located in $O(m)$ time. This is generally regarded as being unacceptable. An obvious improvement is to sort the keys into an ordered sequence and use a binary search. This reduces searching time to $O(\log m)$ steps. Other access methods are offered by various kinds of tree structures (for an overview see [GBY91, Chapter 3]). For example digital trees or tries are recursive tree structures which use decomposition of the key to direct the branching. They allow almost constant access time (actually fast logarithmic) at the expense of significant memory space. These straightforward solutions outperform many previously published methods based on perfect hashing.

2.2 Number theoretical solutions

Various solutions based on "simple" numeric computations have been found. These solutions involve the determination of a small number of numeric parameters (sometimes just one) which are incorporated into a formula involving the keys to determine the hash values. In many of these methods, keys are assumed to be integers, and no conversion from character keys to integers is given. (This is relatively straightforward if large integers are acceptable, in which case character strings can be treated as numbers using a base equal to the size of the character set. However, large integers may increase the complexity of the associated calculations.)

Thus, Sprugnoli [Spr77] presented two algorithms for generating perfect (but not minimal) hash functions. His first method finds the smallest hash table for the given hash function, but can produce very sparse tables and has exponential run time. His second method is faster, but it is not guaranteed to work for all sets. Some remarks on implementation may be found in [AA79]. Sprugnoli suggested that his methods may be applied for small sets with, say, 10 to 12 elements. For larger sets he suggested segmentation, discussed in the next section.

Jaeschke [Jae81] proposed a method for creating minimal perfect hash functions and proved it always works. Unfortunately, the algorithm for finding its major parameter has exponential time complexity and is impractical for sets with more than about 20 elements. Also, the parameter can be as large as $O(2^m)$. Chang [Cha84] developed a scheme for generating minimal perfect hash functions, based on the Chinese remainder theorem. Unfortunately it requires a prime number function for the keys, and no general such function is known. Chang implemented an algorithm for finding the major parameter in $O(m^2 \log m)$ time. This is polynomial time, in contrast to the exponential time required by Jaeschke's algorithm. However it is polynomial time in terms of the number of arithmetic operations, but the number of bits to represent the parameter is $O(m \log m)$. Even for quite small sets the parameter can be very large. For example, for the set of the first 64 primes it is approximately 1.92×10^{124}, requiring 413 bits in binary representation.

Chang and Lee [CL86] modified Chang's approach to make it directly applicable to character keys. However, they made an assumption that each key can be uniquely identified by a pair of characters, which is false for some small, ordinary sets. They provided an algorithm which computes associated parameter values and proved that the resulting function is indeed a minimal perfect hash function. They reported success with several nontrivial (but relatively small) sets of words. As in the previous algorithm, parameter values can be large.

Trying to overcome drawbacks of Chang's algorithm, Chang and Chang [CC88] proposed a new minimal perfect hashing scheme. The method removes the first impediment of Chang's algorithm, where either the keys have to be relatively prime or a prime number function is required for W. Also the method for parameter computation is quite straightforward, and Chang and Chang show that the computation time required is $O(m \log(\max_i w_i))$ arithmetic operations. Unfortunately, again, the parameter value tends to be very large, and the number of bits required to store it is $O(m \log m)$. Evaluation of the hash function takes roughly $O(m \log(\max_i w_i))$ time. Since $\max_i w_i$ must be at least m and, generally, is much greater, this is a factor of m times worse than binary search on an ordered array and logarithmically worse than linear search on an unsorted sequence. Another variation of Chang's solution was proposed by Winters [Win90b]. He used the fact that in order to be able to use Chang's scheme [Cha84] integer keys do not need to be prime, merely relatively prime and presented a method which transforms an arbitrary set of nonnegative integers into a set of nonnegative and relatively prime integers. In addition, he proposed a probabilistic method to convert an input set $\{w_1, w_2, \ldots, w_m\}$ into a set $\{x_1, x_2, \ldots, x_m\}$, such that $\max_i x_i = O(m^2)$. This conversion is useful if the values of elements are large, compared with their number (for example, if the original keys are character strings treated as numbers using the natural base). Also, in the transformation of an input set into a set of relatively prime integers, differences of elements are used, so the conversion allows a reduction in the size of the parameters of the hash function.

Unfortunately, the major parameter produced by the algorithm is enormous. In a section about complexity analysis of the algorithm, Winters proved that the number of bits required to represent it is $O(m^3 \log m)$ and the time necessary to compute it is roughly $O(m^4 \log m)$. Although the algorithm is guaranteed to stop after a polynomially bounded number of arithmetic steps, the magnitude of the parameter makes the solution absolutely impractical. The time required to evaluate the hash function for one key is approximately $m^2 \log m$ times worse than the time required for a simple linear search through an unordered sequence.

To reduce the magnitude of the parameter Winters [Win90a] suggested a new scheme based on a quotient/remainder splitting of the input set of keys. In the first phase of his method the magnitudes of the keys are reduced so that each key is bounded by m^2. Then the input set is split into a number of subsets until each subset size is less than 3 (or any other predefined constant). For these subsets minimal perfect hash functions are generated. The author proves in [Win90a] that the structure requires $O(m \log m \log\log m)$ bits, and can be constructed in $O(m^2 \log\log m)$ time. The evaluation of such a constructed hash function takes $O(\log\log m) + O(1)$ operations. The method certainly improves on the technique presented in [Win90b], but its space and time requirements make it rather unattractive.

2.3 Perfect hash functions with segmentation

Sprugnoli [Spr77], for sets larger than about 12 elements, suggested the distribution of keys into smaller sets be done by using an ordinary, first-stage hash function which hashes keys into buckets. For all keys in a bucket, a perfect hash function is generated and its description is stored in the bucket. With this approach, Sprugnoli was able to generate a minimal perfect hash function for the 31 most common English words.

Segmentation requires two probes, one to determine the bucket and obtain the information for the second hash function for that bucket, and a second to access the table location containing the key. But it offers an advantage: the perfect hash function is constructed for small subsets, thus significantly improving the chance of success.

Fredman, Komlós and Szemerédi [FKS84] described another segmentation based method for generation of a perfect hash function for integer keys. They assumed that the input set W is a subset of a universe U, $U = \{1, \ldots, u-1\}$ where u is a prime. By careful choice of parameters they showed how to generate a perfect hash function of size $13m$ in expected linear time. They also showed how to generate a function of size $6m$ in $O(m^3 \log u)$ deterministic time, improved to $O(m^3 + m \log m \log\log u)$ random time by Mehlhorn [Meh84]. Jackobs and van Emde Boas [JvEB86] reduced the space requirement of the FKS scheme to $O(\log\log u + m \log\log m)$-bits, while maintaining $O(1)$ access time. Slot and van Emde Boas [SvEB84] showed that a variation of the FKS-scheme can be made space optimal, i.e. stored in $O(\log\log u + m)$ space, at a cost of $O(m)$ evaluation time. Finally, Sis and Siegel [SS90b] using a nontrivial coding showed how to implement the FKS scheme in optimal space, with $O(1)$ evaluation time. The last three methods are for the most part of theoretical importance only, as they are hard to implement and constants associated with the evaluation of the hash functions are prohibitive. A similar scheme, which allows both insertion and deletion of keys proposed by Cormack, Horspool and Kaiserswerth [CHK85] uses $m + O(1)$ space, but may require $O\left(\frac{m^m}{(m-1)!}\right)$ time to construct a perfect hash function.

Another method is due to Du, Hsieh, Jea and Shieh [DHJS83]. It is based on rehashing and segmentation. (However Lewis and Cook [LC88] indicate it has four serious drawbacks.) Yang and Du [YD85] modified the approach by introducing a backtrack search to enhance the success rate. They indicated substantial improvement in performance over the previous solution. However, for sets with more than 50 keys the probability of success even for this method tends rapidly to 0.

2.4 Algorithms based on restricting the search space

Cichelli [Cic80] presented a simple heuristic method for generating hash functions for sets of character keys. His hash function is $h(w) = \text{length}(w) + g(w[1]) + g(w[\text{length}(w)])$, where function g is a parameter of the hash function. Building the hash table requires determining a function g to make $h(w)$ a minimal perfect hash function. This is done by exhaustive search with backtracking, which is sped up by considering the keys in such an order that backtracking is avoided as much as possible.

Cichelli's method is simple and efficient. Unfortunately, mainly because of the exponential time for the searching step, the algorithm works well only when the set of words is fairly small. A Monte Carlo study of Cichelli's method [BF83] showed that the probability of generating a minimal perfect hash function tends quickly to 0 for $m > 30$. Several attempts have been made to overcome these drawbacks, including: Cook and Oldehoeft [CO82]; Cercone, Boates and Krause [CBK85]; Haggard and Karplus [HK86]; Brian and Tharp [BT89]; Gori and Soda [GS89]. However all of these work well only for relatively small sets.

Sager [Sag85], in an optimization of Cichelli's method, presented the *mincycle* algorithm for generating minimal perfect hash functions. Sager introduced better mapping functions and proposed a quite effective ordering heuristic which, for certain parameters of the hash function, may guarantee polynomial average time behavior of the searching step. The pseudo-random mapping functions "seem to work well" for sets of up to several hundred keys. However experiments show that for larger sets they are impractical, as they fail to map distinct keys differently. Sager did not give a general scheme for choosing mapping functions if those provided fail. His ordering heuristic uses a dependency graph, G_d, to represent dependencies between hash values of keys. With a graph theoretical approach, the ordering step tries to choose a sequence of the keys which minimizes the amount of backtracking in the searching step. The ordering heuristic is not necessarily optimal, but performs reasonably well in most cases.

Based on experimental results, Sager claimed his algorithm runs in $O(m^4)$ time. He gave no proof of this claim but mentioned a formal proof which shows that, for certain parameters for which the running time of the algorithm is $O(m^6)$, the ordering step can always be expected to dominate the searching step. The mincycle algorithm certainly improves on Cichelli's method, but experimental evidence shows that it is impractical for sets with more than about 500 words. This limitation is addressed in subsequent improvements whose structure is similar to the mincycle algorithm, with enhancements to each step making them applicable for much larger sets.

Fox, Heath and Chen [FHC89] and Fox, Heath, Chen and Daoud [FHCD92] made effective use of randomness and introduced more complicated mapping functions which are guaranteed to work. They observed that the average degree of the vertices in the dependency graph is low and suggested a faster technique for selecting words by choosing vertices according to degree. Their ordering technique is much faster than that used by Sager but the quality of ordering, measured with respect to the amount of backtracking executed in the searching step, is worse. The backtracking is then effectively reduced by introducing a randomness in the exhaustive search.

The searching step of the algorithm differs slightly from that proposed by Sager, but from an algorithmic point of view it has similar characteristics. Fox, Heath and Chen generated minimal perfect hash functions for sets with up to 420000 words. (Their algorithm took 1624 seconds on a Sequent Symmetry for 420878 words.) Also based on experimental results, they asserted that the searching step requires $O(m)$ time on average to complete its task. As the ordering step needs $O(m \log m)$ time, they claimed that their algorithm runs in $O(m \log m)$ expected time. In [FHCD92], because of a small factor associated with the $m \log m$ term in the ordering step, they made a claim that the algorithm runs in "practically" $O(m)$ time. Nevertheless, in a strict mathematical sense the ordering step has $O(m \log m)$ time complexity. There is no strong theoretical evidence that the searching step has the claimed expected linear time complexity. Moreover there are classes of dependency graphs for which the algorithm will fail, although the probability of generating such a graph is very small.

Independently, Czech and Majewski [CM92] proposed a different modification of the mincycle algorithm. They observed that the output of the ordering step may be viewed as a spanning tree T of G_d. The spanning tree is built so that the total length of fundamental cycles $L(T)$ is minimized, so they have to search for a spanning tree that generates fundamental cycles with minimum length. Unfortunately, this problem is known to be $\mathcal{NP}$-complete [DPK82], but Deo, Prabhu and Krishnamoorthy discussed some approximate solutions. Based on those ideas, Czech and Majewski developed a heuristic method which gives a lower $L(T)$ value for G_d. After a spanning tree is built, an ordering procedure is executed. At each step it selects a word for which the corresponding edge lies on a maximal number of shortest fundamental cycles of G_d with respect to T. This technique, much faster than the one introduced by Sager, finds an ordering of the keys with virtually the same quality as Sager's. Czech and Majewski also modified the searching step by introducing backtrack pruning. The time complexity of the ordering step is $O(m^2)$ and can be reduced to $O\left(\frac{L(T)^2}{m} \log m\right)$. As the ordering step always dominated the searching step, they claimed that its complexity determines the time complexity of the algorithm.

This method, like Fox, Chen and Heath's solution, reduces the time complexity of the mincycle algorithm. Czech and Majewski generated minimal perfect hash functions for sets with up to 32768 words, in times which compared favorably.

However, in this case too, there is no mathematical proof given that the expected time complexity of the searching step is bounded by a polynomial function of m.

2.5 Algorithms based on sparse matrix packing

Recently two polynomial time algorithms based on sparse matrix packing have been proposed. The main idea is that, once we have two selector functions $\alpha(w)$ and $\omega(w)$ into integers such that the ordered pair $(\alpha(w), \omega(w))$ is unique for each key, a very simple method can be used to construct a minimal perfect hash function. The pair $(\alpha(w), \omega(w))$ is used to address entries in two-dimensional array. As each pair points to a unique location, it is enough to store at this location the value of the hash function for the associated key. Such a hash function may require $O(m^2)$ space. To reduce the space requirements, a sparse matrix packing algorithm is used (compare [Meh84, p. 108–118]). For a matrix with $O(m)$ elements and $O(m^2)$ size there exist several packing techniques which can compress it into linear space in $O(m^2)$ time.

Brain and Tharp [BT90] designed such an algorithm for generating a perfect hash function with specific selector functions. They reported success with 5000 words chosen from the standard Unix$^{\text{TM}}$ dictionary. Another algorithm based on sparse matrix packing was described by Chang and Wu [CW91]. Basically their method differs from that of Brain and Tharp in the packing procedure. This method may lead to slight improvement in packing, but requires a more complicated hash function. As in the first method, the time complexity is $O(m^2)$.

3 Optimal solutions

In order to generate a minimal perfect hash function we first compute a special kind of function from the m keys into a graph with $n = O(m)$ vertices and m edges. The special feature is that the resulting graph must be acyclic, a property we achieve probabilistically. Then we refine this function (deterministically) to a minimal perfect hash function. Note that this replaces the three step (Mapping, Ordering, Searching) approach used in other algorithms by a two step process, which is simpler to implement and to analyze. A version of the algorithm is presented in [CHM92], in more detail.

The expected time for finding the hash function is linear in the number of keys. The new algorithm, based on random graphs, finds minimal perfect hash functions of the form:

$$h(w) = \Big(g(f_1(w)) + g(f_2(w))\Big) \bmod m$$

where $f_i : W \to \{0, \ldots, n-1\}$ and $g : \{0, \ldots, n-1\} \to \{0, \ldots, m-1\}$. The f_i functions are auxiliary hash functions selected from a class of universal hash functions. Function g is implemented as a table lookup. Consequently the evaluation time depends on how quickly the auxiliary functions can be computed. We will see that it can be done in fast constant time.

Consider the following problem. For a given undirected graph $G = (V, E)$, $|E| = m$, $|V| = n$ find a function $g : V \to [0, m-1]$ such that the function $h : E \to [0, m-1]$

defined as:

$$h\Big(e = \{u, v\} \in E\Big) = \Big(g(u) + g(v)\Big) \bmod m$$

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 endpoints modulo the number of edges is a unique integer in the range $[0, m-1]$.

This problem is not always solvable if arbitrary graphs are considered. Easy examples are complete graphs on $8k+5$, $k \geq 0$ vertices, K_{8k+5}, or unicyclic graphs on $4k+2$ vertices, C_{4k+2}, for $k > 0$. However, if the graph G is acyclic, a very simple procedure can be used to find values for each vertex, as follows. Associate with each edge e a unique number $h(e) \in [0, m-1]$, in any order. For each connected component choose a vertex v. For this vertex set $g(v)$ to 0. Traverse the graph using a depth-first search (cf. [Tar72]) (or any other regular search on a graph), beginning with vertex v. If vertex w is reached from vertex u, and the value associated with the edge $e = \{u, w\}$ is $h(e)$, set $g(w)$ to $(h(e) - g(u)) \bmod m$. Apply this method to each component of G. (Notice that we have reversed our original problem, by defining the values of the function h first and then searching for suitable values for function g.)

To prove the correctness of the method it is sufficient to show that the value of function g is computed exactly once for each vertex. This property is clearly fulfilled if G is acyclic. The solution to this graph problem becomes the second part of our algorithm for generating the minimal perfect hash function, called the assignment step, and is achieved in deterministic linear time.

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 is mapped into a graph $G = (V, E)$, where $V = \{0, \ldots, n-1\}$, with n determined later, $E = \{\{f_1(w), f_2(w)\} : w \in W\}$, and $f_i : U \to \{0, \ldots, n-1\}$. The step is repeated until graph G is acyclic. 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 $e = \{v_1, v_2\} \in E$ corresponds uniquely to some key w, such that $f_i(w) = v_i$, $1 \leq i \leq 2$, the search for the desired function is straightforward. We simply set $h(e = \{f_1(w), f_2(w)\}) = i - 1$ if w is the i-th 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. For most classes suggested in the literature the selection is a simple (quasi-random) generation of a few, usually numeric, parameters which completely characterize any member of $\mathcal{H}$. 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 a polynomial class of universal hash functions, $\mathcal{H}_n^d = \{f_{\overline{a}} : U \rightarrow \{1, \ldots, n\}, \overline{a} = (a_0, a_1, \ldots, a_d) \in U^{d+1}\}$ where

$$f_{\overline{a}}(w) = \left(\sum_{i=0}^{d} a_i w^i \bmod u \right) \bmod n.$$

To select a member of $\mathcal{H}_n^d$ we need to generate $d+1$ random numbers in the range $[0, u-1]$. Carter and Wegman prove that $\mathcal{H}_n^d$ is $(d+1)$-universal. (For the purposes of our method we require u to be greater than n.)

An alternative class $\mathcal{R}(t, n, d)$, in some senses more random than $\mathcal{H}_n^d$, was introduced and analyzed by Dietzfelbinger and Meyer auf der Heide [DM90]. Here $\mathcal{R}(t, n, d) = \{f : U \rightarrow \{0, \ldots, n-1\}, f = f(\phi, \psi, b_1, \ldots, b_t)$ for some $\psi \in \mathcal{H}_t^d$, $\phi \in \mathcal{H}_n^d$, $b_1, \ldots, b_t \in \{1, \ldots, n\}\}$, and $f = f(\phi, \psi, b_1, \ldots, b_t)$ is defined by

$$f(w) = \left(\phi(w) + b_{\psi(w)} \right) \bmod n.$$

This class is $(d+1)$-universal, as well, but if $t = m^\delta$, $0 < \delta < 1$, many of its probability bounds are the same as for random functions (cf. [DM90]). Another class, $F_{M,G}$, using explicit and randomized constructions of expanders, was suggested by Siegel [Sie89] (expanders are defined and discussed in [Bol85, Chapter 13]). Siegel's functions need linear space and construction time and constant evaluation time, if the universe has size m^k for some constant k. A drawback is that both evaluation time, construction time and space contain a factor exponential in k. 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 Meyer 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 j-th character by $w[j]$. A member of this class, a function $f_i : \Sigma^* \rightarrow \{0, \ldots, n-1\}$ 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. The class is analyzed in the next section.

By treating each character $w[j]$ as a number we obtain an equivalent class, where f_i is defined as:

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

These can be stored in somewhat less space at the expense of greater time for hash function evaluation on common machine architectures (since table lookups are replaced by multiplications). In fact we can characterize suitable functions by as little as one random number, at the expense of greater computation time. However, as we shall see, our space requirements are dominated by the space for storing the function g, so this hardly seems worthwhile.

The above defined classes allow 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 $\sum_{i=1}^{L} |\Sigma|^L = |\Sigma|(|\Sigma|^L - 1)/(|\Sigma| - 1) \sim |\Sigma|^L$ 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. Notice that in the RAM model we assume that we can multiply, divide, etc. two numbers b bits long in constant time. However this is only true if b is bounded by a constant. Otherwise we need at least $O(b)$ time to compute the result. Hence there is an intrinsic connection between the uniform cost measure in the RAM model and the assumption that processing strings character by character takes 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.

To avoid self-loops, we modify the definition of f_2. When it is computed, we check if $f_2(w) = f_1(w)$. If so, we set $f_2(w) = (f_2(w) + r(f_2(w))) \bmod n$, where r generates a "random" number in the range $[1, n-1]$, dependent on its argument. For $n = O(m)$ the size of g is $O(n \log m)$ bits, which is optimal for the class of order preserving minimal perfect hash functions. To store the mapping functions f_i we need $O(\log u)$ bits if the f_i's are selected from $\mathcal{H}_n^d$, $O(m^\delta \log n + \log u)$ bits if $f_i \in \mathcal{R}(m^\delta, n, d)$, $O(\alpha n^\epsilon)$ bits, for fixed $\epsilon < 1$ and α depending on u, if $f_i \in F_{M,G}$ and $O(|\Sigma| L \log n)$ bits if $f_i \in \mathcal{C}_n$. In any of the above cases the size of a program that evaluates the hash function does not exceed $O(m \log m + \log u)$ bits. The $\log u$ factor is not optimal. To obtain the optimal space complexity we need to reduce the size of the universe. The reduction is based on [FKS84, Lemma 2]. By [FKS84, Lemma 2] there exists a prime $q < m^2 \log u$ that does not divide any key in W, and that separates these elements into distinct residue classes mod$q$. The number of bits required by a binary representation of q is clearly $O(\log m + \log\log u)$. Now the keys in W are bounded by $m^2 \log q$ and $W \subseteq U' = \{0, \ldots, u' = m^2 \log q - 1\}$. Consequently, the $\log u'$ factor is bounded by $O(\log m + \log\log u)$ which is optimal. Unfortunately this type of magnitude reduction may require $O(m^2 \log u)$ time. Hence in practice we sacrifice space in order to gain time, especially since the $\log u$ term dominates if and only if $u = \Omega(m^m)$.

4 Complexity analysis

In this section we show that expected time complexity of the algorithm is linear in the number of words. The proofs depend on analysis of random graphs, for which [ER60, Bol85, Pal85] are references.

As the result of the technique used to generate edges of the graph there may be some dependency among them. However, due to the large degree of randomness introduced by the mapping functions, the assumption that the graphs are generated uniformly at random should give quite accurate results, especially since our graphs are quite sparse. We henceforth make this assumption in our theoretical analysis, referring to it as the *uniformity assumption.*

(For character keys this assumption applies as long as $m \ll |\Sigma|^L$, where L is the maximum key length. This is of course the situation in all practical contexts. If m is close to $|\Sigma|^L$ there is no point in generating a minimal perfect hash function, as (almost) all possible keys are present. In such a case the most efficient method is to treat the keys as numbers and simply store them in the locations addressed by these numbers.)

In each iteration of the mapping step, the following operations are executed: (i) generation of tables of random integers; (ii) computation of values of auxiliary functions for each word in a set; (iii) testing if the generated graph G is acyclic. Operation (i) takes time proportional to the maximum length of a word in the set W (times size of alphabet Σ, if two dimensional tables are used). For a particular set and predefined alphabet this may be considered constant. Operations (ii) and (iii) need $O(m)$ and $O(m + n)$ time, respectively. Hence, the complexity of a single iteration is $O(m + n)$.

Let p denote the probability of generating an acyclic graph with m edges and n vertices, and for convenience let $q = 1 - p$. The probability of i iterations in the mapping step is pq^{i-1}. Let Y be a random variable, with probability density function

$$f(y) = \begin{cases} pq^{y-1} & \text{for } y > 0 \\ 0 & \text{otherwise} \end{cases}$$

By standard probability arguments, the mean of Y, which is equal to the expected number of iterations executed in the mapping step, is $1/p$ and its variance is q/p^2. Also, the probability that the number of iterations in the mapping step exceeds some k is q^k.

It remains to determine the threshold function for a random graph to be acyclic. The notion of threshold functions was introduced by Erdös and Rényi [ER60]. They used the following model to study the structural properties of random graphs. A random graph with n vertices is given. At time 0 it has no edges in it, i.e. it has n separate components, each being a single vertex. Then the graph gains new edges at random, one at a time (hence the number of edges may be considered as *time* elapsed since the beginning of the "life" of the graph). Depending on the method of selecting edges we obtain slightly different models. Erdös and Rényi [ER60, ER61] and Bollobás [Bol85] consider a model, called $\mathcal{G}_m$, in which no graph may have self-loops or multiple edges, where at time t an edge is selected from a pool of $\binom{n}{2}-(t-1)$ edges without returning it to the pool. Flajolet, Knuth and Pittel [FKP89] analyze, in addition to $\mathcal{G}_m$, a model, named *the uniform model*, in which at time t a pair of vertices $\{x, y\}$ is generated, where x and y are uniformly distributed between 1 and n, and all n^2 pairs are equally likely. It is easy to observe that model $\mathcal{G}_m$ can be derived from the uniform model by disregarding pairs in which $x = y$ or those where the edge $\{x, y\}$ duplicates a previous edge. If we only reject pairs with $x = y$ we obtain a class of graphs with no self-loops, but allowed to have multiple edges, where at time t an edge is selected from a pool of $\binom{n}{2}$ edges. In this context we introduce model $\mathcal{G}_m^\kappa$, where $\kappa > 0$ denotes the minimum length of a cycle in a random graph. Naturally, model $\mathcal{G}_m$ is now identical to $\mathcal{G}_m^3$ while the uniform model is equivalent to $\mathcal{G}_m^1$.

While studying the evolution of random graphs, Erdös and Rényi discovered that for a number of fundamental structural properties A there exists a function $A(n)$, tending monotonically to ∞ for $n \to \infty$, such that the probability that a graph has property A tends to 1 if $\lim_{n\to\infty} m(n)/A(n) = \infty$ and to 0 otherwise. The $m(n)$ term is the number of edges of the graph depending on n. Many threshold functions for model $\mathcal{G}_m^3$ are given in [ER60], [Bol85] and [Pal85]. We present the analysis for model $\mathcal{G}_m^2$. First we prove the following lemma:

Lemma 1 *Let X_k denote the number of cycles of length k. Let $n = cm$, for some constant c. Then the expected number of cycles of length k is*

$$E(X_k) = \binom{n}{k}\frac{(k-1)!}{2}\frac{m!\left(\binom{n}{2}-k\right)^{m-k}}{(m-k)!\binom{n}{2}^m}$$

while the expected number of multiple edges is

$$E(X_2) = \sum_{2\le j\le m}\binom{n}{2}\frac{\left(\binom{n}{2}-1\right)^{m-j}}{\binom{n}{2}^m}\binom{m}{j}$$

Moreover, for $k \ge 2$ and $n \to \infty$

$$E(X_k) \le \left(\frac{2}{c}\right)^k\frac{1}{2k}$$

Proof. Consider initially only cycles. The total number of graphs with n vertices and m edges in model $\mathcal{G}^2_m$ is $\binom{n}{2}^m/m!$. From k given vertices we can form $\frac{1}{2}(k-1)!$ cycles of order k. These k vertices can be selected in $\binom{n}{k}$ ways. The remaining $m-k$ edges can be arranged in $\left(\binom{n}{2}-k\right)^{m-k}/(m-k)!$ ways. By the above argument the total number of graphs containing a cycle of length k is $\binom{n}{k}\frac{(k-1)!}{2}\left(\binom{n}{2}-k\right)^{m-k}/(m-k)!$. Dividing by the number of graphs gives us the first part of the lemma.

For large n we observe that $\binom{n}{k} \leq n^k/k!$, $\binom{n}{2} \approx n^2/2 = cnm/2$ and $\frac{m!}{(m-k)!} \leq m^k$. Also

$$\frac{\left(\binom{n}{2}-k\right)^{m-k}}{\binom{n}{2}^m} = \frac{\left(1-\frac{k}{\binom{n}{2}}\right)^m}{\binom{n}{2}^k\left(1-\frac{k}{\binom{n}{2}}\right)^k} \leq \binom{n}{2}^{-k}$$

Making the appropriate substitutions in the formula for $E(X_k)$ proves the lemma for $k > 2$.

Now consider multiple edges. This case can be reduced to a classic and well understood problem, the occupancy problem (cf. [Fel68]). It asks for the probability p_j that a specified urn contains exactly j balls out of m randomly distributed among N urns. The answer, an instance of the binomial distribution, is $p_j = \binom{m}{j}\frac{1}{N^j}\left(1-\frac{1}{N}\right)^{m-j}$. In our case we have $N = \binom{n}{2}$ urns, corresponding to all possible edges in a graph, m balls, corresponding to actual edges, and j is the number of edges that comprise a single multiple edge. The expected number of such edges, for $j \geq 2$ is $E(u_j) = Np_j$. Hence $E(X_2) = \sum_{2\leq j\leq m} E(u_j)$.

To obtain the second part of the lemma we notice that $\frac{N}{N^j}\left(1-\frac{1}{N}\right)^{m-j} \leq \frac{1}{N^{j-1}}$ and $N \approx cmn/2$, and consequently:

$$\begin{aligned}
E(u_j) &= N\binom{m}{j}\frac{1}{N^j}\left(1-\frac{1}{N}\right)^{m-j} \\
&\leq \binom{m}{j}\frac{1}{N^{j-1}} \\
&\leq \frac{m^j}{j!}\frac{\left(\frac{2}{c}\right)^j}{2m^j n^{j-2} j!} \\
&= \left(\frac{2}{c}\right)^j \frac{1}{2j!n^{j-2}}
\end{aligned}$$

It is easy to see that for $c \geq 2$ and $j \geq 3$, $\lim_{n\to\infty} E(u_j) = 0$. Hence for $n \to \infty$ the expected number of multiple edges is equal to

$$E(X_2) = E(u_2) = \left(1-\frac{1}{\binom{n}{2}}\right)^{m-2}\frac{\binom{m}{2}}{\binom{n}{2}} \leq \left(\frac{2}{c}\right)^2\frac{1}{2\times 2} = \frac{1}{c^2}.$$

□

Theorem 2 *Let G be a random graph in model $\mathcal{G}^2_m$ with $n = cm$ vertices, for some constant $c > 0$. Then*

$$\lim_{n\to\infty} P(X_k = j) = \frac{\lambda_k^j e^{-\lambda_k}}{j!}$$

where

$$\lambda_k = \lim_{n\to\infty} E(X_k) = \left(\frac{2}{c}\right)^k \frac{1}{2k}$$

Proof. By Lemma 1 and [ER60, Theorem 3a]. □

Corollary 3 *Let $m = cm$, for some constant $c > 2$. Then the probability that a random graph G in model $\mathcal{G}^2_m$ is acyclic, for $n \to \infty$, is*

$$p = \exp\left(-\sum_{k>1} \lim_{n\to\infty} E(X_k)\right).$$

Proof. This follows from Theorem 2 by setting $j = 0$ and summing over all k's. □

Corollary 4 *Let G be a random graph in model $\mathcal{G}^2_m$ with n vertices and m edges. Then if $n = cm$ holds with $c > 2$ the probability that G is acyclic, for $n \to \infty$, is*

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

Proof. To prove the corollary we need to evaluate the sum:

$$\sum_{k>1} \lim_{n\to\infty} E(X_k) = \sum_{k>1} \frac{1}{2} \frac{\left(\frac{2}{c}\right)^k}{k}$$

We denote $\frac{2}{c}$ by a. Thus we have

$$\begin{aligned}
\sum_{k>1} \lim_{n\to\infty} E(X_k) &= \frac{1}{2}\sum_{k>1}\frac{a^k}{k} \\
&= \frac{1}{2}\int \frac{d}{da}\left(\sum_{k>1}\frac{a^k}{k}\right) da \\
&= \frac{-a}{2} + \frac{1}{2}\ln\left(\frac{1}{1-a}\right) \\
&= \frac{-1}{c} + \frac{1}{2}\ln\left(\frac{c}{c-2}\right)
\end{aligned}$$

Hence we compute the probability p of an acyclic graph, for $c > 2$:

$$\begin{aligned} p &= \exp\left(-\sum_{k>1} \lim_{n\to\infty} E(X_k)\right) \\ &= \exp\left(\frac{1}{c} - \ln\sqrt{c/(c-2)}\right) \\ &= e^{1/c}\sqrt{\frac{c-2}{c}}. \end{aligned}$$

□

Corollary 5 *The expected complexity of the probabilistic algorithm if $n = cm$, $c > 2$, is $\Theta(m)$. The average number of iterations is $e^{-1/c}\sqrt{\frac{c}{c-2}}$.*

Remark. This corollary relies on the uniformity assumption. To avoid it one must prove that graphs generated by the algorithm possess some characteristics of truly random graphs. Unfortunately, the question of providing a list of properties for graphs, so that the list can be used to prove "quasi-randomness" has been settled only for dense graphs [CGW89]. For sparse graphs it is still an important open problem. It is however possible to prove that the above defined mapping functions generate "quasi-random" graphs for $m = O(n^2)$.

In the case when $u = O(n)$, that is the magnitude of keys is comparable with the magnitude of values kept in array g we may use a simple technique to reduce the storage requirements of the hash function. We may notice that array g has at most $2m$ entries occupied, as there can be only $2m$ distinct vertices. We extend the g array so than it can hold $3m$ integers and use its unused entries to keep the keys. The degrees of vertices in a random graph are binomially distributed, with parameters $2m$ and $p_d = 1/n$ [Pal85, Chapter 3]. As m tends to infinity the distribution can be approximated by the Poisson distribution, and the number of vertices of degree 0 (i.e. unused locations) is

$$nP\Big(\operatorname{dg}(v \in V) = 0\Big) \to \frac{e^{-2mp_d}(2mp_d)^0}{0!} \approx cm \times e^{-2/c}$$

As a result we can expect to fit all m integers into unused locations of array g for $c > 2.345$.

In both cases, for character and for integer keys, the form of the hash function can be changed to $h(w) = g(f_1(w)) \oplus g(f_2(w))$, where $\oplus$ denotes exclusive or. This gives a slight improvement in speed of generation and evaluation of the hash function. We can speed-up the algorithm even further by early detection of cycles. One possible method is to use a set union algorithm [TVL84]. There are several set union algorithms with the worst-case complexity $O(n + m\alpha(n,n))$, where $\alpha(n,n)$ is a functional inverse of Ackermann's function. This gives us a theoretically inferior solution. However, as pointed out in [TVL84] "for all practical purposes, $\alpha(m,n)$ is a constant no larger than four." In fact $\alpha(n,n) \leq 4$ if $n \leq 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$, with fifteen 2's in the exponent. Moreover, linear time performance of set union algorithms is expected on the average [KS78, BS85, Yao85]. The benefit we get is that unsuccessful attempts are interrupted as soon as possible. We have observed that this type of solution indeed gives better performance of the algorithm, especially for c close to 2.

5 Experimental results

In order to verify the soundness and applicability of the theoretical model we carried out a set of more than 10000 experiments. The new algorithm, without any specific improvements, was implemented in the C language. All experiments were carried out on a Sun Sparc station 2, running under the SunOS^{tm} operating system. We ran separate experiments for integer and character keys. We looked at both random keys and nonrandom keys, with the nonrandom keys selected to provide what might be adverse cases for the algorithm.

For integer keys the results are presented in Table 1. The mapping functions f_i, $1 \leq i \leq 2$ were selected from class $\mathcal{H}_n^3$. Each row in the table represents the average taken over 250 experiments.

m ($c = 2.1$)	iterations random keys	mapping seconds	assignment seconds	total seconds	iterations nonrandom keys
1024	2.350	0.116	0.018	0.133	2.525
2048	2.515	0.244	0.033	0.277	2.365
4096	2.605	0.497	0.065	0.563	2.540
8192	2.295	0.907	0.131	1.038	2.630
16384	2.555	1.952	0.270	2.222	2.730
32768	2.625	3.978	0.553	4.530	2.850
65536	2.950	8.899	1.116	10.015	2.944
131072	2.800	16.921	2.240	19.161	2.708
262144	2.995	35.494	4.496	39.990	2.660
524288	2.750	64.944	9.011	73.955	2.976

Table 1: Experimental results for integer keys.

Entries in the five leftmost columns of the table were generated as follows: for each specified m (number of keys) 250 random keys were selected. In a single experiment m integer keys were randomly selected from the universe $U = \{0, \ldots, 2^{31} - 2\}$. The table entries represent the averages over these 250 trials. The values of *iterations*, *mapping*, *assignment* and *total* are 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. The theoretical predictions fully agree with the experimental data. The rightmost column is discussed below.

The results for character keys treated as sequences of characters from the 26-letter lower-case English alphabet are summarized in Table 2. Words were chosen from 24692 words in a dictionary. The dictionary was obtained by removing from the standard Unix dictionary all words shorter than 3 characters, longer than 18 characters or containing characters other than letters. For each experiment the words were selected using shuffling [Knu73]. For $m > 24692$, artificial sets of random words were generated.

m	iterations random keys c=2.1	mapping seconds	assignment seconds	total seconds	iterations nonrandom keys c=2.22
1024	2.248	0.068	0.016	0.085	2.092
2048	2.540	0.134	0.030	0.164	2.228
4096	2.536	0.246	0.056	0.302	1.916
8192	2.828	0.526	0.123	0.650	2.080
16384	2.620	0.972	0.255	1.227	2.284
24692	2.880	1.565	0.392	1.958	2.248
32768	2.660	2.109	0.529	2.638	2.300
65536	2.700	4.189	1.067	5.256	2.232
131072	2.824	8.582	2.148	10.730	2.544
262144	2.868	18.022	4.620	22.642	2.472
524288	2.756	33.448	8.563	42.011	2.608
1048576	2.711	64.402	17.179	81.581	2.348

Table 2: Experimental results for character keys.

The reason for focussing on this method is its better performance in practical situations, with realistically sized alphabets and key lengths. Notice that computing two polynomials of degree 3 takes substantial time, about twice as much as evaluating two functions from class $\mathcal{C}_n$.

For both integer and character keys we carried out 'torture tests'. Random sets of keys were replaced by sets of m successive keys, with the first key chosen randomly. In the case of character keys the key length was set to the minimum required length. As for the random sets, the average over 250 experiments was computed. The results are shown in the rightmost column of each table, with only the average number of iterations in the mapping step listed.

In the case of integer keys the theoretical predictions were again confirmed. However for the class $\mathcal{C}_n$ we observed a deterioration in performance. The closer c was to 2 the bigger the difference between the model and the observed number of iterations. However for alphabets with more than 4 symbols the observed averages never exceeded more than twice the model means.

With character keys and the class $\mathcal{C}_n$ there are some obvious dependencies in the functions evaluated, especially for small alphabets and c close to 2. The theoretical model does not well reflect this situation and we do not recommend the character key method in these circumstances. Our general recommendation is that the character method is preferable when the alphabet size exceeds the length of the strings.

6 Conclusions

We have described and analyzed a practical method for generating order preserving minimal perfect hashing functions which is optimal. Since it is order preserving the keys may be arbitrarily arranged in the hash table, a property which may be useful in some applications. Theoretical analyses based on random graphs and practical evidence show that the expected time taken to generate the function is linear in terms of the size of the set. The space required for generation and evaluation is almost linear (but the nonlinear component may be ignored in practical application). Evaluation of the hash function is done in fast, constant time, involving little more than two standard hashes.

7 Acknowledgments

The second author was supported in part by Australian Research Council grant A49030651 and in part by DIMACS/Rutgers NSFSTC88-09648.

References

[AA79] M.R. Anderson and M.G. Anderson. Comments on perfect hashing functions: A single probe retrieving method for static sets. *Communications of the ACM*, 22(2):104–105, February 1979.

[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.

[BF83] R.C. Bell and B. Floyd. A Monte Carlo study of Cichelli hash-function solvability. *Communications of the ACM*, 26(11):924–925, November 1983.

[Bol85] B. Bollobás. *Random Graphs*. Academic Press, Inc., London, Orlando, San Diego, New York, Toronto, Montreal, Sydney, Tokyo, 1985.

[BS85] B. Bollobás and I. Simon. On the expected behaviour of disjoint set union algorithms. In *17th Annual ACM Symposium on Theory of Computing – STOC'85*, pages 224–231, May 1985.

[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.

[CGW89] F.R.K. Chung, R.L. Graham, and R.M. Wilson. Quasi-random graphs. *Combinatorica*, 9(4):345–362, 1989.

[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*, 1(4), 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.

[DPK82] N. Deo, G.M. Prabhu, and M.S. Krishnamoorthy. Algorithms for generating fundamental cycles in a graph. *ACM Transactions on Mathematical Software*, 8(1):26–42, March 1982.

[ER60] P. Erdös 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.

[ER61] P. Erdös and A. Rényi. On the evolution of random graphs. *Bull. Inst. Internat. Statis.*, 38:343–347, 1961. Reprinted in J.H. Spencer, editor, *The Art of Counting: Selected Writings*, Mathematicians of Our Time, pages 569–573. 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.

[FHC89] E. Fox, L. Heath, and Q.F. Chen. An $O(n \log n)$ algorithm for finding minimal perfect hash functions. Technical Report TR-89-10, Virginia Politechnic Institute and State University, Blacksburg, VA, April 1989.

[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.

[FKP89] P. Flajolet, D.E. Knuth, and B. Pittel. The first cycles in an evolving graph. *Discrete Mathematics*, 75:167–215, 1989.

[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.

[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.

[Knu73] D.E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Mass., 2nd edition, 1973.

[KS78] D.E. Knuth and A. Schönhage. The expected linearity of a simple equivalence algorithm. *Theoretical Computer Science*, 6:281–315, 1978.

[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.

[Mai83] H.G. Mairson. The program complexity of searching a table. In *24th Annual Symposium on Foundations of Computer Science – FOCS'83*, pages 40–47, Tuscon, AZ, November 1983.

[Meh82] K. Mehlhorn. On the program size of perfect and universal hash functions. In *23rd Annual Symposium on Foundations of Computer Science – FOCS'82*, pages 170–175, Chicago, IL, October 1982.

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

[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.

[Pal85] E.M. Palmer. *Graphical Evolution: An Introduction to the Theory of Random Graphs*. John Wiley & Sons, New York, 1985.

[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*, 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.

[Tar72] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[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.

[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.

[Yao85] A.C. Yao. On the expected performance of path compression algorithms. *SIAM Journal on Computing*, 14:129–133, February 1985.

[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.