An optimal algorithm for generating minimal perfect hash functions[*]

Zbigniew J. Czech
Institute of Computer Science
Silesia University of Technology
44–100 Gliwice
Poland

George Havas
Department of Computer Science
Key Centre for Software Technology
University of Queensland
St. Lucia, Queensland 4072
Australia

Bohdan S. Majewski
Department of Computer Science
Key Centre for Software Technology
University of Queensland
St. Lucia, Queensland 4072
Australia

Abstract

A new algorithm for generating order preserving minimal perfect hash functions is presented. The algorithm is probabilistic, involving generation of random graphs. It uses expected linear time and requires a linear number words to represent the hash function, and thus is optimal up to constant factors. It runs very fast in practice.

Keywords: 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 $h : W \to I$,

[*]appeared in *Information Processing Letters*, 43(5):257–264, October 1992.

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. An overview of perfect hashing is given in [18], §3.3.16 and the area is surveyed in [25]. Some recent independent developments appear in [13, 14, 16].

Various algorithms with different time complexities have been presented for constructing perfect or minimal perfect hash functions, including [3, 4, 5, 6, 7, 8, 17, 10, 19, 20, 22, 30]. In 1985 Sager proposed the mincycle algorithm [28] which uses graph considerations. The author claimed that the mincycle algorithm has complexity $O(m^4)$. Based on this algorithm other solutions have been developed [9, 14, 15, 16], with mainly experimental evidence of time performance.

We present a new algorithm based on random graphs for finding 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 strings into integers, and g is a function that maps integers into $[0, m-1]$. We show that the expected time complexity is $O(m)$. The space required to store the generated function is $O(m \log m)$ bits, which is optimal for order preserving minimal perfect hash functions (see [21]).

2 The new algorithm

Consider the following problem. For a given undirected graph $G = (V, E)$, $|E| = m$, $|V| = n$ find a function $g : V \rightarrow [0, m-1]$ such that the function $h : E \rightarrow [0, m-1]$ 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 taken 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. 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 a unique number $h(e) \in [0, m-1]$ in any order. For each connected component of G choose a vertex v. For this vertex, set $g(v)$ to 0. Traverse the graph using a depth-first search (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 the above method to each component of G. Pseudocode is given in Fig. 2, which solves a problem like that addressed in [27]. (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 and is called the assignment step.

Now we are ready to present the new algorithm for generating a minimal perfect hash function. We denote the length of the word w by $|w|$ and its i-th character by $w[i]$. The algorithm comprises two steps: mapping and assignment. In the mapping step a graph $G = (V, E)$ is constructed, where $V = \{0, \ldots, n-1\}$ with n determined later, and $E = \{(f_1(w), f_2(w)) : w \in W\}$. We introduce auxiliary functions f_1 and f_2 which are designed to be two independent random functions mapping W into $[0, n-1]$. There are various possibilities. Here we choose the functions to be:

$$f_1(w) = \left(\sum_{i=1}^{|w|} T_1(i, w[i])\right) \bmod n$$

$$f_2(w) = \left(\sum_{i=1}^{|w|} T_2(i, w[i])\right) \bmod n$$

where T_1 and T_2 are tables of random integers modulo n for each character and for each position of a character in a word.

The space required by tables T_1 and T_2 is $O(\log n)$ bits, since each entry is a number in the range $[0, n-1]$ and there is in effect a constant number of entries (actually dependent on the length of keys and the size of character set). As long as n fits into one computer word this is $O(1)$ words. If n is not less than the alphabet size, by treating each character $w[i]$ as a number we obtain another suitable pair of mapping functions:

$$f_k(w) = \left(\sum_{i=1}^{|w|} T_k(i) \times w[i]\right) \bmod n.$$

These can be stored in 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 even greater computation time. However our space requirements for increasing m are dominated by the space for storing the function g, so such considerations are of interest only for small m.

Our goal is to find values of T_1 and T_2 so that the graph G is acyclic. Because we have no easy deterministic method for doing this, we randomly generate tables repeatedly, until we obtain an acyclic graph (see Fig. 1).

Once an acyclic graph is generated the assignment step is executed. Notice that generating a minimal perfect hash function can be reduced to the problem described at the beginning of this section. For an acyclic graph, each edge $e = (u, v) \in$

```
repeat
    initialize $E := \emptyset$;
    randomly generate tables $T_1$ and $T_2$;
    for $w \in W$ loop
        $f_1(w) := \left(\sum_{j=1}^{|w|} T_1(j, w[j])\right) \bmod n$;
        $f_2(w) := \left(\sum_{j=1}^{|w|} T_2(j, w[j])\right) \bmod n$;
        add the edge $(f_1(w), f_2(w))$ to graph $G$;
    end loop;
until $G$ is acyclic;
```

Figure 1: The mapping step

```
procedure traverse(u : vertex);
begin
    visited[u] := TRUE;
    for $w \in$ neighbours($u$) loop
        if not visited[$w$] then
            $g(w) := (h(e = (u, w)) - g(u)) \bmod m$;
            traverse($w$);
        end if;
    end loop;
end traverse;

begin
    visited[$v \in V$] := FALSE;
    for $v \in V$ loop
        if not visited[$v$] then
            $g(v) := 0$;
            traverse($v$);
        end if;
    end loop;
end;
```

Figure 2: The assignment step

```
function h(w : string) : integer;
begin
    u := $\left(\sum_{j=1}^{|w|} T_1(j, w[j])\right) \bmod n$;
    v := $\left(\sum_{j=1}^{|w|} T_2(j, w[j])\right) \bmod n$;
    return $\left(g(u) + g(v)\right) \bmod m$;
end;
```

Figure 3: Evaluating the hash function

E corresponds uniquely to some word w (such that $f_1(w) = u$ and $f_2(w) = v$) so 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. Then values of function g for each $v \in V$ are computed by the assignment step. The function h is a minimal perfect hash function for W.

Evaluation of the hash function is done in fast, constant time, involving little more than two standard hashes. Pseudocode is given in Fig. 3.

3 Complexity analysis

In this section we show that expected time complexity of the algorithm is linear in the number of words.

As a result of the technique used to generate edges of the graph there is some dependency among them. However, due to the large degree of randomness introduced by the mapping functions, the assumption that the m-edged 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. We also treat the alphabet size and maximum key length as constants, a reasonable assumption for any specific application area. (In fact m is bounded by the alphabet size raised to the maximum key length, but this is not a practical restriction.)

The second step of the algorithm, assignment, runs in $O(m + n)$ time. 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 at most time proportional to the maximum length of a word in the set W times size of alphabet Σ, which is a constant. Operations (ii) and (iii) need $O(m)$ and $O(m + n)$ time, respectively. Hence, the complexity of a single iteration is $O(m + n)$.

We now show that the expected number of iterations in the mapping step can be made constant by suitable choice of n. Let p_a denote the probability of generating an acyclic graph with m edges and n vertices. Let X be a random variable such that $p(X = i) = p_a(1 - p_a)^{i-1}$. By standard probability arguments, the mean of X, which is equal to the expected number of iterations executed in the mapping step, is $1/p_a$ and its variance is $(1-p_a)/p_a^2$. Also, the probability that the number of iterations in the mapping step exceeds some k is $(1-p_a)^k$.

To obtain a high probability of generating an acyclic graph in an iteration we must deal with very sparse graphs. We choose $n = cm$, for some constant c. Detailed probabilistic arguments appear in [26]. Briefly, they proceed as follows. For random labeled graphs with m edges and $n = cm$ vertices as $n \to \infty$, the expected number of cycles of length k tends towards $2^k/(2kc^k)$ [1, p. 98]. This result is for graphs with no self-loops ($k = 1$) or multiple edges ($k = 2$), however it may be extended to cover them. Then, the probability of having an acyclic graph tends towards $\exp\left(-\sum_{k=1}^{n} 2^k/(2kc^k)\right)$ [12]. Since, for $c > 2$, $\lim\limits_{n\to\infty} \sum_{k=1}^{n} 2^k/(2kc^k) = \frac{1}{2}\ln\left(\frac{c}{c-2}\right)$, the probability of getting an acyclic graph tends towards $p_a^\infty = \sqrt{\frac{c-2}{c}}$. For $c \leq 2$, $p_a^\infty = 0$.

Thus, for $c > 2$ the probability of generating an acyclic graph approaches a nonzero constant, so we choose $n > 2m$. For $n = 3m$, the expected number of iterations for large m is $E(X) \approx 1/p_a^\infty = \sqrt{3}$. Therefore, the complexity of the mapping step is $O(m+n)$, and the complexity of the algorithm is $O(m+n)$. Since $n = cm$ the complexity of the algorithm is linear in m, the number of words.

We can slightly improve the performance of the algorithm by modifying the functions f_1 and f_2 so that there are no self-loops. One way is to change the definition of f_2 to ensure that $f_2(w) \neq f_1(w)$, another is to generate bipartite graphs. For the former case $p_a^\infty = e^{1/c}\sqrt{\frac{c-2}{c}}$. If bipartite graphs are generated, the probability of generating a cycle-free graph increases to $p_a^\infty = \frac{\sqrt{c^2-4}}{c}$. Other improvements can be made if special properties of the words in W are taken into account.

4 A simple example

Consider the set of 12 month names, abbreviated to the first three characters. We want to construct a minimal perfect hash function so that the i-th month, $i \in \{1,\ldots,12\}$ is kept in the $(i-1)$-th location of the hash table.

We select c to be $2\frac{1}{12}$, hence $n = 25$. Moreover we notice that the second and the third characters of the keys are unique for any key, therefore we restrict the definition of tables T_1 and T_2 so that there are only two rows in each table. The space required to store such tables is $2 \times 2 \times 26 = 104$ bytes. Suppose that in the mapping step the randomly generated contents of tables T_1 and T_2 are as shown in Fig. 4 *a)*, with unused letters omitted. Then, for each key we compute the edge, which corresponds to it. Thus we have: $f_1(jan) := (T_1(2,a) + T_1(3,n)) \bmod 25 = (11+19) \bmod 25 = 5$, $f_2(jan) := (T_2(2,a) + T_2(3,n)) \bmod 25 = (5+7) \bmod 25 = 12$; $f_1(feb) := (13 + 9) \bmod 25 = 22$, $f_2(feb) := (21+9) \bmod 25 = 5$; $f_1(mar) := (11+1) \bmod 25 = 12$, $f_2(mar) := (5+17) \bmod 25 = 22$. The last edge has closed a cycle $(5,12,22)$ and there is no point in computing edges for the remaining keys with the current contents of tables T_1 and T_2. (Although the option of early detection of cycles was not included in the pseudocode given in Fig. 1, it is quite easy to implement. We use a set union

a)

T_1

	a	b	c	e	g	l	n	o	p	r	t	u	v	y
2	11		1	13				21	17			1		
3		9	21		13	5	19		20	1	0		3	12

T_2

	a	b	c	e	g	l	n	o	p	r	t	u	v	y
2	5		2	21				24	8			12		
3		9	23		5	2	7		12	17	2		11	8

b)

T_1

	a	b	c	e	g	l	n	o	p	r	t	u	v	y
2	19		3	14				7	20			24		
3		11	21		15	14	10		3	2	17		1	15

T_2

	a	b	c	e	g	l	n	o	p	r	t	u	v	y
2	3		13	7				11	21			22		
3		10	12		19	3	10		2	8	1		24	15

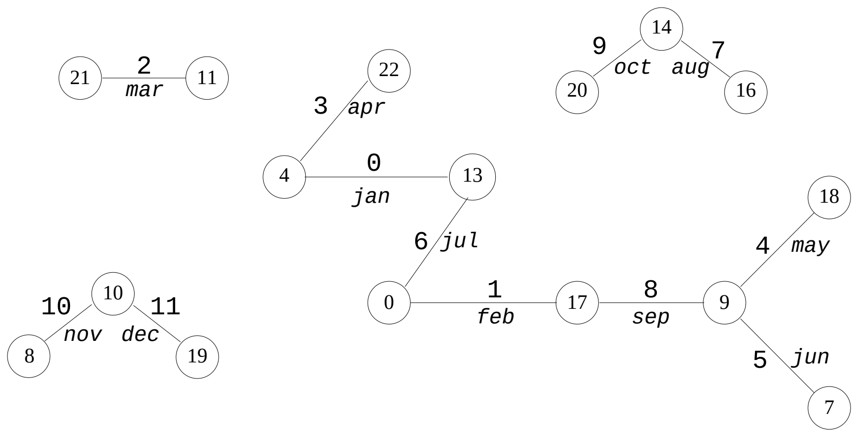
Figure 4: Contents of the mapping tables: *a)* during the first iteration; *b)* during the second iteration

algorithm [29] to do so. This results in a theoretically inferior solution, as the best set union algorithms have worst-case complexity $O(n + m\alpha(n, n))$, where $\alpha(n, n)$ is the functional inverse of Ackermann's function. However, linear time performance of set union algorithms is expected on the average [24, 2, 31], and, as the authors of [29] point out "for all practical purposes, $\alpha(m, n)$ is a constant no larger than four.")

Because of the cycle, the mapping process has to be repeated. The contents of tables T_1 and T_2 generated in the second iteration are shown in Fig. 4 *b)*. This time the mapping leads to an acyclic graph, shown in Fig. 5. In the assignment step for each connected component we select a vertex and assign 0 to it. Then we perform a regular search on the component, computing the values associated with the remaining vertices.

We start with vertex 0, hence $g(0) := 0$. Suppose we explore the right branch first. Thus $g(17) := (1 - g(0)) \bmod 12 = 1$, $g(9) := (8 - g(17)) \bmod 12 = 7$, $g(18) := (4 - g(9)) \bmod 12 = 9$ and $g(7) := (5 - g(9)) \bmod 12 = 10$. Next, after returning to vertex 0, we explore the left branch. Here we set $g(13) := (6 - g(0)) \bmod 12 = 6$, $g(4) := (0 - g(13)) \bmod 12 = 6$ and $g(22) := (3 - g(4)) \bmod 12 = 9$. This ends the assignment step for the largest component. The same procedure is then applied to the remaining components. It is easy to see that $g(8) = 0$, $g(10) = 10$, $g(19) = 1$; $g(11) = 0$, $g(21) = 2$ and $g(14) = 0$, $g(16) = 7$, $g(20) = 9$ are suitable values. This ends the generation phase of the hash function.

Now, to calculate the hash table address for *nov*, say, we compute $f_1(nov) := (7+1) \bmod 25 = 8$ and $f_2(nov) := (11+24) \bmod 25 = 10$. Then the hash table address of *nov* is $(g(8)+g(10)) \bmod 12 = (0+10) \bmod 12 = 10$. (With no extra work(!) we have gained also the information that *nov* is the $10+1 = 11$-th month of the year.)



	0	4	7	8	9	10	11	13	14	16	17	18	19	20	21	22
g	0	6	10	0	7	10	0	6	0	7	1	9	1	9	2	9

Figure 5: The graph generated in the second iteration of the mapping step

5 Experimental results

The new algorithm, without any specific improvements, was implemented in the C language. All experiments were carried out on Sun SPARC station 2, running under the SunOS^{tm} operating system. The results are summarized in Table 1. An entry in the table produced for the algorithm was generated as follows: for each specified m (number of words) 250 random sets of words were selected. The table entries represent the averages over these 250 trials. 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 [23]. For $m > 24692$, artificial sets of random words were generated. The values of m, *iterations*, *mapping*, *assignment* and *total* are the number of words, 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.

The experimental results fully back the theoretical considerations. Also, the time requirements of the new algorithm are very low. Observe that the average number of iterations is approximately equal to $\sqrt{3}$ as indicated by the theory. Likewise

$m = n/3$	iterations	mapping	assignment	total
512	1.704	0.037	0.010	0.047
1024	1.684	0.052	0.019	0.072
2048	1.776	0.095	0.037	0.132
4096	1.676	0.169	0.067	0.236
8192	1.668	0.320	0.142	0.463
16384	1.680	0.628	0.293	0.921
24692	1.688	0.950	0.444	1.394
32768	1.636	1.353	0.597	1.949
65536	1.696	2.718	1.198	3.916
131072	1.676	5.448	2.416	7.864
262144	1.768	11.273	4.813	16.087
524288	1.736	22.493	10.414	32.907

Table 1: Experimental results

the mapping, assignment and total times grow approximately linearly with m. A comparison with the timing results given in [16] reveals that this algorithm is much faster than that given there. For example, their algorithm took 763.07 seconds to generate a minimal perfect hash function for 524288 keys on a Sequent machine.

In the implementation of the algorithm we used an edge-oriented representation of graphs [11]. This allowed us to handle edges as concrete objects, represented by integers, and not as pairs of vertices. Because of this, the space complexity of the algorithm is linear in the number of words too, with a very small constant factor.

6 Conclusions

A new algorithm for generating order preserving minimal perfect hash functions has been developed. The expected time complexity of the algorithm is $O(m)$, so the algorithm is time optimal. Its space complexity, also optimal, is $cm \log m + O(1) \log n$ bits, or $cm + O(1)$ words, as long as n fits into a word. Observe that the i-th word of W is placed at $(i-1)$-th location of the hash table, hence the generated hash function preserves the order of the words in an input. This allows arbitrary arrangement of them, which may be useful in some applications. The generated function is quickly computable, and the space needed to store it may be made as small as $m(2+\epsilon)$, $\epsilon > 0$. Extensive experimental results have confirmed the theoretical results. They also have shown that the time requirements of the new algorithm are very low, even for very large sets.

7 Acknowledgement

We thank Nick Wormald of the University of Melbourne for help with calculating the probability of generating an acyclic graph. We also thank the anonymous referees whose thoughtful comments led to an improved presentation. The second author was supported in part by Australian Research Council grant A49030651 and in part by DIMACS/Rutgers NSFSTC88-09648.

References

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

[2] 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.

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

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

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

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

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

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

[9] Z.J. Czech and B.S. Majewski. Generating a minimal perfect hashing function in $O(m^2)$ time. *Archiwum Informatyki*, 1(4), 1992.

[10] 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.

[11] J. Ebert. A versatile data structure for edge-oriented graph algorithms. *Communications of the ACM*, 30(6):513–519, June 1987.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

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

[21] 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.

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

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

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

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

[26] 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.

[27] T.J. Sager. A new method for generating minimal perfect hash functions. Technical Report CSc–84–15, University of Missouri-Rolla, Department of Computer Science, 1984.

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

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

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

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