A Linear Time Algorithm for Finding Minimal Perfect Hash Functions

ZBIGNIEW J. CZECH* AND BOHDAN S. MAJEWSKI†

* *Institutes of Computer Science, Silesia University of Technology, and Polish Academy of Sciences, 44-100 Gliwice, Poland*

† *Key Centre for Software Technology, University of Queensland, Queensland 4072, Australia*

A new algorithm for finding minimal perfect hash functions (MPHF) is proposed. The algorithm given three pseudorandom functions h_0, h_1 and h_2, searches for a function g such that $F(w) = (h_0(w) + g(h_1(w)) + g(h_2(w)))$ mod m is a MPHF, where m is a number of input words. The algorithm involves generation of random bipartite graphs and runs in linear time. The hash function generated is represented by using $2m + O(1)$ memory words of log m bits each. The empirical observations show that the algorithm runs very fast in practice.

Received July 7, 1992, revised November 25, 1992

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 $k \geqslant m$. 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 words are mapped into 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, $k \geqslant m$. If $k = m$, then we say that h is a *minimal perfect hash function* (MPHF). 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.

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 [20, section 3.3.16] and the area is surveyed in [27].

Various algorithms with different time complexities have been presented for constructing perfect or minimal perfect hash functions, including [4–8, 19, 21–23, 34]. In 1985 Sager proposed the mincycle algorithm [32] 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 [12, 13, 16–18, 28].

Adopting the general approach taken by Sager [32], we present a new algorithm based on random graphs. We show that the expected time complexity of the algorithm is practically linear. It requires also a linear number of memory words to represent the hash function.

2. THE MINCYCLE ALGORITHM

The *mincycle* algorithm proposed by Sager [32] searches for the MPHF of the form:

$$h(w) = (h_0(w) + g(h_1(w)) + g(h_2(w))) \bmod m$$

where h_0, h_1 and h_2 are auxiliary pseudorandom functions, and g is a function, implemented as a lookup table, whose values are established during the exhaustive search. The algorithm uses a *mapping → ordering → searching* (MOS) approach[1] in which the construction of the MPHF is accomplished in three steps. First, the *mapping* step transforms a set of words to a set of *triples* of integers. The mapping step has to preserve the 'uniqueness', i.e. if two words are distinguishable in the original universe so they have to be in the new one. In general, this property is hard to achieve, and may require a significant effort. Second step, *ordering*, places the words in a sequential collocation that determines the precedence in which hash values are assigned to words. Words are divided into subsets, $W_0, W_1, \ldots, W_k$, such that $W_0 = \emptyset$, $W_i \subset W_{i+1}$ and $W_k = W$, for some k. The sequence of these subsets is called a *tower* and each subset $X_i = W_i - W_{i-1}$ in the tower is called a *level*. The hash values must be assigned to the members of a level at the same time, since assignment of a hash value to any of them determines the hash values for all others. The third step, *searching*, tries to extend the desired function h from the domain W_{i-1} to W_i. This is the only step of potentially exponential time complexity, since if the searching step encounters W_i for which h cannot be extended, it backtracks to earlier subsets, assigns differ-

[1] This classification was introduced by Fox *et al.* [16].

ent hash values to the words of these subsets and tries again to recompute hash values for successive subsets.

We shall discuss the first two steps of the mincycle algorithm in more detail. In the mapping step, three auxiliary pseudorandom functions are defined:

$$h_0(w) = |w| + \sum_{j=1 \text{ by } 3}^{|w|} \text{ord}(w[j]),$$

$$h_1(w) = \left(\sum_{j=1 \text{ by } 2}^{|w|} \text{ord}(w[j]) \right) \bmod r,$$

$$h_2(w) = \left(\left(\sum_{j=2 \text{ by } 2}^{|w|} \text{ord}(w[j]) \right) \bmod r \right) + r.$$

Integer r is a parameter of the mincycle algorithm. It determines the size of table g, $|g| = 2r$, containing the description of the MPHF. Therefore, it is desirable for g to be as small as possible. Sager chooses r to be the smallest power of 2 greater than $m/3$. The functions defined above convert each word w into a hopefully unique triple of integers $(h_0(w), h_1(w), h_2(w))$. When this condition is not satisfied, the functions h_0, h_1 and h_2 must be modified. However, Sager does not provide a generic method suitable to accomplish such a modification.

In the ordering step the tower of subsets is constructed. First, the dependency graph $G = (R, E)$,

$$R = \{h_1(w) | w \in W\} \cup \{h_2(w) | w \in W\},$$

$$E = \{(h_1(w), h_2(w)) | w \in W\}$$

for a set of words is built. Each word is associated with edge $e = (h_1(w), h_2(w))$. The dependency graph that is bipartite and consists of $|R| = n = 2r$ vertices and $|E| = m$ edges represents constraints among words. Observe that allocating a place in the hash table for word w requires selecting the value $U(w) = g(h_1(w)) + g(h_2(w))$. There may exist a sequence of words $\{w_0, w_1, \ldots, w_{k-1}\}$ such that $h_1(w_i) = h_1(w_{i+1})$ and $h_2(w_{i+1}) = h_2(w_{(i+2) \bmod k})$, for $i = \{0, 2, 4, \ldots, k-2\}$. Once words $w_0, w_1, \ldots, w_{k-2}$ are allocated some places in the hash table, both $g(h_1(w_{k-1}))$ and $g(h_2(w_{k-1}))$ are set (note that $h_1(w_{k-2}) = h_1(w_{k-1})$ and $h_2(w_{k-1}) = h_2(w_0)$). Hence, word w_{k-1} cannot be allocated an arbitrary place, but must be placed in the hash table at location

$$h(w_{k-1}) = (h_0(w_{k-1}) + U(w_{k-1})) \bmod m.$$

In our sequence, the words $w_0, w_1, \ldots, w_{k-2}$ are independent, i.e. they have a choice of a place in the hash table, whereas the word w_{k-1} is dependent, i.e. it has not such a choice. We shall call these words as *canonical* and *noncanonical*, respectively. It is easy to see that

$$U(w_{k-1}) = g(h_1(w_{k-1})) + g(h_2(w_{k-1})) = \sum_{p \in path(w_{k-1})} (-1)^p U(w_p)$$

where $path(w_{k-1})$ is a sequence of words $\{w_0, w_1, \ldots, w_{k-2}\}$, and thus

$$h(w_{k-1}) = (h_0(w_{k-1}) + \sum_{p \in path(w_{k-1})} (-1)^p U(w_p)) \bmod m.$$

If the place $h(w_{k-1})$ is occupied, a collision arises and no minimal perfect hash function for selected values of g can be found. In such a case, the search step executes backtrack and tries to find different values of g that do not lead to a collision.

This dependency of words is reflected in the dependency graph by cycles. There may be, and usually are, many cycles. Each of them corresponds to a sequence of words similar to the described. The core of Sager's heuristic of the ordering step is to find such an order of words, that words without a choice are processed by the searching step as soon as possible. The ith level of the tower, W_i, $i \geqslant 1$, is equal to $W_{i-1} \cup X_i$, where $W_0 = \emptyset$ and X_i is a group of words selected by the following method. Choose an edge (possibly a multiple edge[2]) lying on a maximal number of minimal length cycles in the dependency graph. Let X_i be all words associated with the chosen edge. Remove the edge from the graph and merge its endpoints. Repeat this procedure until all edges are removed from the dependency graph. The maximal number of such repetitions is $O(m)$ and an edge lying on maximal number of cycles of minimum length can be selected in $O(r^3) = O(m^3)$ time [32]. Consequently, the time complexity of the ordering step is $O(m^4)$.

The heuristic which gave the name to the whole algorithm tries to ensure that each time an edge is selected, it is done in such a way that the maximum number of dependent words is placed in a tower. Moreover, because the selected edge not only lies on a cycle of minimum length but on the maximal number of such cycles, it ensures that in subsequent steps the selection will be optimal or at least close to optimal. For a multiple edge, we select an arbitrary word to be canonical, and the remaining become noncanonical. A single edge always corresponds to a canonical word.

Based on experimental results Sager claimed that the time required to complete the ordering step dominates over the potentially exponential time of the searching step. Hence, the 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 experimental evidence shows that although the mincycle algorithm performs well for sets of size up to about 500 words, it is impractical for sets larger than this limit. The reasons are:

1. The running time of the algorithm ($O(m^4)$) is closer to hours than minutes for sets of more than 500 words, even for fast machines.
2. The memory requirement of the algorithm is very high. The mincycle algorithm, even when implemented very carefully, needs $O(m^2)$ space, with quite a large constant factor. For example, for relatively

[2] A multiple edge constitute the words w with identical pairs $(h_1(w), h_2(w))$.

small sets (up to 700 words) it uses more than 2 MB of memory.
3. Poor pseudorandom functions, that do not yield distinct triples when used with sets of several hundred words.

3. THE NEW ALGORITHM

The class of functions we search for is the same as in Sager's method. However, by setting the parameter $r = m$, i.e. $n = 2m$, we keep the size of dependency graphs to be exactly a linear function of the size of word sets. Furthermore, the dependency graphs we deal with are sparse. We shall see that this gives us a very fast searching step. The new algorithm adopts the MOS approach.

3.1. The Mapping Step

The mapping step computes three tables of random integer numbers, T_0, T_1 and T_2, one for each of the functions h_0, h_1 and h_2. Each table contains a random number for each position i in the word, and each possible character at this position. Given a word as the character string $w = a_1 a_2 \ldots a_{|w|}$, the triple is computed using the following formulas:

$$h_0(w) = \left(\sum_{i=1}^{|w|} T_0[i_m, a_i] \right) \bmod m,$$

$$h_1(w) = \left(\sum_{i=1}^{|w|} T_1[i_m, a_i] \right) \bmod r,$$

$$h_2(w) = \left(\left(\sum_{i=1}^{|w|} T_2[i_m, a_i] \right) \bmod r \right) + r,$$

where $i_m = ((i + |w| - 1) \bmod |w|_{max}) + 1$ and $|w|_{max}$ is the maximum word length. The values $h_1(w)$ and $h_2(w)$ define the dependency graph $G_0(R, E_0)$, where $\forall w \in W$, $R = \{h_1(w)\} \cup \{h_2(w)\}$ and $E_0 = \{(h_1(w), h_2(w))\}$. The idea of using random number tables to find a triple is due to Fox *et al.* [17]. We slightly modify their method by starting to fetch the numbers from the tables at a position that depends on the word length. This ensures more even utilization of the table elements and increase the randomness of dependency graphs that are constructed from triples.

While generating the triples it is essential that all of them are distinct for a given set of words. Let $t = mr^2$ be the size of the universe of triples and assume that the triples $(h_0(w), h_1(w), h_2(w))$, $w \in U$ (U is a universe the words are chosen from), are random.[3] Then, the probability of distinctness for m triples chosen uniformly at random from t triples is [17]:

$$p(m, t) = \frac{t(t-1)\cdots(t-m+1)}{t^m} \sim \exp\left(-\frac{m^2}{2t}\right).$$

[3] The method used to compute the triples implies that there is some dependency among them. However, owing to the large degree of randomness introduced by tables T_0, T_1 and T_2, the assumption that the triples, and thus the m-edged dependency graphs, are generated uniformly at random should give quite accurate results, especially since our graphs are sparse.

Since $t = mr^2$ and $r = m$, we have

$$p(m, t) \sim \exp\left(-\frac{1}{2m}\right).$$

Thus, the probability to get distinct triples goes quickly to 1 for increasing m.

Consider the time complexity of the mapping step. The generation of the tables of random numbers takes time proportional to the maximum length of a word in the set W times the size of alphabet Σ. This can be considered constant for a particular set and a predefined alphabet. Since the generation of triples needs $O(m)$ time, the time complexity of the mapping step is $O(m)$.

As we have already mentioned, the table g containing the description of the MPHF is of size $|g| = 2r$. In addition to g, in the new algorithm we have to store also the tables T_0, T_1, and T_2. Hence, since $2r = 2m$, we need $2m + O(1)$ memory words of $\log m$ bits each, to represent the MPHF. This memory requirement is higher than Mehlhorn's lower bound of $1.4427m/\log m$ memory words [29, 16].

3.2. The Ordering Step

In the ordering step the tower is built. It contains only the words of W whose hash values depend on each other. These words correspond to cycles and multiple edges in the dependency graph. All other words of W are *free*, i.e. an arbitrary hash value can be assigned to each of them. We shall denote the set of free words as W_a. Consequently, after generating the dependency graph G_0, the subgraph $G = (R, E)$ that contains only the multiple edges and the edges lying on cycles is constructed. For this purpose the biconnected components of G_0 are found. The edges of the biconnected components of size greater than 1 are placed into E. The algorithm to find biconnected components of G_0 in $O(|R| + |E_0|) = O(m)$ steps is well known [1].

Our heuristic for building the tower is much simpler than the Sager's. Namely, once the dependency graph G is constructed, we first put the multiple edges $\mathscr{E}$ into the tower beginning with the edges of highest multiplicity. These edges are then deleted from G and a set of the fundamental cycles for the resultant graph G_1 is found by using the breadth-first search. The fundamental cycles are considered in order of increasing length. The edges of the shortest cycle are put to the tower one at a time. Each edge is then deleted from all the cycles it lies on, and the next shortest cycle is considered. While building the tower we maintain the spanning forest $\mathscr{F}$ that consists of the edges corresponding to the canonical words in the tower. The forest is used to determine the *paths* for non-canonical words (Figure 1).

We now present three lemmas that estimate the expected total length of cycles, the expected number of cycles, and the expected number of multiple edges, in a random bipartite graph. We shall consider the *uniform model* of graph evolution [15]. In this model the life of

```
F := ∅;
i := 0; -- current level of the tower
for each multiple edge E in G do
  add to the tower an arbitrary edge e ∈ E as canonical word and
    the remaining edges of E as noncanonical words;
  delete E from G by merging the endpoints of E;
  i := i + 1; -- begin the next level of the tower
  F := F ∪ e; -- add e to the forest
end for;
find fundamental cycles in G1 = G − {E} and structure them into heap;
form the sets of cycles members[e] each edge e in G1 belongs to;
while heap not empty do
  c := del_min(heap); -- take the shortest cycle from the heap
  for e in c do
    if e is not in the tower then
      if there is a path between the end vertices of e in F then
        add e to the tower as noncanonical word;
        make path(e) = list of ⟨sign: (+1, −1); t: 1 .. u⟩;
      else
        i := i + 1;
        add e to the tower as a canonical word;
        F := F ∪ e;
      end if;
      for j in members[e] do
        decrease length of cycle[j];
        restore the heap; -- move cycle[j] up
      end for;
    end if;
  end for;
end while;
```

FIGURE 1. Building the tower.

the graph begins as a set of $n = 2r$ separate vertices. Then it evolves by gaining m edges in the following procedure. At each step we generate an ordered pair (x, y), where x and y are uniformly distributed between 1 and r, and all r^2 pairs are equally likely. The nondirected edge x–y is then added to the graph. In this way we obtain a multigraph, which may have multiple edges. In addition, we shall take into account the order in which graphs gain their edges. The total number of bipartite graphs of $n = 2r = 2m$ vertices and m edges, that evolve in the way described above is m^{2m}.

LEMMA 3.1. Let C_{2k} denote the number of cycles of length $2k$. Then the expected total length of cycles in the graph is

$$\sum_{k=1}^{m/2} 2k \cdot E(C_{2k}) = O(\sqrt{m}).$$

Proof. To build a cycle of length $2k$, we select $2k$ vertices and connect them with $2k$ edges in any order. There are $\binom{m}{k}^2$ ways to choose $2k$ vertices out of $2m$ vertices of a graph, $k!k!/2k$ ways of connecting them into a cycle, and $(2k)!$ possible orderings of the edges. The cycle can be embedded into the structure of the graph in $\binom{m}{m-2k} m^{2(m-2k)}$ ways. Hence, the number of graphs containing a cycle of length $2k$ is $\binom{m}{k}^2((k!)^2/2k)(2k)!\binom{m}{m-2k}m^{2(m-2k)}$. Therefore, the expected number of cycles of length $2k$ in the graph is

$$E(C_{2k}) = \frac{\binom{m}{k}^2 ((k!)^2/2k)\,(2k)!\,\binom{m}{m-2k}\, m^{2(m-2k)}}{m^{2m}}$$

$$= \frac{1}{2k}\left(\frac{m(m-1)\cdots(m-k+1)}{m^k}\right)^2 \times \frac{m(m-1)\cdots(m-2k+1)}{m^{2k}}.$$

Using an asymptotic estimate from Palmer [30, p. 129]

$$\frac{(m)_i}{m^i} = \frac{m(m-1)\cdots(m-i+1)}{m^i} \sim e^{-i^2/2m - i^3/6m^2}$$

we approximate the sum with an integral

$$\sum_{k=1}^{m/2} 2k \cdot E(C_{2k}) = \sum_{k=1}^{m/2} \left(\frac{m(m-1)\cdots(m-k+1)}{m^k}\right)^2 \times \frac{m(m-1)\cdots(m-2k+1)}{m^{2k}}$$

$$\sim \int_1^{m/2} e^{-3k^2/m}\,dk \leqslant \sqrt{\frac{m}{3}} \int_0^{\infty} e^{-z^2}\,dz$$

$$= \sqrt{\frac{m}{3}}\,\frac{\sqrt{\pi}}{2} \approx \frac{1}{2}\sqrt{m} = O(\sqrt{m}). \qquad \square$$

LEMMA 3.2. The expected number of cycles in the graph is

$$\sum_{k=1}^{m/2} E(C_{2k}) = O(\ln m).$$

Proof. We have

$$\sum_{k=1}^{m/2} E(C_{2k}) = \sum_{k=1}^{m/2} \frac{1}{2k}\left(\frac{m(m-1)\cdots(m-k+1)}{m^k}\right)^2 \times \frac{m(m-1)\cdots(m-2k+1)}{m^{2k}}$$

$$\sim \sum_{k=1}^{m/2} \frac{1}{2k} e^{-3k^2/m}$$

$$= \frac{1}{2}\left[\sum_{k=1}^{\sqrt{m}} \frac{e^{-3k^2/m}}{k} + \sum_{k=\sqrt{m}+1}^{m/2} \frac{e^{-3k^2/m}}{k}\right]$$

$$\leqslant \frac{1}{2e^{3/m}} H_{\sqrt{m}} + \frac{1}{2e^3}(H_{m/2} - H_{\sqrt{m}})$$

$$\sim \frac{1}{4}\ln m + \frac{1}{80}\ln m = O(\ln m). \qquad \square$$

LEMMA 3.3. Let e_j denote the number of multiple edges of multiplicity j in the graph. Then

$$\lim_{m\to\infty} \sum_{j=3}^{m} E(e_j) = 0.$$

Proof. The expected number of multiple edges of multiplicity j is $E(e_j) = N \cdot p_j$, where $N = m^2$ is the number of possible places an edge can be inserted in the graph and p_j is the probability that j edges are inserted in a given place. Since p_j is determined by the binomial distribution

$$p_j = \binom{m}{j}\left(\frac{1}{N}\right)^j\left(1 - \frac{1}{N}\right)^{m-j}$$

we get

$$E(e_j) = N\binom{m}{j}\left(\frac{1}{N}\right)^j\left(1-\frac{1}{N}\right)^{m-j}$$
$$\leqslant \binom{m}{j}\frac{1}{N^{j-1}} \leqslant \frac{m^j}{j!}\frac{1}{m^{2(j-1)}} = \frac{1}{j!m^{j-2}}$$

what gives the lemma. □

Consider the time complexity of the whole ordering step that comprises finding the fundamental cycles and building the tower. By Lemma 3.3 we can omit the multiple edges in our analysis. The cost of finding the fundamental cycles of G_1 is proportional to the total length of these cycles. By Lemma 3.1 this length cannot exceed $O(\sqrt{m})$. While placing the edges of the fundamental cycles in the tower, the following operations are executed: (i) selecting the shortest cycle in the heap; (ii) finding a path in $\mathcal{F}$ between the end vertices of a non-canonical edge, and making the path list for it; (iii) restoring the heap. Operation (i) takes time $O(v \log v)$, that by Lemma 3.2 is $O(\ln m \log(\ln m))$. Finding the paths for all non-canonical edges in operation (ii) requires at most $O(m_1 v) = O(\sqrt{m} \ln m)$ steps, whereas making the path lists is done in $O(m_1) = O(\sqrt{m})$ steps. The cost of operation (iii) is at most $O(m_1 \log v) = O(\sqrt{m} \log(\ln m))$. All these costs imply that the time complexity of the ordering step is less than $O(m)$.

3.3. The Searching Step

In the searching step, the following combinatorial problem is solved: find $U(w_i) \in [0 .. m-1]$, $i = 1, 2, \ldots, k$, where k is the height of the tower, such that values $h(w_i) = (h_0(w_i) + U(w_i)) \bmod m$ for canonical words $w_i \in Y_k$, and $h(w_j) = (h_0(w_j) + \Sigma_{p \in path(w_j)}(-1)^p U(y_p)) \bmod m$ for non-canonical words $w_j \in W - W_a - Y_k$ are all distinct, i.e. for any w_1 and $w_2 \in W - W_a$, $h(w_1) \neq h(w_2)$. To find $U(w_i)$s we perform the exhaustive search at every level X_i of the tower. We start with $U(w_i) = 0$ for each canonical word w_i, i.e. we attempt to locate it at the position $h_0(w_i)$ in the hash table. Note that since values h_0 are random, we begin to locate all the canonical words of the tower in a random probe of places. Once the hash value for the canonical word w_i on a given level of the tower is found, the value of $U(w_i)$ is known. It enables us to compute the hash values for the non-canonical words at the level. Following [17], we shall call the set of the hash values of words at a given level X_i of the tower as a *pattern* of size $s = |X_i|$. Clearly, if all places defined by the pattern are not occupied, the task on a given level is done and the next level is processed. Otherwise, the pattern is moved up the table modulo m until the place where it fits is found. Except for the first level of the tower, this search is conducted for the hash table that is partially filled. Thus, it may happen that no place for the pattern is found. In such a case the searching step backtracks to earlier levels, assigns different hash values for words on these levels, and then again recomputes the hash values for successive levels (Figure 2).

The exhaustive search applied here has a potential worst-case time complexity exponential in the number of words to be placed in the hash table. However, if this number is small as compared with the table size, the search is carried out in the table that is mostly empty and can be done in linear time.

In our case, the words to be placed in the hash table during the search correspond to multiple edges and the edges of fundamental cycles in graph G_1. By Lemmas 3.3 and 3.1, as $m \to \infty$ the number of multiple edges goes to 0, and the number of edges on fundamental cycles $m_1 = O(\sqrt{m})$. Among the latter, $m_1 - v$ edges are canonical. They constitute patterns of size 1 and are placed independently of each other at the positions $h_0(w)$. The other v edges are non-canonical (dependent) and form patterns of size $s > 1$. Assuming that i random elements of the hash table are occupied, the probability of successfully placing in one probe a pattern of size $s > 1$ is:

$$p_s = \frac{\binom{m-i}{s}s!}{m^s} = \frac{(m-i)(m-i-1)\cdots(m-i-s-1)}{m^s}.$$

Since $i < m_1$, $s \leqslant v$, $m_1 \sim \sqrt{m}$ and $v \sim \ln m$, we have $\lim_{m\to\infty} p_s = 1$. Thus, one can neglect the existence of non-canonical edges. Treating these edges as canonical, the search can be approximated to a task of placing m_1 words (edges) in the hash table of size m using $h_0(w)$ as the primary hash function and the linear probing to resolve collisions. The expected number of probes to place a single word in the table containing i words is [26]:

```
-- cardW[0 .. k] table contains the cardinalities
-- of subsets W0, W1, ..., Wk of the tower
for i in 1 .. k do -- mark "starting point" at each level
  h[cardW[i - 1]] := virgin;
end for;
for i in 0 .. m - 1 do -- free all places in the hash table
  taken[i] := false;
end for;
i := 1;
while i in 1 .. k do
  j := cardW[i - 1]; -- the first (canonical) word on level i
  if h[j] = virgin then
    h[j] := h0[j]; -- find place for canonical word
    while taken[h[j]] ≠ false do
      h[j] := (h[j] + 1) mod m;
    end while;
  else
    taken[h[j]] := false; -- free the previous place, find the next one
    repeat
      h[j] := (h[j] + 1) mod m;
    until taken[h[j]] = false cor h[j] = h0[j];
    if h[j] = h0[j] then
      h[j] := virgin; -- no place found
    end if;
  end if;
```

FIGURE 2. Searching for the hash values (**cor** and **cand** denote the conditional **or** and **and** operations, respectively).

$$C_i' \approx \frac{1}{2}\left[1 + \left(\frac{1}{1-\alpha_i}\right)^2\right]$$

where $\alpha_i = i/m$ is the load factor of the hash table which varies between 0 and $(m_1 - 1)/m = O(1/\sqrt{m})$. The total number of probes to place all m_1 words is then

$$\sum_{i=0}^{m_1-1} C_i' \leqslant \frac{m_1}{2}\left[1 + \left(\frac{1}{1-1/\sqrt{m}}\right)^2\right] \sim m_1 = O(\sqrt{m}).$$

Thus, as $m \to \infty$ each word is placed in the hash table in a constant time, and the search is done in time $O(\sqrt{m})$.

Our practical experiments showed that m_1 is bounded by $\frac{1}{4}\sqrt{m}$ (Figure 4), what is two times less than our theoretical result (Lemma 3.1). Furthermore, $v < \frac{1}{4}\ln m$, and in fact for $m = 1000 \ldots 100\,000$, v can be treated as a small constant (Figure 5). For the tables of size $m = 100\,000$ the average number of words to be placed in the table was 53.66, so the tables were almost empty

```
  if h[j] = virgin then -- no place for canonical word, backtrack
    i := i - 1;
    free places h[cardW[i - 1] + 1] .. h[cardW[i] - 1];
  else
    U[i] := (h[j] - h0[j]) mod m;
    taken[h[j]] := true;
    conflict := false;
    j := j + 1;
    while not conflict cand j < cardW[i] do
      h[j] := (h0[j] + Σ_{p∈path(j)} U[p.t] × p.sign) mod m;
      if taken[h[j]] = false then
        taken[h[j]] := true;
        j := j + 1;
      else
        conflict := true;
        free places h[cardW[i - 1] + 1] .. h[j - 1];
      end if;
    end while;
    if not conflict then
      i := i + 1; -- proceed to the next level
    end if;
  end if;
end while;
compute h values for free words;
```

FIGURE 3. Searching for the hash values (continued).

indeed when the search for these words was executed. The column *e_src* of Table 4 contains the execution time of the exhaustive search. One can see that the search time is negligible as compared to the total execution time of the algorithm.

Once the hash values for m_1 words are determined, the hash values for free words are computed (see Figure 3). It is accomplished by a single pass through the hash table and locating successive words in empty places. Clearly, it takes $O(m)$ time. Summing up, the time complexity of the whole searching step is $O(m)$.

Lastly, using the values $U(w_i)$ found during the search, the table g is calculated. For this purpose the $O(n)$ algorithm FINDg given in [32] is applied (the algorithm in [32] has $O(n^2)$ time complexity, but a simple change of the graph representation yields a linear time algorithm).

We complete this section with an example illustrating the work of the new algorithm.

Example. Consider the set of the month names, abbreviated to the first three characters $W = \{jan, feb, \ldots, dec\}$, $|W| = m = 12$. Thus, the size of array g is $|g| = 2r = 2m = 24$. Suppose that in the mapping step the randomly generated contents of tables T_0, T_1 and T_2 are as shown in Table 1, with the values for unused letters omitted.

The triples computed for each word are as follows: $h_0(jan) = (T_0[1, 'j'] + T_0[2, 'a'] + T_0[3, 'n']) \bmod m = (4 + 3 + 1) \bmod 12 = 8$ (the row indices are calculated from the formula $i_m = ((i + |w| - 1) \bmod |w|_{max}) + 1$, cf. Section 3.1), $h_1(jan) = (T_1[1, 'j'] + T_1[2, 'a'] + T_1[3, 'n']) \bmod r = (6 + 7 + 5) \bmod 12 = 6$, $h_2(jan) = ((T_2[1,'j'] + T_2[2,'a'] + T_2[3,'n']) \bmod r) + r = ((8 + 11 + 9) \bmod 12) + 12 = 16$, $h_0(feb) = (11 + 4 + 5) \bmod 12 = 8$, $h_1(feb) = (11 + 7 + 1) \bmod 12 = 7$, ..., etc. The values of the mapping functions for all words are shown in Table 2.

Since there are no identical triples, the mapping step is completed. The dependency graph $G_0 = (R, E_0)$ generated in the mapping step is shown in Figure 6.

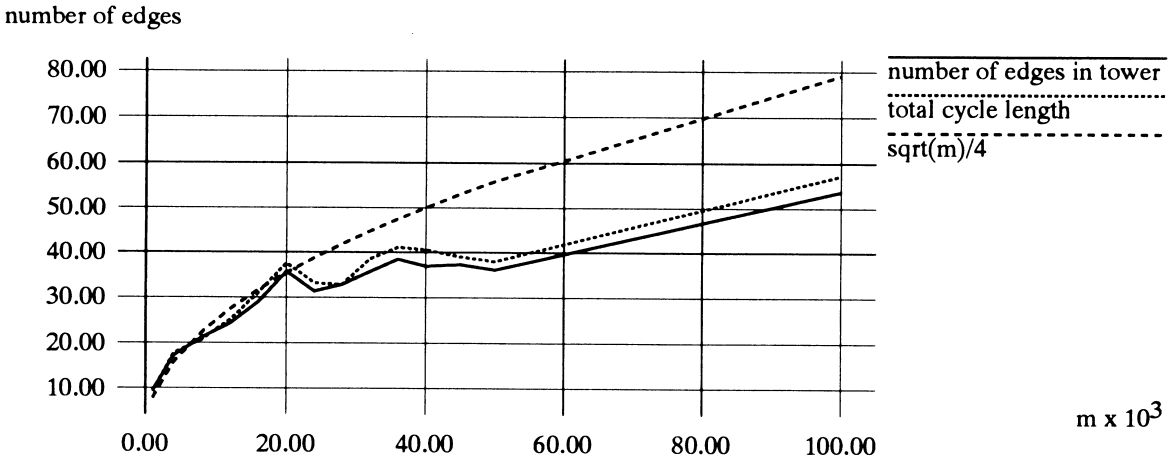

FIGURE 4. The average number of edges in the tower and the average total length of fundamental cycles, m_1.

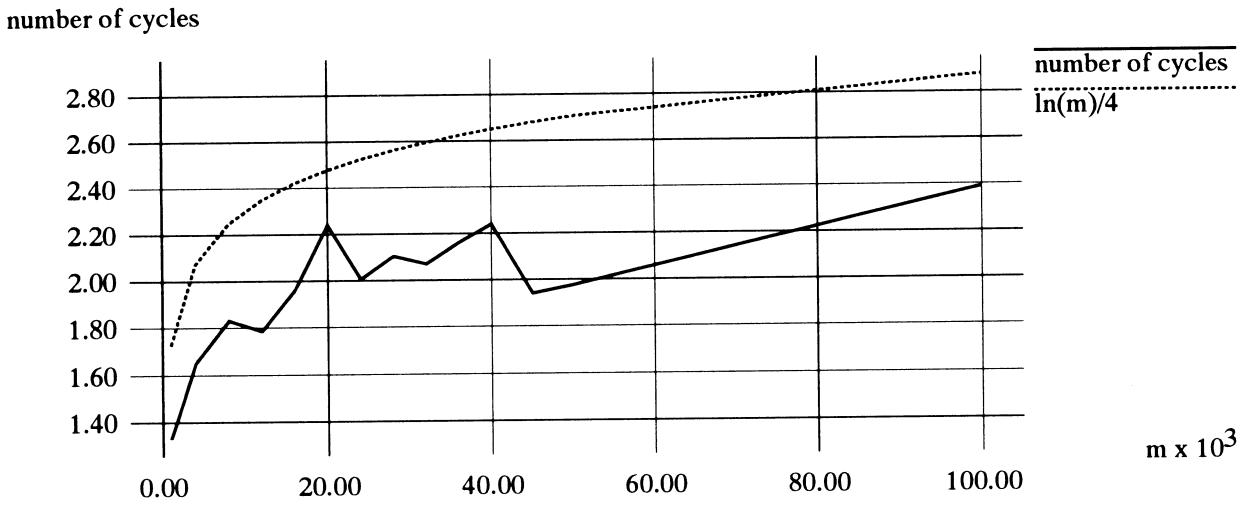

FIGURE 5. The average number of fundamental cycles, v.

TABLE 1. The mapping tables

T_0:

a	b	c	d	e	f	g	j	l	m	n	o	p	r	s	t	u	v	y
3			1		11		4		5	5	8			8				
3		2		4							11	6				11		
	5	10				7		10		1		5	7		2		10	6

T_1:

a	b	c	d	e	f	g	j	l	m	n	o	p	r	s	t	u	v	y
4			7		11		6		9	8	10			3				
7		3		7							11	9				1		
	1	2				10		11		5		5	8		3		10	1

T_2:

a	b	c	d	e	f	g	j	l	m	n	o	p	r	s	t	u	v	y
6			11		1		8		8	3	8			2				
11		2		11							3	8				5		
	3	10				9		9		9		11	1		6		1	11

TABLE 2. The values of the mapping functions

	jan	*feb*	*mar*	*apr*	*may*	*jun*	*jul*	*aug*	*sep*	*oct*	*nov*	*dec*
h_0	8	8	3	4	2	4	1	9	5	0	2	3
h_1	6	7	0	9	5	0	6	3	3	4	5	4
h_2	16	15	20	15	18	22	22	20	12	16	19	20

It contains only one cycle (6, 16, 4, 20, 0, 22), i.e. the subgraph $G=(R, E)$ is unicyclic. This cycle is also the sole fundamental cycle of G. The words associated with the edges in $E_0 - E$ constitute the set of *free* words, which in our case is $W_a = \{feb, apr, may, aug, sep, nov\}$.

In the ordering step the single cycle of G is put onto the heap. Then repeatedly an edge is removed from it and the length of the cycle is decreased. Obviously, this does not rearrange anything in the heap. Any edge of the cycle may become the single noncanonical edge. We choose edge (6, 16) as the non-canonical. Thus, the tower of subsets has the following form: $X_0 = \emptyset$, $X_1 = \{jun\}$, $X_2 = \{mar\}$, $X_3 = \{jul\}$, $X_4 = \{dec\}$, $X_5 = \{oct, jan\}$. The path for *jan* is $\langle +5-4+3+2-1\rangle$.

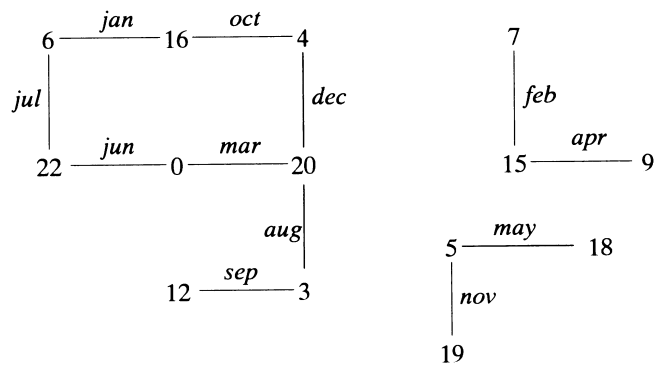

FIGURE 6. The dependency graph G_0.

In the searching step, each word w will be placed first at the position $h_0(w)$. If this position is occupied, a word is moved to some other, yet unused place of the hash table. So, *jun* is placed at address 4, *mar* at address 3 and *jul* at address 1. The word *dec* cannot be placed at the position $h_0(dec) = 3$, as it is occupied by the word *mar*. Hence it is moved to position 5. The next word, *oct*, is put at location 0. Now, using the path, we compute the place determined by the former keys for *jan*: $h(jan) = 8 + (0 - 2 + 0 - 0 + 0) = 6$. This address is free and the whole cycle has been placed successfully in the hash table in a single attempt. The remaining words from the subset W_a are placed in the unused entries of the hash table. The final positions of the words are {6, 7, 3, 2, 9, 4, 1, 10, 11, 0, 8, 5}. Once the places for the words are established, the content of array g is computed (Table 3).

Now, to calculate the hash table address for, say, *may*, we compute $h_0(may) = (5 + 3 + 6) \bmod 12 = 2$, $h_1(may) = (9 + 7 + 1) \bmod 12 = 5$, and $h_2(may) = ((8 + 11 + 11) \bmod 12) + 12 = 18$. Then the hash table address of *may* is $(h_0(may) + g(5) + g(18)) \bmod 12 = (2 + 0 + 7) \bmod 12 = 9$. □

4. EXPERIMENTAL RESULTS

The new algorithm was implemented in the C language. All experiments were carried out on Sun SPARC station 2, running under the SunOS™ operating system. The results are summarized in Table 4. An entry in the table produced for the algorithm was generated as follows: for each specified m (number of words) 200 random sets of words were selected. The table entries represent the averages over these 200 trials. Words were chosen from 24 692 words in a dictionary. The dictionary was obtained by removing from the standard Unix dictionary all words shorter than three characters, longer than 18 characters or containing characters other than letters. For each experiment the words were selected using shuffling [25]. For $m > 24\,692$, artificial sets of random words were generated. The values of m, *map*, *ord*, *e_src*, *src* and *total* are the number of words, time for the mapping step, time for the ordering step, time for the exhaustive search, time for the whole searching 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. The total execution time grows approximately linearly with m. A comparison with the timing results given in [17] reveals that our algorithm is much faster than that given there. For example, their algorithm took 190.93 s to generate a minimal perfect hash function for 131 072 keys on a Sequent machine.

TABLE 4. Results of experiments

m	*map*	*ord*	*e_src*	*src*	*total*
1000	0.058	0.238	0.000	0.023	0.318
2000	0.098	0.282	0.000	0.047	0.427
3000	0.142	0.322	0.000	0.070	0.534
4000	0.183	0.366	0.000	0.091	0.641
5000	0.224	0.409	0.000	0.115	0.747
6000	0.264	0.452	0.000	0.140	0.855
7000	0.305	0.495	0.000	0.160	0.960
8000	0.347	0.534	0.000	0.186	1.067
9000	0.392	0.577	0.000	0.209	1.178
10000	0.439	0.619	0.000	0.232	1.290
12000	0.539	0.722	0.000	0.288	1.549
16000	0.697	0.867	0.000	0.374	1.938
20000	0.870	1.027	0.000	0.470	2.367
24000	1.043	1.187	0.000	0.567	2.797
28000	1.365	1.374	0.001	0.666	3.405
32000	1.556	1.544	0.001	0.766	3.866
36000	1.748	1.696	0.001	0.866	4.311
40000	1.951	1.866	0.001	0.966	4.783
45000	2.207	2.091	0.000	1.089	5.387
50000	2.437	2.260	0.001	1.200	5.897
100000	4.486	4.018	0.000	2.405	10.909

In the implementation of the algorithm we used an edge-oriented representation of graphs [14]. 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.

5. CONCLUSIONS

The new algorithm for finding minimal perfect hash functions has been presented. It is based on generation of random bipartite graphs of n vertices and m edges. By setting $n = 2m$ we obtain sparse graphs, what allows to find the hash function in linear time. The hash function generated is represented by using $2m + O(1)$ memory words of $\log m$ bits each. The empirical observations show that the algorithm is very fast.

It was shown in [13] that by setting $n = (2 + \varepsilon)m$, $\varepsilon > 0$, the algorithm we have presented can be converted into a probabilistic one that does not execute the exhaustive search. However, since n determines the size of the description of the MPHF we need more storage to represent the function generated by this algorithm.

TABLE 3. Array g

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
g(i)	0	0	0	1	2	0	0	0	0	11	0	0	5	0	0	11	10	0	7	6	0	0	0	0

Acknowledgements

Z.J.C. was supported by the faculty grant BW-746/RaU2/92 of the Silesia University of Technology.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA (1974).
[2] C. Berge, *The Theory of Graphs and its Applications*. John Wiley, New York (1962).
[3] B. Bollobas, *Random Graphs*. Academic Press, New York (1985).
[4] M. D. Brian and A. L. Tharp, Near-perfect hashing of large word sets. *Software—Practice and Experience*, **19**, pp. 967–978 (1990).
[5] J. Cercone, J. Boates and M. Krause, An interactive system for finding perfect hash functions. *IEEE Software*, **2**, pp. 38–53 (1985).
[6] C. C. Chang, The study of an ordered minimal perfect hashing scheme. *Commun. ACM*, **27**, pp. 384–387 (1984).
[7] C. C. Chang and R. C. T. Lee, A letter-oriented minimal perfect hashing scheme. *The Computer Journal*, **2**, pp. 277–281 (1986).
[8] R. J. Cichelli, Minimal perfect hash functions made simple. *Commun. ACM*, **23**, pp. 17–19 (1980).
[9] C. R. Cook and R. R. Oldehoeft, A letter oriented minimal perfect hashing function, *Sigplan Notices*, **17**, pp. 18–27 (1982).
[10] Z. J. Czech, G. Havas and B. S. Majewski, *An Optimal Algorithm for Generating Minimal Perfect Hash Functions*. TR-217, The University of Queensland, Key Centre for Software Technology, Queensland (1991).
[11] Z. J. Czech, G. Havas and B. S. Majewski, *Practically Optimal Algorithms for Minimal Perfect Hashing*. TR-222, The University of Queensland, Key Centre for Software Technology, Queensland (1992).
[12] Z. J. Czech and B. S. Majewski, Generating a minimal perfect hash function in $O(M^2)$ time. *Archiwum Informatyki Teoretycznej i Stosowanej*, **4**, pp. 3–20 (1992).
[13] Z. J. Czech, G. Havas and B. S. Majewski, An optimal algorithm for generating minimal perfect hash functions. *Information Process. Lett.*, **43**, pp. 257–264 (1992).
[14] J. Ebert, A versatile data structure for edge-oriented graph algorithms. *Commun. ACM*, **30**, pp. 513–519 (1987).
[15] P. Flajolet, D. E. Knuth and B. Pittel, The first cycles in an evolving graph. *Discrete Mathematics*, **75**, pp. 167–215 (1989).
[16] E. Fox, L. Heath and Q. F. Chen, *An $O(n \log n)$ Algorithm for Finding Minimal Perfect Hash Functions*. TR-89-10, Virginia Politechnic Institute and State University, Blacksburg, VA (1989).
[17] E. A. Fox, L. S. Heath, Q. Chen and A. M. Daoud, Practical minimal perfect hash functions for large databases. *Commun. ACM*, **35**, pp. 105–121 (1992).
[18] E. Fox, Q. F. Chen and L. Heath, *LEND and Faster Algorithms for Constructing Minimal Perfect Hash Functions*. TR-92-2, Virginia Polytechnic Institute and State University, Blacksburg, VA (1992).
[19] M. L. Fredman, J. Komlós and E. Szemerédi, Storing a sparse table with $O(1)$ worst-case access time. *J. ACM*, **31**, pp. 538–544 (1984).
[20] G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures*. Addison-Wesley, Reading, MA (1991).
[21] M. Gori and G. Soda, An algebraic approach to Cichelli's perfect hashing. *BIT*, **29**, pp. 2–13 (1989).
[22] G. Haggard and K. Karplus, Finding minimal perfect hash functions. *ACM SIGSCE Bull.*, **18**, pp. 191–193 (1986).
[23] G. Jaeschke, Reciprocal hashing: a method for generating minimal perfect hashing functions. *Commun. ACM*, **24**, pp. 829–833 (1981).
[24] G. Jaeschke and G. Osterburg, On Cichelli's minimal perfect hash functions method. *Commun. ACM*, **23**, pp. 728–729 (1980).
[25] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 2nd edn. Addison-Wesley, Reading, MA (1981).
[26] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Searching and Sorting*. Addison-Wesley, Reading, MA (1973).
[27] T. G. Lewis and C. R. Cook, Hashing for dynamic and static internal tables. *IEEE Computer*, **21**, pp. 45–56 (1988).
[28] B. S. Majewski, N. C. Wormald, Z. J. Czech and G. Havas, *A Family of Generators of Minimal Perfect Hash Functions*. TR92-16, DIMACS at Rutgers University, Piscataway, NJ (1992).
[29] K. Mehlhorn, On the program size of perfect and universal hash functions. *Proc. of the 23rd IEEE Symposium on Foundations of Computer Science*, pp. 170–175, Chicago (1982).
[30] E. M. Palmer, *Graphical Evolution: An Introduction to the Theory of Random Graphs*. New York, John Wiley & Sons (1985).
[31] T. J. Sager, *A New Method for Generating Minimal Perfect Hash Functions*. Tech. Rep. CSc-84-15, University of Missouri-Rolla, Rolla, MO (1984).
[32] T. J. Sager, A polynomial time generator for minimal perfect hash functions. *Commun. ACM*, **28**, pp. 523–532 (1985).
[33] R. Sprugnoli, Perfect hashing functions: a single probe retrieving method for static sets. *Commun. ACM*, **20**, pp. 841–850 (1977).
[34] V. G. Winters, Minimal perfect hashing in polynomial time. *BIT*, **30**, pp. 235–244 (1990).