Programming Techniques and Data Structures

Ellis Horowitz
Editor

A Polynomial Time Generator for Minimal Perfect Hash Functions

THOMAS J. SAGER

ABSTRACT: A perfect hash function PHF is an injection F from a set W of M objects into the set consisting of the first N nonnegative integers where N >= M. If N = M, then F is a minimal perfect hash function, MPHF. PHFs are useful for the compact storage and fast retrieval of frequently used objects such as reserved words in a programming language or commonly employed words in a natural language.

The mincycle algorithm for finding PHFs executes with an expected time complexity that is polynomial in M and has been used successfully on sets of cardinality up to 512. Given three pseudorandom functions h_0, h_1, and h_2, the mincycle algorithm searches for a function g such that $F(w) = (h_0(w) + g \circ h_1(w) + g \circ h_2(w)) \bmod N$ is a PHF.

1. INTRODUCTION

A *perfect hash function* PHF is an injection F from a set W of M objects into the set consisting of the first N consecutive nonnegative integers where $N >= M$. If $N = M$, then we say that F is a *minimal perfect hash function*, MPHF. Minimal perfect hash functions are useful for compact storage and fast retrieval of frequently employed sets of objects such as reserved words in a programming language or commonly used words in a natural language.

This article presents the mincycle algorithm for finding minimal perfect hash functions. Unlike the algorithms for generating perfect hash functions presented by Sprungnoli [8], Cichelli [2], and Jaeschke [6] whose expected execution time is exponential in $M = \text{card}(W)$, the mincycle algorithm's expected execution time is polynomial in M. It is, therefore, practical to use the mincycle algorithm to find PHFs for considerably larger sets than those on which previously known algorithms for this purpose are practical. We have used the mincycle algorithm successfully on sets of cardinality up to 512.

More recently, Chang [1] has discovered an algorithm which appears to have time complexity $O(M^2 \log(M))$.[1] This method requires the existence of an injection p from W into the set of prime integers. Chang, however, gives no general method for finding such injections. Given $W = \{w_1, w_2, \ldots, w_M\}$ and such an injection p, Chang's algorithm finds an integer C such that $\forall i \mid 1 <= i <= M$, $i - 1 = C \bmod p(w_i)$. Unfortunately, the number of bits required to represent C appears to be proportional to $M \log(M)$. Given $p(w_i)$ = the ith prime number and $M = 64$, the approximate value of C would be $1.92 \star 10^{124}$ and the binary representation of C would require 413 bits. Assuming a 32-bit word, each application of the MPHF generated by Chang's algorithm would require fetching 13 memory words and executing 13 divide operations. In contrast, the MPHF for the same set generated by the mincycle algorithm would require fetching only two memory words and executing no divide instructions.

The mincycle algorithm and Cichelli's algorithm can be viewed as variations on the same basic theme. In fact, the mincycle algorithm was discovered while searching for optimizations to Cichelli's algorithm.

In Section 2, terminology is discussed and in Section 3, the mincycle algorithm is introduced. Section 4 con-

[1] We consider that the number of bits B required to represent the product of the first M primes is proportional to $M \log(M)$ and that the time required to do arithmetic operations on B bit integers is proportional to B.

tains a brief discussion and analysis of the mincycle algorithm. In Section 5, Cichelli's algorithm is presented and the similarities and differences between Cichelli's algorithm and the mincycle algorithm are discussed. Finally, in Section 6, a moral is presented.

2. TERMINOLOGY

In this section the terminology used in subsequent sections of this article is introduced. It is assumed that the reader is familiar with the basics of graph theory. An excellent introduction to graph theory can be found in Harary [4].

I is the set of all integers and $[i\,..\,j] = \{x \in I \mid x >= i$ and $x <= j\}$. We use angular brackets, $\langle\ \rangle$, to denote ordered pairs. card(X) is the cardinality of the set X. We use $(\Sigma f(x), x \in X)$ or $(\Sigma f(i), i := a$ to b by $c)$ to denote summations instead of the more commonly used forms. $f: A \rightarrow B$ means that f is a function from the domain A into the set B.

A tower of subsets of W is a sequence of subsets of W: $W_0, W_1, \ldots, W_k$ such that $W_0 = \emptyset$, $W_k = W$ and $\forall i \in [1\,..\,k]$, $W_{i-1} \subseteq W_i$. If the set inclusion is always proper, then we call the tower a monotonic tower. k is the height of the tower.

A multiset is, loosely speaking, a set which can contain a given element zero or more times. More precisely, a multiset A is denoted by its characteristic function ϕ from a universal domain into the nonnegative integers. If ϕ is the characteristic function of a multiset A, $a \in A$, and $\phi(a) = i$, then we say that i is the multiplicity of a in A.

A graph H is an ordered pair $\langle V, E\rangle$ where V is called the vertex set and E is called the edge set. We use the term graph synonymously with the term undirected graph with no loops. Thus, V is a finite set of objects and E is a multiset where each member of E is a subset of V of cardinality exactly 2.

A path p over the graph $H = \langle V, E\rangle$ is a sequence of edges $e_0, e_1, \ldots, e_t$ such that $\forall i \in [0\,..\,t]$, e_i has positive multiplicity and $\exists$ a sequence, $q = v_0, v_1, \ldots, v_{t+1}$ over V, $\forall i \in [1\,..\,t]$, $v_i \in e_i \cap e_{i-1}$, $v_0 \in e_0$, $v_{t+1} \in e_t$ and each member of the sequence q is distinct except possibly $v_0 = v_{t+1}$. If $v_0 = v_{t+1}$, then p is called a cycle. $t + 1$ is the length of p. We use the term path (cycle) synonymously with the term elementary path (cycle).

Sometimes we wish to discuss how many cycles of a given length m an edge $e = \{v, v'\}$ lies on. We define this concept as follows: Let ϕ be the characteristic function of E. If $\phi(e) = 0$, then e is on no cycles of any length. Let $\phi(e) > 0$. Then e is on no cycles of length less than 2 and exactly $\phi(e) - 1$ cycles of length 2. Now let $m > 2$ and n be the cardinality of the set of all distinct paths of length $m - 1$ from v to v' over H. Then e is on n cycles of length m. For completeness, we say that e is on exactly one cycle of length infinity. If there are no cycles of finite length over H, then we say that H is cycle-free.

3. THE MINCYCLE ALGORITHM

We break the problem of finding a PHF down into three parts. In Part 1, we choose certain parameters to the PHF, F. In Part 2, we choose a monotonic tower of subsets of W: $\phi = W_0 \subset W_1 \subset \cdots \subset W_{k-1} \subset W_k = W$. In Part 3, we perform an exhaustive search for a PHF. The ordering chosen in Part 2 is used to minimize the amount of work necessary to perform the exhaustive search. More precisely:

Part 1

We must choose the following parameters:

R_1 and R_2	– two disjoint finite sets of elements and
h_0: $W \rightarrow I$,	(where I is the set of integers)
h_1: $W \rightarrow R_1$,	and
h_2: $W \rightarrow R_2$	– three quickly computable pseudorandom functions.

Letting $R = R_1 \cup R_2$, our problem can now be restated as:

Given W, N, R_1, R_2, h_0, h_1, and h_2 satisfying all the above constraints, find a function $g: R \rightarrow [0\,..\,N-1]$ such that $F(w) = (h_0(w) + g \circ h_1(w) + g \circ h_2(w)) \bmod N$ is a PHF.

In general, unless we have felt a compelling reason to do otherwise, we have chosen:

$R_1 \quad = [0\,..\,r-1]$
$R_2 \quad = [r\,..\,2r-1]$
$h_0(w) = (\text{length}(w) + (\Sigma\ \text{ord}(w[i]), i := 1$ to length(w) by 3))
$h_1(w) = (\Sigma\ \text{ord}(w[i]), i := 1$ to length(w) by 2) mod r and
$h_2(w) = (\Sigma\ \text{ord}(w[i]), i := 2$ to length (w) by 2) mod $r + r$,

where:

1. r = smallest power of 2 greater than card(W)/3,
2. each $w \in W$ is considered as the sequence of characters $w[1], w[2], \ldots, w[\text{length}(w)]$, and
3. ord is the function which maps each character onto its representation as a binary integer.

The function ord is, of course, system dependent. The above choices seem to work well in most cases although they tend to be somewhat conservative. An example of a case where they did not work well is:

W = the set of all predeclared identifiers in the Pascal language,
N = card(W) = 40, and
ord uses the EBCDIC character code.

There we found that h_1 and h_2 both agreed on the two predeclared identifiers "ORD" and "READ", and $h_0(\text{"ORD"}) \equiv h_0(\text{"READ"})$ mod 40. Therefore, no PHF of the type searched for could exist. However, substituting

$$h_0(w) = (\text{length}(w) + (\Sigma\ \text{ord}(w[i]),\ i := 1 \text{ to length}(w) \text{ by } 3)) \bmod 64$$

we were able to find a MPHF for this set.

Part 2

Having chosen the parameters R_1, R_2, h_0, h_1, and h_2 in Part 1, we now choose a monotonic tower of subsets of W: $\phi = W_0 \subset W_1 \subset \cdots \subset W_{k-1} \subset W_k = W$.

Informally, in Part 3 at each step we will, for some $i \in [1 \,..\, k]$, attempt to extend the desired function F from the domain W_{i-1} to the domain W_i. We would, therefore, like the W_is to be as large as possible and the quantity of work necessary to extend F as small as possible. In this regard, Part 2 of the mincycle algorithm is not necessarily optimal, but as will be seen in Section 4, it can be expected to perform "reasonably well" in most cases. In practice, we have not found a case yet in which Part 2 of the mincycle algorithm did not behave adequately.

Basically, in Part 2, W_i is chosen from W_{i-1} so that $\forall x$ and $y \in W_i - W_{i-1}$, $F(y)$ is *uniquely determined* by F restricted to $W_{i-1} \cup \{x\}$ and W_i is at least as large as any subset of W with the above property.

We will make this notion of *uniquely determined* more precise in the following section. Now, consider the sequence of graphs: $H_0, H_1, \ldots, H_k$ where $\forall i \in [0 \,..\, k]$:

$H_i = \langle P_i, E_i \rangle$,

P_i = the partition of R generated by the smallest equivalence relation containing $\{\langle h_1(w), h_2(w) \rangle \mid w \in W_i\}$, and

E_i = the multiset of edges over P_i whose characteristic function is $\phi_i(\{p, q\}) = \text{card}(\{w \in W - W_i \mid \{h_1(w), h_2(w)\} \subseteq p \cup q\})$.

Now inductively, W_i (and hence H_i) is computed from H_{i-1} as follows:

Choose p and q such that $\{p, q\}$ is an edge of the graph H_{i-1} lying on a maximal number or minimal length cycles over H_{i-1}. Then let $X_i = \{w \in W - W_{i-1} \mid \{h_1(w), h_2(w)\} \subseteq p \cup q\}$ and let $W_i = W_{i-1} \cup X_i$.

The details of this step are given in Figures 1 and 2. We note that Algorithm BESTEDGE which finds an edge lying on a maximal number of minimal length cycles of the input graph has complexity $O(V^3)$ where V is the cardinality of the vertex set of the input graph. Since BESTEDGE is executed at most card(R) times and the size of the input vertex set is bounded above by card(R), it can be seen that the complexity of Part 2 of the mincycle algorithm is $O(\text{card}^4(R))$.

Part 3

$\forall i \in [1 \,..\, k]$, let $X_i = W_i - W_{i-1}$ and choose arbitrarily a canonical member x_i of X_i. In addition, let

$$Y_i = \{x_j \mid j \in [1 \,..\, i]\}.$$

We wish to find a function $g: R \rightarrow [0 \,..\, N-1]$ which makes $F: W \rightarrow [0 \,..\, N-1]$ a PHF where

$$F(w) = (h_0(w) + g \circ h_1(w) + g \circ h_2(w)) \bmod N.$$

```
algorithm BUILDTOWER; -- builds tower of subsets of W.
input
    M:        integer;    -- number of words in W.
    cardR:    integer;    -- number of vertices.
    mult:     array [0..cardR-1, 0..cardR-1] of integer;
              -- multiplicity matrix
    wrdlists:   array [0..cardR-1, 0..cardR-1] of listofwords;
              --  wrdlists[i,j] = list({w in W | {h1(w), h2(w)} = {i, j}})
output
    k:        integer;  -- height of tower.
    X:        array [1..cardW] of listofwords;  -- tower.
begin
    wordsleft := M;  k := 0;  maxvert := cardR-1;
    while wordsleft > 0 do
        k := k+1;
        BESTEDGE(cardR, maxvert, mult, v1, v2);
        X[k] := wrdlists[v1, v2];
        wordsleft := wordsleft - mult[v1, v2];
        forall i in [0..maxvert] do
            mult[v1, i] := mult[v1, i] + mult[v2, i];
            wrdlists[v1, i] := merglists( wrdlists[v1, i],
                                          wrdlists[v2, i] );
        endfor;
        copy row v1 of mult to column v1 of mult;
        copy row v1 of wrdlists to column v1 of wrdlists;
        copy row maxvert of mult to row v2 of mult;
        copy row maxvert of wrdlists to row v2 of wrdlists;
        copy row maxvert of mult to column v2 of mult;
        copy row maxvert of wrdlists to column v2 of wrdlists;
        maxvert := maxvert -1
    endwhile
end BUILDTOWER;
```

FIGURE 1. Mincycle Algorithm, Part 2, BUILDTOWER

```
algorithm BESTEDGE;   -- finds edge on max number of min length cycles.
input   --  assume input graph contains an edge of positive multiplicity.
    cardR:  integer;  -- order of multiplicity matrix.
    n:      integer;  -- number of vertices in graph - 1.
    mult:   array [0..cardR-1, 0..cardR-1] of integer;
        --  multiplicity matrix.
output
    a, b:   integer;  -- 2 vertices of edge.
internal
    paths: array [0..n, 0..n] of record
        minlnth:   integer; -- length of shortest path.
        nminl:     integer; -- number of shortest length paths.
        nminl1:    integer; -- number of paths of shortest length + 1.
        end paths;
begin
    limit := maxint / 2;  minmult := 0;
    forall x, y in [0..n] do
        with paths[x,y] do
            if mult[x,y] > minmult then
                minmult := mult[x,y]; a := x; b := y
            endif;
            if mult[x,y] > 0 then minlnth := 1; nminl := 1; nminl1 := 0
            else minlnth := limit; nminl := 0; nminl1 := 0;
            endif
        endwith
    endfor
    if minmult = 1 then      -- no cycle of length 2 exists
        forall x in [0..n] do
            forall y, z in [0..n] such that x, y and z are distinct do
                w := paths[y,x].minlnth + paths[x,z].minlnth;
                if w <= limit then
                   with paths[y,z] do
                      if w = minlnth + 1 then
                         nminl1 := nminl1 + 1; limit := w; even := false
                      elsif w = minlnth then
                         nminl := nminl + 1;
                         if w < limit then limit := w; even := true endif
                      elsif w = minlnth - 1 then
                         if w < limit then
                            nminl1 := nminl; limit := w + 1; even := false
                         endif
```

FIGURE 2. Mincycle Algorithm, Part 2, BESTEDGE

Now $\forall i \in [0 \,..\, k]$ and $w \in W_i$, let path(w) = $y_0, y_1, \ldots, y_t$ be the unique sequence over Y_i such that the sequence of edges over the vertex set R,

$$\{h_1(y_0), h_2(y_0)\}, \{h_1(y_1), h_2(y_1)\}, \ldots, \{h_1(y_t), h_2(y_t)\},$$

forms a path from $h_1(w)$ to $h_2(w)$. That such a unique sequence must always exist and that t must always be even is shown in the following section. Algorithms for finding such paths are well known.

Now, given an injection. F: $W_{i-1} \rightarrow [0 \,..\, N-1]$ where $i <= k$, we may attempt to extend F to the domain W_i by searching for a value $n \in [0 \,..\, N-1]$ with the following property: $\forall w \in X_i$, let $F(w) = (h_0(w) + (\Sigma(-1)^j U(y_j), j \in [0 \,..\, t]))$ mod N where path(w) = $y_0, y_1, \ldots, y_t$ and U: $Y_i \rightarrow [0 \,..\, N]$ is defined by the equation

$$U(x_j) = \begin{cases} (F(x_j) - h_0(x_j)) \bmod N & \text{if } 0 < j < i \\ n & \text{if } j = i. \end{cases}$$

Then F: $W_i \rightarrow [0 \,..\, N-1]$ is an injection.

The next step in the mincycle algorithm is to attempt to extend F incrementally from $W_0 = \emptyset$ to $W_k = W$ by the above method. Since, at each step, there may be no such values n or many such values n, it is necessary to use a backtracking algorithm to perform an exhaustive search. Thus, this step has a potential worst-case time

```
                        minlnth := w; nminl := 1
                    elsif w < minlnth - 1 then
                        minlnth := w; nminl := 1; nminl1 := 0;
                    endif
                endwith
            endif
        endfor
    endfor
endif;
if limit < maxint / 2   -- min length cycle is finite and > 2.
    maxncyc := 0;   --  maximum number of cycles.
    case even of
        true:
            forall x, y in [0..n] such that x<y and mult[x,y]=1 do
                ncyc := 0;
                forall z in [0..n] do
                    if (path[x,z].minlnth = limit) and
                            (path[y,z].minlnth = limit - 1) then
                        ncyc := ncyc + path[x,z].nminl - 1;
                    endif
                endfor;
                if ncyc > maxncyc then
                    maxncyc := ncyc; a := x; b := y
                endif
            endfor;
        false:
            forall x, y in [0..n] such that x<y and mult[x,y]=1 do
                ncyc := 0;
                forall z in [0..n] do
                    if (path[x,z].minlnth = limit - 1) and
                            (path[y,z].minlnth = limit - 1) then
                        ncyc := ncyc + 1;
                    endif
                endfor;
                if ncyc > maxncyc then
                    maxncyc := ncyc; a := x; b := y;
                endif
            endfor
    endcase
endif
end BESTEDGE;
```

FIGURE 2. (*Continued.*) Mincycle Algorithm, Part 2, BESTEDGE

complexity exponential in card(W). Fortunately, since we have chosen R to be approximately the same size as W in Part 1 and chosen the W_is carefully in Part 2, we find that, at least for sets of size 512 or less, we can, in practice, expect the execution time of Part 3 to be dominated by the execution time of Part 2. Therefore, the expected time complexity of the entire algorithm is merely card$^4(R)$. The details of this step are given in Algorithm SEARCH in Figure 3.

If we are successful in extending the function F to W, then all that remains is to find a function g such that $\forall w \in W$, $F(w) = (h_0(w) + g \circ h_1(w) + g \circ h_2(w)) \bmod N$. That at least one such function g exists is shown in the following section. Algorithm FINDg of Figure 4 will find such a function g. On the other hand, if we are unsuccessful in extending F to W, then we must return to Part 1 and choose a different set of parameters.

A short example of an application of the mincycle algorithm is given in Figure 5.

4. A BRIEF DISCUSSION OF THE MINCYCLE ALGORITHM

In this section we wish to informally analyze and justify the correctness of the mincycle algorithm. A more formal treatment of this subject has been given by the author in [7].

In Section 3, we used the concept *uniquely determined* loosely. We now make this concept more precise.

Definition: Let $w \in W$ and $X \subseteq W$. Then we say $F(w)$ is *uniquely determined* by the restriction of F to X or equivalently, w *depends on* X (since F does not figure in the definition) iff $\exists$ a function, $a: X \to [0\,..\,N-1]$, $\forall$ functions $g: R \to [0\,..\,N-1]$,

$$(g \circ h_1(w) + g \circ h_2(w))$$
$$\equiv (\Sigma\; a(x)(g \circ h_1(x) + g \circ h_2(x)),\ x \in X) \bmod N.$$

Now let $w \in W$ and $X \subseteq W$. Since the graph, $H = (R, \{\{h_1(x), h_2(x)\} | x \in X\})$ is bipartite, it can be shown that w *depends on* X iff $\exists$ a path over H from $h_1(w)$ to $h_2(w)$. Furthermore, this path is necessarily of odd length. Thus, if

$$\{h_1(w_0), h_2(w_0)\}, \{h_1(w_1), h_2(w_1)\}, \ldots, \{h_1(w_{2t}), h_2(w_{2t})\}$$

is a path over the graph H from $h_1(w)$ to $h_2(w)$, then $\forall$ functions g,

$$F(w) = (h_0(w) + g \circ h_1(w) + g \circ h_2(w)) \bmod N$$
$$= ((\Sigma(-1)^j(F(w_j) - h_0(w_j)),\ j := 0 \text{ to } 2t) + h_0(w)) \bmod N.$$

Also, from the construction of W_i from H_{i-1}, it can be shown that $\forall\ i \in [0\,..\,k]$ and $w \in W$, $w \in W_i$ iff $\exists$ a path,

```
algorithm SEARCH;   --  tries to find PHF from tower of subsets.
input
    N:      integer;    --  table size.
    M:      integer;    --  number of words.
    k:      integer;    --  height of tower.
    cardW:  array [0..k] of integer;    -- card of tower members.
    path:   array [0..M-1, 1..k] of {-1, 0, 1};
        --  jth row represents the path for jth word.
        --  path[j,t] = if (x[t] is an even member of path[w[j]]) then 1
        --  elsif (x[t] is an odd member of path[w[j]]) then -1 else 0.
    h0:     array [0..M-1] of integer;
output
    U:      array [1..k] of [0..N];
        --  U[i] represents g(h1(x[j])) + g(h2(x[j])).
    F:      array [0..M-1] of [0..N-1]  -- a PHF.
    success:    boolean;
begin
    for i := 1 to k do U[i] := N endfor;
    i := 1;
    while i in [1..k] do
        U[i] := (U[i] + 1) mod (N+1);
        if U[i] = N then
L2:         i := i-1
        else
            noconflict := true;
            j := cardW[i-1];
            while noconflict and j < cardW[i] do
                F[j] := ((Σ path[j,t] * U[t], t:= 1 to k) +
                        h0[j]) mod N;
                if (forall m in [0..j-1], F[m] <> F[j])
                then j := j + 1
                else noconflict := false
                endif
            endwhile;
            if noconflict then
L1:             i := i+1
            endif
        endif
    end_while;
    success := i > k
end SEARCH;
```

FIGURE 3. Mincycle Algorithm, Part 3, SEARCH

```
algorithm FINDg;
input
    N:      integer;    --  size of table.
    cardR:  integer;    --  size of vertex set.
    Umat:   array [0..cardR-1, 0..cardR-1] of [0..N];
        --  Umat[i, j] = if there exists t,
        --  {h1[x[t]], h2[x[t]]} = {i, j} then U[t] else N.
output
    g:      array [0..cardR-1] of [0..N-1]; --  desired function.
internal
    mark:   array [0..cardR-1] of boolean;
    procedure TRAVERSE(i: [0..cardR-1]);
    begin
        mark(i) := true;
        forall j in [0..cardR-1] do
            if Umat[i,j] < N and not mark[j] then
                g(j) := ( Umat[i,j] - g(i) ) mod N;
                TRAVERSE(j)
            endif
        endfor
    end TRAVERSE;
begin FINDg
    forall i in [0..cardR-1] do mark(i) := false endfor;
    forall i in [0..cardR-1] do
        if not mark(i) then
            g(i) := 0;
            TRAVERSE(i)
        endif
    endfor
end FINDg;
```

FIGURE 4. Mincycle Algorithm, Part 3, FINDg

Given: $W = \{AA, AAD, AB, BAA, BB, FA\}$.
Choose: $N = 6$, $r = 4$ and ASCII character code.
Results:

	AA	*AAD*	*AB*	*FA*	*BB*	*BAA*
h_0	67	68	67	72	68	69
h_1	1	1	1	2	2	3
h_2	5	5	6	5	6	5
F	1	2	3	0	4	5

i	1	2	3	4	5	6	7	0
X_i	$\{AA, AAD\}$	$\{AB\}$	$\{FA, BB\}$	$\{BAA\}$				
x_i	*AA*	*AB*	*FA*	*BAA*				
$U[i]$	0	2	0	2				
$g(i)$	0	0	2	0	0	2	0	0

where $U[i] = (g \circ h_1(x_i) + g \circ h_2(x_i)) \bmod N$.

FIGURE 5. Example of Application of Mincycle Algorithm

```
algorithm CSEARCH;  --  tries to find PHF from tower of subsets.
input
    N:      integer;     --  table size.
    M:      integer;     --  number of words.
    k:      integer;     --  height of tower.
    cardW:  array [0..k] of integer;     -- card of tower members.
    h0:     array [0..M-1] of integer;
    h1:     array [0..M-1] of [1..k];
        --  h1[j] = index of array U representing h1[w[j]].
    h2:     array [0..M-1] of [1..k];
        --  h2[j] = index of array U representing h2[w[j]].
    umax:   integer;     --  highest value of U[i]'s attempted.
output
    U:      array [1..k] of [0..umax];
        --  U[i] represents g(x[i]).
    F:      array [0..M-1] of integer    -- a PHF.
    Fmin:   integer;     --  minimum value of the PHF, F.
    success:    boolean;
begin
    for i := 1 to k do U[i] := umax endfor;
    i := 1;
    while i in [1..k] do
        U[i] := (U[i] + 1) mod (umax+1);
        if U[i] = umax then
L2:         i := i-1
        else
            noconflict := true;
            j := cardW[i-1];
            while noconflict and j < cardW[i] do
                F[j] := h0[j]) + U[h1[j]] + U[h2[j]];
                if (forall m in [0..j-1], 0 < abs(F[m] - F[j]) < N)
                then j := j + 1
                else noconflict := false
                endif
            endwhile;
            if noconflict then
L1:             i := i+1
            endif
        endif
    end_while;
    Fmin := min(F[j],j in [0..M-1]);
    success := i > k
end CSEARCH;
```

FIGURE 6. Part 3 of Cichelli's Algorithm

p, over the graph, $H_i' = (R, \{\{h_1(y), h_2(y)\} \mid y \in Y_i\})$ from $h_1(w)$ to $h_2(w)$ where k, H_i, W_i, and Y_i are as defined in Section 3. Thus, given Y_i, W_i is as large as possible subject to the constraint that $\forall\, w \in W_i$, *w depends on* Y_i, and furthermore, $\forall\, i \in [0 \,.\,.\, k]$, the graph, H_i' is cycle-free. Therefore, given the function F restricted to Y_k, there exists some function g such that $\forall\, y \in Y_k$, $F(y) = (h_0(y) + g \circ h_1(y) + g \circ h_2(y)) \bmod N$. Thus, should the Algorithm SEARCH succeed in finding an injection $F: W \rightarrow [0 \,.\,.\, N-1]$, this function g is necessarily consistent with F.

Now, it is also shown by the author in [7] that maximizing the cardinality of each member W_i of the tower of subsets in Part 2 of the mincycle algorithm subject to the constraint that $\forall\, w \in W_i$, *w depends on* Y_i, minimizes the expected execution time of Part 3 of the mincycle algorithm. Unfortunately, such a maximized tower of subsets may not exist and, even if it does, it is not clear that there exists an algorithm of "reasonable" time complexity for finding it. However, Part 2 of the mincycle algorithm does find a tower of minimal height. This tower appears to be "reasonably close" to the optimal and, in most cases, appears to behave "almost optimally."

Now, we noted in Section 3 that Part 2 of the mincycle algorithm has complexity $O(\text{card}^4(R))$ and that the worst-case time complexity of Part 3 is exponential in card(W). However, we also note that if card(R) is "large enough" with respect to card(W), we can expect to do very little backtracking in Part 3, (execution of statement L2: in Algorithm SEARCH), and Part 2 will therefore dominate Part 3.

We have determined experimentally that if card(R) = card(W) <= 512, then Part 2 can, indeed, be expected to dominate Part 3. We can also show formally that ∃ a function $f(n) \mid O(f(n)) = n^{3/2}$ and if card(R) = f(card(W)), then Part 2 can always be expected to dominate Part 3.[2] We do not know whether there exists a function $f'(n)$ such that $O(f'(n)) < n^{3/2}$ and if card(R) = f'(card(W)), then Part 2 can always be expected to dominate Part 3. However, from the above discussion, it can be seen that the mincycle algorithm has an expected time complexity no worse than proportional to $\text{card}^6(W)$.

5. CICHELLI'S ALGORITHM

The mincycle algorithm grew out of an attempt to optimize Cichelli's algorithm for generating PHFs. To compare the two algorithms, we present Cichelli's algorithm here in a form similar to the presentation of the mincycle algorithm in Section 3. Cichelli's original presentation of his algorithm [2] was in a considerably different form.

Like the mincycle algorithm, Cichelli's algorithm can be broken down into three parts. In Part 1, some general parameters are chosen. Cichelli allows for some slight variation in the choice of these parameters, but in general he chooses:

R: the set of all characters,
$h_0: W \to I$ is defined by the equation $h_0(w)$ = length(w),
$h_1: W \to R$ is defined by the equation $h_1(w)$ = first(w) and
$h_2: W \to R$ is defined by the equation $h_2(w)$ = last (w).

[2] I wish to acknowledge my thanks to Ralph W. Wilkerson of the Department of Computer Science at the University of Missouri-Rolla and Selden Trimble of the Department of Mathematics at the University of Missouri-Rolla for their help in proving this result.

In Part 2, Cichelli constructs a tower of subsets of W which is not necessarily monotonic by the following procedure:

1. $\forall\, c \in R$, let $p(c) = \text{card}(\{w \in W \mid \text{first}(w) = c\}) + \text{card}(\{w \in W \mid \text{last}(w) = c\})$.
2. Let $R' = \{c \in R \mid p(c) > 0\}$ and $k = \text{card}(R')$.
3. Rename the members of R': $x_1, x_2, \ldots, x_k$ so that if $1 <= i <= j <= k$ then $p(x_i) >= p(x_j)$.
4. $\forall\, i \in [0 \,..\, k]$, let $W_i = \{w \in W \mid \text{first}(w), \text{last}(w)\} \subseteq \{x_j \mid j \in [1 \,..\, i]\}\}$.

In Part 3, Algorithm CSEARCH in Figure 6 is executed. CSEARCH is practically identical to SEARCH except that:

1. The input, path, is replaced by the inputs $h1$ and $h2$.
2. The test for conflict has been changed to $0 < \text{abs}(F[i] - F[j]) < N$.
3. The maximum value allowed in the array U is arbitrary and has been replaced by the input, $u$max. $U[i]$ now represents $g(x_i)$.
4. CSEARCH actually searches for a function of the form $F(w) = h_0(w) + g \circ h_1(w) + g \circ h_2(w)$ whose range is [Fmin .. Fmin + N − 1] for some nonnegative integer, Fmin. Fmin is an output of CSEARCH.

In Figure 7, a short example of an application of Cichelli's algorithm is given.

We now list six similarities and differences between the mincycle algorithm and Cichelli's algorithm.

1. Cichelli's algorithm places a strict upper bound on card(R) whereas in the mincycle algorithm card(R) can be increased without bound.

2. The mincycle algorithm always minimizes the height of the chosen tower of subsets of W whereas Cichelli's algorithm does not necessarily do this.

Given: W = {AA, AAD, AB, BAA, BB, FA} and N = 6.
Results:

	AA	AB	BAA	BB	AAD	FA
h_0	2	2	3	2	3	2
h_1	A	A	B	B	A	F
h_2	A	B	A	B	D	A
F	2	4	5	6	3	7

i	1	2	3	4
x_i	A	B	D	F
$W_i - W_{i-1}$	{AA}	{AB, BAA, BB}	{AAD}	{FA}
$U[i] = g(x_i)$	0	2	0	5

FIGURE 7. Example of Application of Cichelli's Algorithm

Dataset	Size	—Mincycle— Time[2]	r	Code	Cichelli Time[2]
Pascal reserved words[1]	36	55	8	EBCDIC	183
Pascal predeclared identifiers	40	201	16	EBCDIC	3499[3]
Pascal reserved words and predeclared identifiers	76	326	16	EBCDIC	>50K[4]
ASCII control mnemonics	34	145	16	EBCDIC	1125
First 120 words in the Prologue to Chaucer's Canterbury Tales (truncated to nine characters)	120	36420	32	EBCDIC	>50K[4]
256 most commonly used words in the English language according to Dewey [3].	256	45058	128	ASCII	>50K[4]

[1] Includes OTHERWISE.
[2] Compiled under PASCAL 8000 compiler with T- option and run on an IBM 4341. Time is in milliseconds of CPU time as reported by the CLOCK function.
[3] ODD was omitted since h_0, h_1, and h_2 agree on ORD and ODD.
[4] Program had not terminated after 50,000 milliseconds. No attempt was made to increase the time limit.

FIGURE 8. Some Statistics for Minimal Perfect Hash Functions

3. Given equality of the heights of the towers chosen by the two algorithms, for fixed i the cardinality of the ith member of the tower of subsets chosen by the mincycle algorithm tends to be larger than the cardinality of the ith member of the tower of subsets chosen by Cichelli's algorithm.

4. Cichelli's algorithm allows very few possible choices for the functions h_0, h_1, and h_2. The mincycle algorithm allows for a much wider variety of choices. Cichelli's algorithm (as noted by Jaeschke and Osterburg [5]) might fail on a certain W because h_0, h_1, and h_2 behave pathologically on that particular W. The mincycle algorithm, however, can adjust these three functions as necessary to avoid such pathological behavior.

5. Although Cichelli's choices for h_0, h_1, and h_2 might appear simpler and faster than those chosen in the mincycle algorithm, the mincycle algorithm could choose, for example: $h_0(w) = \text{length}(w)$, $h_1(w) = \text{ord}(\text{first}(w))$, and $h_2(w) = 256 + \text{ord}(\text{last}(w))$. Now, if $g(h_1(w))$ were implemented as a lookup in a different table than $g(h_2(w))$, the resulting function would be equally fast.

6. For sets of cardinality 20 or more, the mincycle algorithm tends to be faster than Cichelli's algorithm. For sets of cardinality 60 or more, Cichelli's algorithm is often impractical, whereas the mincycle algorithm is practical for sets of cardinality 512 or more, provided that the cardinality of R is increased to an appropriate value.

We have coded both algorithms in standard Pascal, performing, in both cases, considerable although equivalent optimizations over the algorithms as presented in this article. Some typical results are summarized in Figure 8.

6. MORAL

Cichelli [2] draws the moral, "When all else fails, try brute force."[3] from his research on minimal perfect hash functions. From my own research in the same area, I feel this moral is inappropriate. I would, therefore, like to suggest the following moral, "With adequate forethought, brute force solutions can usually be avoided."

[3] Cichelli attributes this statement to J. Gillogly.

REFERENCES

1. Chang, C.C. The study of an ordered minimal perfect hashing scheme. *Commun. ACM 27*, 4 (Apr. 1984), 384–387.
2. Cichelli, R.J. Minimal perfect hash functions made simple. *Commun. ACM 23*, 1 (Jan. 1980), 17–19.
3. Dewey, G. *Relativ Frequency of English Speech Sounds.* Harvard University Press, Cambridge, Mass. 1923.
4. Harary, F. *Graph Theory.* Addison-Wesley, Reading, Mass. 1969.
5. Jaeschke, G., and Osterburg, G. On Cichelli's minimal perfect hash functions method. *Commun. ACM 23*, 12 (Dec. 1980), 728–729.
6. Jaeschke, G. Reciprocal hashing: A method for generating minimal perfect hashing functions. *Commun. ACM 24*, 12 (Dec. 1981), 829–833.
7. Sager, T.J. A new method for generating minimal perfect hash functions. Tech. Rep. CSc-84-15, University of Missouri-Rolla, Rolla, Mo. Nov. 1984.
8. Sprugnoli, R. Perfect hashing functions: A single probe retrieval method for static sets. *Commun. ACM 20*, 11 (Nov. 1977), 841–850.

CR Categories and Subject Descriptors: E.2 [**Data**]: Data Storage Representation—*hash-table representation*; H.3.3 [**Information Storage and Retrieval**]: Information Storage and Retrieval—*retrieval models, search process, selection process*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: searching, hashing, minimal perfect hashing

Received 4/83; revised 11/84; accepted 11/84

Author's Present Address: Thomas J. Sager, Department of Computer Science, University of Missouri-Rolla, Rolla, MO 65401.