Appl. Math. Lett. Vol. 2, No. 3, pp. 287–291, 1989
Printed in Great Britain

0893-9659/89 $3.00 + 0.00
Maxwell Pergamon Macmillan plc

An Informal Analysis of Perfect Hash Function Search

NICK CERCONE
School of Computing Science
Simon Fraser University
Burnaby, British Columbia, Canada

MAX KRAUSE
MacDonald Detwiller and Associates
3751 Shell Road
Richmond, British Columbia, Canada

Abstract. A brief explanation of perfect hash function search is presented followed by an informal analysis of the problem.

1. INTRODUCTION

Given a set of N keys and a hash table of size $r \geq N$, a *perfect hash function* maps the keys into unique hash table addresses. The hash table loading factor [LF] is the ratio of the number of keys to the table size N/r. A *minimal perfect hash function* maps N keys into N contiguous locations for a LF of one. Weiderhold [1] distinguishes deterministic and probabilistic direct access methods. Perfect hash functions are deterministic and do not permit collisions, thus guaranteeing single probe retrieval.

Perfect hash functions are difficult to find, even when almost minimal solutions are accepted. Knuth [2] estimates that only one in 10 million functions is a perfect hash function for mapping the 31 most frequently used English words into 41 addresses. Cichelli [3] devised an algorithm for computing machine independent, minimal perfect hash functions of the form:

hash value = hash key length + associated value of the key's first letter + associated value of the key's last letter

Cichelli's machine independent hash function algorithm incorporates a two-stage ordering procedure for keys which effectively reduces the the size of the search for associated values but excessive computation is still required to find hash functions for sets of more than 40 keys. Cichelli's method is also limited since two keys with the same first and last letters and the same length are not permitted.

2. CICHELLI'S ALGORITHM

The following is an outline of Cichelli's perfect hash function algorithm.

Algorithm 0

- *step1:* compare each key against the rest. If two keys have the same first and last letters and the same length, report conflict and stop; otherwise continue.
- *step2:* order the keys by non-increasing sum of frequencies of occurrence of first and last letters.
- *step3:* reorder the keys from the beginning of the list so that if a key has first and last letters which have appeared previously in the list, then that key is placed next in the list.
- *step4:* add one word at a time to the solution, checking for hash value conflicts at each step. If a conflict occurs, go back to the previous word and vary its associated values until it is placed in the hash table successfully, then add the next word.

We now give an informal analysis of the complexity of Cichelli's algorithm.

- *step1:* as formulated here, this is an $O(N^2)$ computation. The same check can be made by isolating and sorting the first and last letter for each key, then sorting the N keys into lexicographical order on these sets of isolated letters. We can then make one pass through the keys comparing neighboring keys for matching groups of isolated letters. The cost of this procedure would then be dominated by the cost of the lexicographical sort, which can be done in time proportional to $N \log_2 N$.
- *step2:* this initial ordering tallies the frequency of occurrence of first and last letters, which requires one pass over the N keys. A second pass is then made to calculate the sum of frequencies of each key. Sorting the keys into decending order of this sum, the dominant cost of this step, requires time proportional to $N \log_2 N$.

AML 2 3—F

step3: the second ordering is an $O(N^2)$ heuristic. As each key is added to the new ordering, the remaining keys in the old ordering are scanned to decide which, if any, of them now have their hash values determined. This may require (N-1)*(N-2)/2 operations.

step4: despite the tendency of the two orderings to reduce the search, for most sets of keys, the backtracking phase of this algorithm is the most expensive. An average case complexity measure is difficult to calculate; Simon and Kadane [4] estimate an average search to include about one-half the total search space, giving a casual estimate of $O(m^2/2)$, where m is the size of the domain of values for each letter and s is the number of letters which occur in first or last position.

Cichelli's algorithm uses key length and the first and last letters (without regard to letter position) as the hash identifier. The number of keys which can be distinguished is restricted to P*CH(*A*,2) where P is the maximum key length, CH is the familiar choose function, and *A* is the cardinality of the alphabet. Integer assignment values are found using a simple backtracking process. Cichelli proposes no method of choosing a value of m, the size of the domain of associated letter values. This is an important parameter of the problem since m is the branching factor of the backtrack search tree.

We have found that the time required to find a perfect hash function using this method varies greatly, depending less on the number of keys in the problem set than on the relationships among keys in terms of shared letters, Krause [5]. Because Cichelli's algorithm relies on a relatively uninformed exhaustive search of the solution space, the cost of finding a solution can be quite high. This, in turn, limits the maximum size of the problem sets to which the algorithm can be applied.

We next present an informal analysis of the nature of the problem. This analysis leads to some methods for overcoming the limitations of Cichelli's strategy while permitting us to retain its benefits.

3. AN INFORMAL ANALYSIS OF THE PROBLEM

We identify four subproblems: (1) choosing a set of formal properties of the keys to be used in the hashing function; (2) choosing a method of searching the space of possible solutions; (3) ordering the search variables to improve the performance of the search method; and (4) finding ways of enforcing a reasonable degree of minimality of the solution.

Choosing Hash Identifiers

A key is identified to be a sequence of length no greater than P, made up of symbols from alphabet A. We assume that A has a lexicographical ordering defined on it. T, a space of possible keys, is determined by a given P and A. If T = card(T) and A = card(A), then

$$T = A^P + A^{P-1} + \ldots + A = \Sigma_i A^i \ (1 \le i \le P) = A*(A^P-1)/(A-1) = \Theta(A^P)$$

as A becomes large. When A becomes arbitrarily large, the limit of $A/(A-1)$ approaches 1, reducing the resultant factor A^P-1 to A^P. Thus T grows at a rate polynomial in A and exponential in P. For example, consider A = the 26 lower case Roman letters and P=6; $T=\Sigma 26^i \approx 3.2*10^8$.

Letter Order in Keys

The number of keys which can be distinguished when the set of properties which are used as hash identifiers are not ordered is given by the expression CH(*A*+i-1,1), (1≤i≤P), where CH(n,m) is the familiar *choose* function, defined as CH(n,m) = n!/(m!*(n-m)!). If A=26 and P=6, then the size of the key space is

$$CH(A+i-1,1),\ (1 \le i \le P) = CH(26,1)+CH(27,2)+\ldots+CH(31,6) = 906{,}091 \approx 9*10^5.$$

Compare this number with $3.2*10^8$ distinguishable keys for the same values of A and P when the order of occurrence is taken into account. Without ordering, only about one in 350 keys in this example key space can be distinguished.

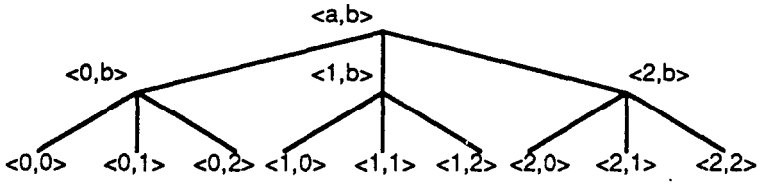
Hash Identifiers

We propose *Algorithm 1* to incorporate a procedure which automatically chooses, for each subset of keys of the same length, the smallest set of letter positions which distinguishes each key, when the order of occurrence of letters within a key is disregarded [5]. Since each letter has one associated value regardless of its position of occurrence, a key's hash address is determined by the combination of letters in chosen positions. The number of different (unordered) subsets of letters is much smaller than the key space with the same maximum length, so the number of subsets of the key space which can be processed via Algorithm 1 is restricted. To keep the search for, and subsequent use of, a perfect hash function as simple as possible, we select the smallest subset of P letter positions in keys of length P which distinguish each of the given keys. The limit on the number of keys which can be distinguished using only one letter position is A. It may be possible to distinguish up to A^2 keys with two chosen letter positions, but only if ab≠ba. In algorithm 1 this distinction is not made, which reduces the number of different keys which can be recognized by this method to $A(A\text{-}1)/2$ when two letter positions are chosen.

Algorithm 3, called algorithm cbk in [6], relies interactively on human judgment to choose a set of charac-teristics for the hash function. It also takes into account the position of occurrence of letters and thus has the greatest possible discriminatory power.

Assignment of Associated Letter Values

An efficient search of integer assignment of values to letters which map the keys into the hash table is necessary to find an acceptable solution in reasonable time. If we view the search space as a tree, as shown in Figure 1, there is a path of polynomial cost from the root (initial state of search) to each of an exponential number of possible solutions. In Figure 1, we have a set of three keys (N=3), two letters from chosen positions (s=2) and a maximum associated values of two (m=3, M=[0,1,2]). The number of different assignments of integers to letters is m^s, the number of leaf nodes in the tree. At tree depth one, the letter 'a' is assigned a value which determines the hash address of the key 'aa'. At depth two, the assignment of a value to 'b' determines the hash addresses of the keys 'bb' and 'ab'. The problem is to find a path which leads to an acceptable solution while generating as little as possible of the search tree, i.e., a classical backtrack search.



Small search space where the set of keys is (aa,ab,bb), L=2, M=N=3, and the letters are ordered <a,b>. The leaf nodes are the $m^s=3^2=9$ possible combinations of associated values. Minimal solutions are <0,1>, <1,0>, <1,2>, and <2,1>, non-minimal ones are <0,2> and <2,0>.

Figure 1. Small example of the search space.

Importance of Variable Ordering

For perfect hash function search, the criterion for backtracking, call it predicate Q, can be defined as: given an assignment of values $<x_1,...,x_n>$ to the variables $<a_1,...,a_n>$, define

$Q(x1,...,xn)$ = False if there exist k_i, k_j, $i \neq j$, in K such that for both keys, all letters in chosen positions are in $<a_1,...,a_n>$ and $H(K_i)=H(K_j)$.
= True otherwise.

When $Q(x_1,...,x_n)$ is True, then $<x_1,...,x_n>$ represents a perfect hash function for the subset of keys in K for which H now has a value (those for which all letters in selected positions have been assigned a value). The backtrack condition $Q(x_1,...,x_n)$ demands that no two keys have the

same hash address. In order to test the predicate efficiently we keep an array of possible hash addresses where we record which addresses are occupied by keys whose chosen letters have been assigned values previously. When no letter values have been assigned, Q is vacuously true. Suppose that $Q(x_1,...,x_{i-1})$ is satisfied; extending the solution to $Q(x_1,...,x_{i-1},x_i)$ involves two steps: (1) a value x_i is assigned to a_i; and (2) the set of "new" keys, whose hash addresses are dependent on letters which are all found in $a_1,...,a_i$ must have their hash addresses calculated and compared with the present state of the hash table. If we let d_i denote the number of keys for which a_i is the last chosen letter to be assigned a value, then each ordering of the letters will, in general, produce a different vector of values $<d_1,...,d_s>$. The sum of these di, $1 \le i \le s$, is N, the number of keys in the problem set. If we assign unit cost to generating the next trial value for a variable, then the cost of generating the entire tree is the number of nodes in the tree.

The number of nodes in a complete tree with depth s and branching factor m is the sum of the number of nodes at each level in the tree, where the root is at level 0:

$$C_{max} = \Sigma m^i,\ 1 \le i \le s,\ = m*(m^s-1)/(m-1) = O(m^{s+1}) = \Theta(m^s) \text{ as } m \rightarrow \text{infinity}$$

In our application, s is the number of letters to be assigned values and m represents the number of values in the domain of each variable, M=[0...m-1].

The choice of m is therefore critical since the size of the domain from which associated values are chosen determines the branching factor of the search tree. If m is set too small, there may be no solution to the problem set; if we set m's value high enough, say infinity, we are assured a solution exists, there is little reason to expect that a *minimal* solution will be the first one found. In practice we have obtained impressive results by setting m to N although we know of no analytic method of determining the optimal value for m given a set of keys.

In order to determine whether the current partial solution satisfies Q, one must perform d_j tests at each attempt to extend the solution to the j^{th} letter. We can assign to each node at depth j a cost of d_j+1, the cost of generating the next value for a_j plus d_j times the (unit) cost of testing the hash address. If j is defined as d_j+1, then the cost of visiting every node in the tree is $\Sigma\ (c_j*m^j)$, $1 \le j \le s$, which we consider the weighted cost tree [WCT]. Each ordering of the variables determines a (possibly different) value of WCT; we consider that ordering of the variables which give the minimum WCT as the best ordering.

Ordering Search Variables

Given s search variables $a_1,...,a_s$, what is the best ordering. Consider the permutation $B=<a_1,...,a_s>$ and define $D(B)=<d_1,...,d_s>$ to give the number of keys d_i whose hash addresses are newly determined when a_i is assigned a value. Let $C(B)=<c_1,...,c_s>$ be D(B) with one added to each d_i so that c_i is the cost of visiting any node at level i in the tree. We regard C(B) as a vector of coefficients for the series of m_i terms, $1 \le i \le s$, which make up the WCT = (c_i*m^i), $1 \le i \le s$, $= c_0 + c_1*m + ... + c_s*m^s$. The initial term, with c_0 defined to be one (unit time expense), is the cost of generating the root node of the search tree. Note the m factor in each of the terms in the total cost is growing exponentially with its distance from the root.

Examining WCT convinces us that we want the smallest possible values assigned to the coefficients in the order $c_s, c_{s-1},..., c_2, c_1$, where c_s is as small as possible and c_1 is as large as possible. We cannot have $d_s<1$, since at least one key has the last letter in the ordering as its last letter to be assigned a value. The best we can find is a letter a_s which has a frequency count of one so that it can be the determining value of only one key, giving c_s a value of two.

We can show that m^s is larger than the sum of the remaining terms in the polynomial which describes the size of the search tree [5]. Since m^s will contribute most of the cost of the tree, its coefficient in the WCT *must* be the smallest which occurs in any of the s! possible permutations of the variables. We therefore want to find a key which has at least one unique letter occurrence since it is only such a letter which can come last in the ordering and still place a single key in the hash table.

A heuristic ordering strategy for the letters based on this observation would order the letters

by frequency in non-increasing order, so that a_1 would have the highest frequency of occurrence and a_s would have the lowest. We find that this arrangement tends to occur when we first order the keys by sum of letter frequencies, then from each key choose the letters which have not occurred before in decreasing order of frequency of occurrence. The second ordering has the effect of making the coefficients of the m factors of the cost equation increase for the smaller factors and decrease for the larger m factors.

The optimal ordering of the search variables, $B_{min}=<a_1, a_2,..., a_s>$, is that for which the WCT is a minimum. If we were to generate all s! permutations of the letters, we would find that the optimal ordering is that for which D(B), and therefore C(B), has the largest lexicographical sort value.

We can approach the optimal ordering by examining far fewer than s! permutations. This is accomplished by refining the second ordering, as suggested by Slingerland and Waugh [7], such that "each sublist of words which have equal frequency counts be ordered such that the words that will have the greatest second ordering effect, that is, words that will 'expose' the most words from the rest of the list, occur first". This is explained by our model, since at each stage in the reordering process, we select the next key whose new letter will determine the greatest number of hash addresses among those keys which have the highest current sum of frequencies. This strategy tends to increase the coefficients of small m factors and thus decrease the coefficients of large m factors, which, in turn, reduces the WCT.

Note that the WCT indicates only the size of the tree we are searching; it is a measure of the worst case complexity when we seek only one acceptable solution. The greatest value of the backtracking approach is that if we test the validity of all partial solutions, when we find that a partial solution $<x_1,...,x_i>$ does not satisfy Q, we can prune the subtree which has x_i as its root and avoid generating, for a value rejected at level i, $\sum m^j$, $1 \le j \le s-1$, full and partial solutions which have $<x_1, x_2,...,x_i>$ as an initial segment. The cost of this rejected subtree is $\sum (c_{1+j} * m^j)$, $1 \le j \le s-1$. Fortunately, the frequency of occurrence of a letter a_i is an excellent heuristic value for predicting how likely it is that a_i occurs in a key which may collide with other keys.

In general, we may conclude that any polynomial-cost analysis that can be performed dynamically in the depth-first search which allows us to exclude from consideration values in the domain of a search variable will be worth pursuing since an exponentially-growing subtree will be pruned for each potential value we eliminate.

4. POSTSCRIPT

Remember, the ideal backtrack search is one that never backtracks. In order to achieve that level of performance, the search must be organized in such a way that a choice made at any stage of the search is known to be ultimately acceptable. We presented such a strategy in [5].

REFERENCES

[1] Weiderhold, G. (1977) **Database Design**, McGraw Hill, New York.

[2] Knuth, D. (1973) **The Art of Computer Programming 3: Sorting and Searching**, Addison-Wesley, Reading, Massachusetts.

[3] Cichelli, R. (1980) Minimal Perfect Hash Functions Made Simple, **CACM 23**, 17-19.

[4] Simon, H., and Kadane, J. (1976) Problems of Computational Complexity in Artificial Intelligence, in J.F. Traub (ed.) **Algorithms and Complexity**, Academic Press, New York, 281-299.

[5] Krause, M. (1982) Perfect Hash Function Search with Application to Computer Lexicon Design, M.Sc. thesis, Computing Science, Simon Fraser University, Burnaby, B.C.

[6] Cercone, N. (1987) Finding and Applying Perfect Hash Functions. **Applied Math Lets 1(1)**, 25-29.

[7] Slingerland, J., and Waugh, M. (1981) On Cichelli's Algorithm for Finding Minimal Perfect Hash Functions, **CACM 24(5)**, 322.