FINDING MINIMAL PERFECT HASH FUNCTIONS

Gary Haggard
Department of Computer Science
University of Maine at Orono

and

Kevin Karplus
Department of Computer Science
Cornell University

ABSTRACT

A heurisitic is given for finding minimal perfect hash functions without extensive searching. The procedure is to construct a set of graph (or hypergraph) models for the dictionary, then choose one of the models for use in constructing the minimal perfect hashing function. The construction of this function relies on a backtracking algorithm for numbering the vertices of the graph. Careful selection of the graph model limits the time spent searching. Good results have been obtained for dictionaries of up to 181 words. Using the same techniques, non-minimal perfect has functions have been found for sets of up to 667 words.

INTRODUCTION

A minimal perfect hashing function is a one-to-one, onto mapping from a set of keys K to n consecutive integers. This paper presents a method for quickly finding such functions for sets of up to about 180 words. The same techniques can be applied to larger sets to find perfect hash functions (still one-to-one, but $n > |K|$).

Ordinary hash functions are cheap to compute, and families of good hash functions have been described in the literature [CW]. Examining arbitrary hash functions until a perfect one is found has been attempted [Sp], but perfect hash functions are too rare for this to be feasible on sets of size n where n is large (of the $n^{|K|}$ possible hash functions, only $n!/(n-|K|)!$ are perfect. Cichelli presented a method for finding minimal perfect hash functions [C]. Only hash functions of the form

h(word) = g(first letter)
+ g(last letter) + length(word)

were considered. Many useful word sets have no perfect hash function with that form. Even using different functions for the first and last letter, or considering other pairs of letter positions is not enough. For example, the complete list of PASCAL reserved words and pre-declared identifiers contains the six words CASE, ELSE, PAGE, READ, REAL, TRUE, and TYPE. No pair of letter positions can distinguish these words (see Figure 1).

	first letter	second letter	third letter
second letter	REAL=READ		
third letter	REAL=READ	REAL=READ	
fourth letter	TRUE=TYPE	CASE=PAGE	CASE=ELSE

Figure 1. No two selector functions can distinguish CASE, ELSE, PAGE, READ, REAL, TRUE, and TYPE.

Sager [Sa] proposes an optimization for the method of Cichelli which uses a different intermediate process to prepare for the backtracking search for the required functions. Our method generalizes that of Cichelli, and uses a more flexible intermediate processing step to prepare for the backtracking search than Sager. We search for hash functions of the form

$$h(word) = length(word) + \sum_i g_i(\sigma_i(word)),$$

where σ_i (word) selects a letter from the word based on the length of the word, and g_i(letter) is computed) by table lookup. The method described here allows the construction of the 76 word dictionary of Pascal reserved words and predefined identifiers without special considerations.

The search has two parts. First, we look for selector functions such that the vector (length,σ_1,...,σ_m) uniquely identifies each word. The vectors can be thought of as the edges of an m-partite hypergraph whose vertices are the letters selected by σ_i. The word length is kept as a label for the edge. Second, we look for values of g_i(letter) such that each word maps to a different integer in 1,2,...,n where n is the size of the word set. That is, a value is assigned to each vertex of the hypergraph, so that the sum of the edge label and the values on the vertices is a different integers in n for each hyperedge.

Edges incident on vertices of degree one can be assigned any desired hash value, since the vertex can be assigned a value independent of any other vertex value. Thus the vertex assignment problem can be simplified by deleting all edges containing a vertex unique to that edge. The reduced graph may have new vertices of degree one, allowing more edges to be deleted. Repeating the process eventually results in a graph with no vertices of degree one. The removed edges (which we call tree edges) are assigned hash values in reverse order of their removal after all the edges in the reduced graph have been assigned values.

For small sets the tree edges are a substantial part of the graph, and minimal perfect hash function can often be found by hand. For example, for the month abbreviations JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, and DEC, choosing the second and third letters gives a graph containing only tree edges (see Figure 2). Arbitrarily choosing JAN=1,...,DEC=12, we can assign vertex values as shown in Figure 3.

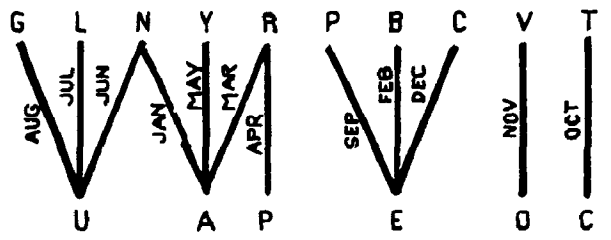

Figure 2. Graph for second and third letters of the abbreviations for month names.

A = -2	N = 0	JAN = 1
U = 3		JUN = 6
	R = 2	MAR = 3
P = -1		APR = 4
	Y = 4	MAY = 5
	L = 1	JUL = 7
	G = 2	AUG = 8
E = -1	B = 0	FEB = 1
	P = 7	SEP = 9
	C = 10	DEC = 12
O = 8	V = 0	NOV = 11
C = 7	T = 0	OCT = 10

Figure 3. Vertex assignments for the abbreviations of month names.

A set of selector functions (choice of letter positions within the words) is chosen by doing a limited search of all sets composed of selector functions from a fixed family of functions. We use a family of 27 different selector functions, but new functions can be easily added to the family. For each set of selector functions, a word hypergraph is built. If no two words map to the same edge with the same length lable, the set of selector functions is accepted, and vertex value assignment starts. If all the sets allow words to map to identical edges, the best few sets are kept, and new larger sets are generated from them. From a set of selector functions, a larger set is constructed by adding a function in the family that is not already in the set.

First we consider the empty set: are the words separated by length alone? Then we consider all extensions of the empty set: does any single selecter function suffice? The best few selector functions are remembered, and all pairs of selector functions that include one of the best functions are tried. This continues for higher dimensions. The best few sets of k selector functions are remembered, and all sets of k+1 functions that include a remembered set are tried. Sets of selector functions are tried until a good set is found, or the size of the sets gets too large.

The quality of a set of selector functions is measured by a weighted sum of the number of distinct edges, the number of tree edges, and the number of vertices in the word hypergraph. Of these, the tree edge count is more important. For more details on the weights used, see [KH].

After the selector functions have been chosen, values have to be assigned to all vertices of the word hypergraph. The tree edges can be removed, and the corresponding vertices assigned values in reverse order after the rest of the vertices have values. For the main body of the graph, the vertex assignment proceeds as follows:

1) choose a vertex,

2) if no legal assignment exists, backtrack and change a previous choice,

3) otherwise, assign the smallest legal value to the vertex (0 if the vertex value is unconstrained),

4) repeat 1-3 until all vertices have been assigned.

The simplest possible backtracking scheme is to have a fixed ordering of the vertices, and undo the most recent choice when a conflict occurs. This scheme works well for small graphs (such as those in [C]), but can take a long time on larger ones. Various heuristics were used to speed up both the vertex choice and the backtracking.

Vertex choice heuristics attempt to choose the most difficult vertices first, thus triggering necessary backtracks as soon as possible. Define E_{min} as the set of edges with the fewest unassigned vertices (excluding edges with all vertices assigned). The best vertex-choice heuristic found was to choose among the vertices with the most edges in E_{min} the vertex that has the widest range of partial sums for edges in E_{min}. This heuristic works well for hypergraphs of dimension one or two, but not as well for higher dimensions. Only edges that have only one vertex value unassigned affect vertex values (we call these almost completed edges). If the graph has no almost completed edges, a value is assigned arbitrarily to the chosen vertex.

Backtracking heuristics are more complicated than the vertex-choice ones. The hash function searches that succeed backtrack rarely, so the heuristics don't affect them much. The searches that run a long time spend almost all the time doing backtracking.

Three different conflicts can trigger backtracking. Edge conficts occur when two different almost completed edges have the same value. Too-big conflicts occur when the range of partial sums for almost completed edges is larger than the range of unused edge values. No-fit conflicts occur when every vertex assignment will make some almost completed edge conflict with an existing completed edge.

For edge conflicts, vertex values are popped until the partial sum of the conflicting edges differ. A larger value is assigned to the most recently popped vertex, and the vertex value assignment proceeds forward again. For too-big conflicts, vertex values are popped until the last vertex removed is in the edge with the smallest partial sum and not in the edge with the largest partial sum, or a vertex assignment can be made so the partial sum of the almost completed edges will fit. For no-fit conflicts, vertex values are popped until some vertex of an almost completed edge has been removed. The almost completed edge with the highest partial sum is excluded, since increasing the assignment for its vertices is not likely to resolve the conflict.

For more details on the backtracking heuristics, and statistics on the occurrences of the different conflicts, see [KH].

Theoretical analysis of running time is difficult, since we lack a convincing model for sets of words. Empirically, our program takes about $.06(\text{words})^{1.5}$ CPU seconds for a successful search on a Vax 11-780. The time doesn't seem to depend on whether a minimal perfect hashing function or a perfect hashing function is sought. Unsuccessful searches take far longer, and have not been allowed to run to completion.

REFERENCES

[C] Richard J. Cichelli. "Minimal Perfect Hash Functions Made Simple." Communications of the ACM 23(1) (January 1980), 17-19.

[CW] J. Lawrence Carter and Mark N. Wegman. "Universal Classes of Hash Functions." Proceedings of the 9th Annual ACM Symposium of the Theory of Computing (May 1977), 106-112.

[KH] Kevin Karplus and Gary Haggard. Finding Minimal Perfect Hash Functions. Cornell Computer Science Technical Report TR84-637 (September 1984).

[Sa] T. J. Sager. "A Polynomial Time Generator for Minimal Perfect Hash Functions. Communications of the ACM 28(5) (May 1985), 523-532.

[Sp] R. Sprognoli. "Perfect hashing functions: A single probe retrieving method for static sets." Communications of the ACM 20(11) (November 1977), 841-850.