OREGON STATE
UNIVERSITY
COMPUTER
SCIENCE
DEPARTMENT

A LETTER ORIENTED MINIMAL PERFECT HASHING FUNCTION

Curtis R. Cook
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

R. R. Oldehoeft
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523

82-1-2

A LETTER ORIENTED MINIMAL PERFECT HASHING FUNCTION

CURTIS R. COOK
COMPUTER SCIENCE DEPARTMENT
OREGON STATE UNIVERSITY
CORVALLIS, OREGON 97330

R. R. OLDEHOEFT
COMPUTER SCIENCE DEPARTMENT
COLORADO STATE UNIVERSITY
FORT COLLINS, COLORADO 80523

ABSTRACT

Cichelli has presented a simple method for constructing minimal perfect hash tables of identifiers for small static word sets. The hash function value for a word is computed as the sum of the length of the word and the values associated with the first and last letters of the word. Cichelli's backtracking algorithm considers one word at a time and performs an exhaustive search to find the letter value assignments. In considering heuristics to improve his algorithm we were led to develop a letter oriented algorithm that handles more than one word per iteration and that frequently outperforms Cichelli's. We also investigate the impact of relaxing the minimality requirement and allowing blank spaces in the constructed table. This substantially reduced the execution time of the algorithm. This relaxation and partitioning data sets are shown to be two useful schemes for handling large data sets.

Key words: hash tables, perfect hashing functions, minimal hash tables.

INTRODUCTION

A minimal perfect hash table is a static data structure with two useful attributes: there is no unused space in the table, and using the associated hashing function one can, in exactly one probe, determine the location in the table of the search key or conclude that the key is not present. Data which might be stored this way include reserved and predefined words in programming languages, common English words, and other set of identifiers.

Jaeschke [2] presented a method for building minimal perfect hash functions. For a set W of positive integers, his method (called reciprocal hashing) attempts to find integer constants C, D and E such that for each integer i in W

$$h(i) = C/(D*i + E) \mod n, \quad W = n$$

is a minimal perfect hashing function. The existence of h is guaranteed. However, the value of C may be very large and the user must "prescribe a limit for the C values to be examined such that the algorithm terminates in economical time."

Cichelli [1] presented an excellent heuristic which will successfully build tables and associated hashing functions for a number of representative data sets.

The problem is interesting both for its applications and because it is essentially hard: the only known methods rely on exhaustive search techniques to place words in the table and/or vary parameters in the hashing function. Because of this, it cannot be claimed that one algorithm is more efficient then another for every possible data set. One can only cite its performance on certain data sets or compare it with other algorithms on various data sets.

Cichelli solved a special case of the general problem. He restricted the form of the hashing function to

F(word) = Length(word) + value(first letter) + value(last letter)

That is, the table position of a word is computed as the sum of its length plus the values associated with the first and last letters of the word. This is also the problem we address in this paper. Building a minimal table involves assigning values to letters so that each word is mapped to a unique spot in the table, and no internal blank spaces remain in the table.

Even though his hashing function is appealing because of its simplicity and machine independence, it is not guaranteed to work in all cases. For example, a set containing two words with the same first and last letter sets and length (e.g., THEN, NEAT) cannot be handled. When this occurs, one can choose letter pairs other than the first and last, or the set can be partitioned into groups so that a perfect table may be constructed for each. Other restrictions on the possibility of perfect table construction can be found in Jaeschke [3].

To find the letter value assignments, Cichelli used a backtracking al-

gorithm that considered one word at a time and performed an intelligent exhaustive search. In our attempts to improve Cichelli's algorithm we were led to develop a letter oriented algorithm that considers more than one word at a time and that frequently outperformed Cichelli's. We also investigated the impact of removing the minimality restriction and allowed blank spaces in the table. This relaxation and partitioning data sets are shown to be two useful schemes for handling large data sets.

CICHELLI'S ALGORITHM

In this section we present a description of Cichelli's backtrack algorithm. We include enough detail to understand it and for comparison with our letter oriented algorithm.

Cichelli's Algorithm (WHASH)

1. Find frequencies of letters occurring at either end of words in set.
2. Order the words by sum of frequencies of first and last letters. This step causes words with frequently occurring letters, and hence most impact on the placement of other words, to be considered early by the algorithm.
3. Adjust ordered list of words. This step moves forward the words whose first and last letter values are determined when the algorithm processes an initial portion of the list. With this adjustment potential collisions will be determined earlier by the algorithm.
4. Assignment of letter values. This is the time consuming exhaustive search part of the algorithm. The algorithm considers each word in order beginning with the first. It attempts to find letter values for any letter or pair of letters not already assigned a value. Beginning with 0 it searches for a value or values that will map the word to an empty table slot. If it finds a value it assigns the letter the value and places the word in the hash table. If no value assignment is possible, the algorithm backs up to the previous word and trys to find another value assignment.

We tried several modifications of Cichelli's algorithm with mixed results. However, we were able to conclude that the algorithm was very sensitive to the word ordering. For several data sets, a slight change in the word ordering led to a significant improvement or degradation in performance.

One important note about the word list size. Since the algorithm performs an exhaustive search, the list should not be large. Cichelli recommends limiting the size to 45. Our results support this.

LETTER ORIENTED ALGORITHM

Our attempts to improve Cichelli's algorithm led to the development of a letter oriented algorithm that deals with letters directly rather than indirectly. The algorithm is called letter oriented because the ordering of the letters determines the word ordering. In this section we present the algorithm and compare it with Cichelli's.

Letter Oriented Algorithm (LHASHA)

1. Compute frequency of letters occurring at either end of words in the set. If a letter appears as both the first and last letter of the same word assign it a large value.
2. Order the letters by decreasing frequency, resolving ties arbitrarily.
3. Let each word be represented by a triple: (letter 1, letter 2, length) where letter 1 is the first letter and letter 2 is the last letter. For each triple (word) interchange letter 1 and letter 2 if letter 2 precedes letter 1 in the letter ordering.
4. Using letter 2 as the key, sort the list of triples in descending order. Figure 1 illustrates the affect of steps 3 and 4 for the set of Pascal Reserved words.
5. Assignment of letter values. Begin with the first triple. For each group of triples with the same letter 2 attempt to find a letter value assignment for letter 2. Note that for this group of triples, either letter 2 = letter 1 or letter 1 precedes letter 2 in the letter ordering and letter 1 has previously been assigned a value. Thus only letter 2 needs to be assigned a value to place the group of triples (words) in the hash table.

   Beginning with 0 search for a value for letter 2 that maps the group of triples to distinct empty table slots.

   If it finds a value it assigns the value to letter 2 and places the group of triples (words) in the table.

   If it does not find a value, it backs up to one of two places depending on the reason for its failure. One reason for failure is that two triples in the group have the same value of letter 1 and length sums. Hence the two triples will have identical hash values for any value assignment to letter 2. In this case the algorithm backs up to the nearest group that computes a value assignment for a letter 1 of one of the two triples. For all other failures, the algorithm backs up to the previous group of triples.

   Note that when the algorithm backs up it unassigns letter values and removes the associated hash table entries along the way.

Assigning large values to letters that occur both first and last in a word (Step 1) forces these letters to the front of the letter ordering (Step 2). This heuristic was based on the following observations. If such a letter occurs later in the table construction process, then the only possible value assignments may easily result in a non-minimal table, especially if more than one-half of the words have already been placed. Further,

all possible table indices for such a word will be either all odd or all even, depending only on the word length and hence there is less flexibiliy in the placement of these words.

Letter ordering after Step 2	Triples (words) ordering after Step 4	
E	(E,E,4)	ELSE
L	(L,L,5)	LABEL
D	(E,D,3)	END
O	(D,O,2)	DO
T	(D,O,6)	DOWNTO
N	(E,O,9)	OTHERWISE
F	(T,O,2)	TO
.	(E,T,4)	TYPE
.	.	.
.	.	.
	.	.

Figure 1. Partial letter ordering and triple ordering after steps 3 and 4.

Notice that the major differences between our letter oriented algorithm and Cichelli's are: The letter oriented algorithm considers groups of words rather than a single word at each step; the letter oriented algorithm is able to back up immediately to the letter value assignment that caused the conflict in some cases; and the letter oriented algorithm searches for a value assignment for only one letter at each step.

RESULTS

Table 1 gives the comparative timings for Cichelli's algorithm (WHASH) and our letter oriented algorithm (LHASHA). All programs were written in Pascal and run on a CDC Cyber 720 computer system under the NOS 1.4 operating system and using University of Minnesota/Zurich Pascal 3.2. Descriptions of the data sets can be found in the Appendix.

The performance of LHASHA has been encouraging. Table 1 shows that for all data sets except ENGWORDS it outperfoms WHASH. Unlike the word oriented WHASH, LHASHA handles words in groups which are associated with a letter, rather than one word at a time. Further, backtracking after discovery of a failure may revert back immediately to the letter assignment that caused the problem, while Cichelli's algorithm backtracked just one word at a time.

Experience and reflection have led us to two variations of LHASHA. Both result from the observation that, since there is considerable arbitrariness in the sorted letter order due to ties among frequencies , a quite variable amount of time can be taken to build a minimal perfect hash table. This reflects our experience with the sensitivity of the word order-

ing in our attempts to discover improvements of Cichelli's algorithm. Also observe that there is considerable freedom in assigning a value to a letter that occurs at the end of only one word. Accordingly, we subtract one from the frequency of a letter occurring at the other end a of word with such a letter, thereby reducing its position in the sorted letter list. Call the modified algorithm that implements this LHASHB.

The third version of the letter oriented algorithm incorporates the previous modification and also includes more information with which to sort letters. For each letter, compute the sum of the frequencies for the letters that appear with it at the other ends of words. Then the sort key is computed by

(letter frequency) * 1000 + (sum of associated frequencies)

The rationale for this alteration is that two letters with the same frequency of occurrence should be considered in an order consistent with the impact of the value assignments will have on other words. Call this version LHASHC.

Both LHASHB and LHASHC result in fewer ties among letter sort keys, and both manage to encode more useful information into the sort key values. The relative performance of the three versions is shown in Table 1. Note that LHASHB and LHASHC do not consistently outperform LHASHA, but LHASHC is often better and never a great deal worse than LHASHA. For an efficient, stable algorithm, LHASHC seems to be the best choice of the three.

One shortcoming we noticed in the letter oriented algorithms was their occasional poor performance in what we call the "end game." This phase occurs when all letters with frequency greater than one have been assigned values; all that remains is to assign values to letters which occur only once. The words involved appear at the end of the word ordering. We thought that with the freedom to assign any value to these letters it would be straightforward for the algorithms to place these words into whatever blank spaces remain in the hash table. However, the algorithms only consider blank spaces at positions beginning at the sum of the length of the word and the value assigned to the other letter. Hence a blank space that occurs early in the table will not be considered and the last word cannot be placed if the only an internal blank is at a position less than the length of the word. Thus the algorithm will do a great deal of backtracking resulting in the reorganization of most of the table.

To avoid this problem we added a special test for handling letters with frequency one to LHASHC so that it looks for blank spaces in the table begining at the first word in the table. Note that this may result in negative letter values in these cases. Call this algorithm LHASHD.

From Table 1 we see that its performance is the same as that of LHASHC except for two cases in which execution was improved by a substantial amount. For the 55 words of a Pascal-like language for process control applications (not included in Appendix) algorithms LHASHA, LHASHB and LHASHC were all unsuccessful after a great deal of computer time, while LHASHD only required 0.3 seconds.

Data set	WHASH	LHASHA	LHASHB	LHASHC	LHASHD
ENGWORDS	0.3	1.7	4.6	3.0	0.2
ASCIIWDS	290.4	0.2	0.1	0.2	0.2
	(1 blank)				
PRESWDS	1.2	0.1	0.1	0.1	0.1
PDEFWDS	90.6	0.2	32.0	0.3	0.3
	(4 blanks)				
OPCODES	0.2	0.1	0.4	1.5	1.4
VALWDS	55.5	1.3	0.3	0.5	0.5
	(15 blanks)				

Table 1. Performance of Algorithms

NON-MINIMAL TABLES

In this section we present the results of an investigation of the effect on algorithm performance when a controlled number of blank slots are allowed in the perfect hash table. We thought that the extra space should speed up the algorithm and allow it to handle large data sets. Table 2 gives performance measurements for several data sets using a version of LHASHC which allows a parameterized number of blanks in the perfect hash table it constructs. Not included in Table 2 are results for LHASHA and LHASHB; they generally performed like LHASHC.

As expected, the small number of blank slots generally reduced the execution time by varying amounts. In a few cases the construction time actually increased slightly before decreasing when a few more blanks were allowed (see results for the data set VALWDS). Because the algorithm always attempts to find small values for letters, most of the blanks occur at the ends of the hash tables. One expecially interesting observation was that, in almost every case, the algorithm showed little or no improvement after more than 10 percent empty slots were allowed. Hence a "10 percent rule" seems to be a good heuristic for how many blank slots to allow in order to achieve optimal performance. For comparison purposes, we also ran each of the algorithms with the hash table size doubled, i.e. with 100 percent blank space. No improvement in algorithm speed was observed over allowing 10 percent blank space.

For a large data set we chose the 105 PL/1 key words and their abbreviations (with some identifiers altered to eliminate conflicts). Each algorithm was unsuccessful when the "10 percent rule" was applied. However, when the table size was doubled, LHASHC constructed a perfect table in 0.8 seconds. While this may suggest doubling table size as a solution for large data sets, when the number of identifiers increases, so does the probability of conflicts in the data set. So it is very likely that either the algorithm or the data set will need to be modified.

Number of Blanks	ENGWORDS 31 words	ASCIIWDS 34 words	PRESWDS 36 words	PDEFWDS 39 words	OPCODES 36 words	VALWDS 57 words
0	3.0	0.2	0.1	0.3	1.5	0.5
1	0.1	0.2	0.1	0.2	0.2	0.4
2	0.1	0.2	0.1	0.3	0.1	0.3
4	0.1	0.2	0.1	0.2	0.1	0.3
6	0.1	0.2	0.1	0.2	0.1	0.9
8	0.1	0.2	0.1	0.2	0.1	1.1
10	0.1	0.2	0.1	0.2	0.1	0.3
Doubled	0.1	0.2	0.1	0.2	0.1	0.3

Table 2. Performance for Non-minimal Tables for LHASHC

PARTITIONED TABLES

For some data sets a single minimal perfect hash table cannot be constructed because of the size of the data set or conflicts among words in the set. In the first case the best algorithms may not execute fast enough to be practical (all versions have exponential time complexity). It is easy to partition the data set (e.g., by subranges of letters using the second letters of words) so that each partition is of manageable size and constant search time is preserved: table selection followed by a single probe into the table.

In the second case the data set may be small enough but contain unresolvable conflicts, or in a large data set the plethora or words and the limited range of word lengths along with the constant number of letters may lead inevitably to the impossibility of table construction. By identifying conflicting words and placing them in different partitions along with others to make the partitions nearly equal in size, minimal perfect tables can be built. However, it may be that table selection will require time proportional to the number of partitions if a clever scheme for table selection cannot be improvised. For example, consider the data set containing the 105 PL/1 key words and their abbreviations. Using the first and last letters, the algorithm detects the following collisions: ACTIVATE VS. ALLOCATE, COMPLETION vs. CONVERSION, COL vs. CTL, CPLN vs. CONN, NOSTRINGSIZE vs. NOZERODIVIDE, NOOFL vs. NOUFL, PREC vs. PROC, NOZDIV vs. NOCONV, and UNFL vs. UNAL. By partitioning the data set into the letter intervals A:E, F:M, N:S, and T:Z using the third letter if word length is at least four, or using the second letter for shorter words, we obtain partitions of sizes 30, 23, 36, and 16 words respectively. Then LHASHD builds minimal perfect hash tables for each in 0.6, 0.1, 4.6, and 0.1 seconds respectively.

Finally, perfect hash tables may have applications in problems involving non-static data. The fast construction seen for 10 percent minimal tables combined with partition mechanisms may provide an information storage and retrieval scheme for large identifer sets which change very slowly relative to frequent searches which must be performed quickly.

CONCLUSIONS

We are fully aware of the dangers of any sweeping conclusions based on the results of a limited number of test data sets. In fact, the test data results underscore the data sensitivity and relative instability of the various algorithms.

In almost all cases the letter oriented algorithms substantially outperformed Cichelli's word oriented algorithm. For all data sets these three letter oriented algorithms ran in less than one-half second.

Relaxing the minimality condition results in only a small speedup in the algorithm in some cases. The "10 percent rule" seems to be a good guideline for the number of blank table slots.

As mentioned by Cichelli, one should not apply these minimal perfect algorithms to data sets larger than about 50 identifiers. Two possible solutions for larger sets are doubling the table size or partitioning the data sets into smaller segments.

At least three open problems remain of interest.

1. Find an efficient algorithm for determining whether or not a minimal perfect hash table exists. If not, find a method of resolving the word conflicts.
2. Find an efficient algorithm to find either another letter pair (other than the first and last) or another scheme to handle conflicts, i.e. two words with the same first and last letter sets and length.
3. Find an efficient algorithm for segmenting large data sets.

Copies of the program are available upon request.

REFERENCES

1. R. J. Cichelli, Minimal perfect hash functions made simple, CACM 23 (1980), pp. 17-19.
2. G. Jaeschke, Reciprocal hashing: a method for generating minimal perfect hashing functions, CACM 24 (1981), pp. 829-833.
3. G. Jaeschke and G. Osterburg, On Cichelli's minimal perfect hash functions method, CACM 23 (1980), pp. 728-729.

APPENDIX

This section lists all of the data sets referenced in this report.

1. ENGWORDS: 31 most frequently occurring English words.
I, IT, FROM, THAT, AT, TO, A, HE, THE, HAVE, OF, ARE, IS, HIS, THIS, AS, WITH, WHICH, IN, NOT, HER, YOU, ON, OR, BUT, FOR, WAS, HAD, BY, BE, AND.

2. ASCIIWDS: 34 identifiers associated with non-printable ASCII characters.

   FF, FS, SYN, EOT, DLE, BS, SUB, ETB, LF, NUL, DEL, RS, CAN, BEL, ESC, STX, ETX, SI, DCI, NAK, HT, CR, SOH, SO, SP, GS, US, EM, ENQ, DCW, DCJ, DCZ, VT, ACK.

3. PRESWDS: the 36 reserved words of Pascal (including OTHERWISE).

   DO, END, ELSE, LABEL, DOWNTO, NIL, TO, OTHERWISE, TYPE, WHILE, OR, NOT, THEN, OF, RECORD, FILE, PACKED, MOD, CASE, PROCEDURE, REPEAT, DIV, AND, FUNCTION, FOR, CONST, IN, SET, GOTO, BEGIN, UNTIL, VAR, WITH, PROGRAM, ARRAY, IF.

4. PDEFWDS: 39 predefined identifiers of Pascal (except ODD, which conflicts with ORD and was omitted).

   TEXT, RESET, MAXINT, TRUE, SQR, SQRT, REWRITE, GET, READLN, EOLN, SIN, CHR, CHAR, TRUNC, READ, ROUND, COS, SUCC, PUT, PACK, DISPOSE, EXP, PAGE, WRITE, NEW, INPUT, OUTPUT, INTEGER, WRITELN, EOF, REAL, FALSE, ABS, PRED, LN, BOOLEAN, ARCTAN, ORD, UNPACK.

5. OPCODES: 36 mnemonics for a simple assembly language.

   SFLGS, STOY, STOAY, BSS, STCTL, STOA, LOADY, LOADAY, ADDAY, LDIA, LOADA, EQUALS, ENTRY, LAND, ADD, JMPY, CLCTL, JMPLDY, END, JPSUB, DECYSZ, MOVAY, NAME, STFLG, EXTERN, MOVYA, COMP, LIOR, JUMP, INCSZ, INCYSZ, USE, VALUE, OUTA, HALT, CLFLG.

6. VALWDS: 57 reserved words of the dataflow language VAL.
REML was changed to REMLO (to avoid conflict with REAL) and TRUE and FALSE were omitted, because TRUE conflicts with TYPE.

   ELSE, ZERODIVIDE, TYPE, END, HIGH, TAGCASE, ERROR, REMH, REPLACE, RESULT, EMPTY, RECORD, THEN, EVAL, LET, ELSEIF, TAG, FOR, REAL, NEGOVER, NEGUNDER, INT, NIL, NULL, AT, ITER, ADDH, FUNCTION, FORALL, MAKE, JOIN, IF, MOD, SIZE, ARRAY, MISSELT, ARITHERROR, ADDL, MIN, EXP, DO, RETURNS, LOW, OTHERWISE, REMLO, UNDEF, CHAR, BOUND, CONSTRUCT, POSOVER, POSUNDER, IS, ONEOF, BEGIN, MAX, BOOL, ABS.