PERFECT HASHING USING SPARSE MATRIX PACKING

MARSHALL D. BRAIN and ALAN L. THARP
Computer Science Department, North Carolina State University, Raleigh, NC 27695-8206, U.S.A.

(Received 4 April 1989; received for publication 3 January 1990)

Abstract—This article presents a simple algorithm for packing sparse 2-D arrays into minimal 1-D arrays in \( O(n^2) \) time. Retrieving an element from the packed 1-D array is \( O(1) \). This packing algorithm is then applied to create minimal perfect hashing functions for large word lists. Many existing perfect hashing algorithms process large word lists by segmenting them into several smaller lists. The perfect hashing function described in this article has been used to create minimal perfect hashing functions for unsegmented word sets of up to 5000 words. Compared with other current algorithms for perfect hashing, this algorithm is a significant improvement in terms of both time and space efficiency.

Key words: Perfect hashing, minimal perfect hashing, hashing, sparse matrix packing

1. INTRODUCTION

Hashing is a technique for building modifiable data structures that maintain efficient direct access retrieval times for all items in the structure. As items are added to the structure, an attempt is made to maintain \( O(1) \) (constant) access time at retrieval. This efficiency is achieved by placing each unique item that is added at a unique address in the structure. Several algorithms have been suggested, but no current hashing algorithm can maintain \( O(1) \) retrieval times over the lifetime of the data structure: eventually multiple items collect at identical addresses, and these collections must be handled in some way. All current algorithms for resolving these collections have a retrieval penalty which causes the retrieval efficiency to degrade beyond the ideal \( O(1) \) efficiency. Several perfect hashing algorithms have been created recently that do allow the \( O(1) \) retrieval of all items in the data structure as long as the items are static. These algorithms start with an original, static list of items, and use various techniques to build a hash function that is specific to that set of items. In a perfect hashing function, each item in the list can be retrieved with \( O(1) \) efficiency.

Interest in perfect hashing algorithms has risen recently with the advent of extremely large static databases, particularly those held on CD-ROMs. CD-ROMs are a non-writable medium, which guarantees static data. CD-ROMs also have slow track seek times, which make multiple accesses to the disk extremely frustrating for the user of the CD-ROM. Perfect hashing functions have been used in this environment to provide quick access to large medical dictionaries [1] and other word lists. The perfect hashing function allows any word to be accessed from the disk with only one disk access operation. Unfortunately, many current perfect hashing functions are severely limited in the number of words that can be processed by the algorithm.

Sprugnoli [2] introduced the first perfect hashing algorithm using relatively small word sets (10-12 words). He developed two techniques—the quotient reduction method and the remainder reduction method. Both methods rely on the algorithm’s ability to create a simple formula of the form \( h = (w - c_1) c_2 \) (quotient reduction method) or \( h = [(c_1 + n \cdot c_2) \mod c_3] / c_4 \) (remainder reduction method) (where \( h \) is the hash value, \( w \) is the word to be hashed, and \( c_1, c_2, c_3, \) and \( c_4 \) are constants calculated by the algorithm) for the word set in question. For larger word sets, Sprugnoli suggests a segmentation process to divide the list into smaller subsets. Sprugnoli’s methods are unique in that they do not require auxiliary information tables at retrieval time—only the four constants must be stored.

Jaeschke [3] developed a reciprocal hashing algorithm for the formation of minimal perfect hashing functions. His algorithm uses simple perfect hashing functions for retrieval, but word sets are limited to fewer than 40 words unless segmentation is used.

Cichelli [4] developed a straightforward minimal perfect hashing algorithm that can be used on word lists up to about 50 words in size. Cichelli’s method uses a table of 26 integers at retrieval time. The perfect hashing algorithm uses a brute-force technique to find a set of values for the table that allows each word to be hashed to a unique address. For larger word sets Cichelli also suggests segmentation.

Karplus and Haggard [5] generalized Cichelli’s algorithm using graph-theoretic methods to hash word sets of up to almost 700 words. Karplus and Haggard also note experimental results that indicate an extremely high efficiency for the algorithm (see Table 3). However, the algorithm has not been successfully applied to word lists greater than 700

281
words, and the words in the sample lists [6] seem to have been carefully chosen. For example, one list contains the words *Tannarice, thugslee* and *thyroglobulin*, but omits more common words such as *the, there* and *through*.

Chang [7] built an ordered minimal perfect hashing algorithm using the *Chinese remainder theorem*. An ordered perfect hashing function allows items in the static list to be arranged in any order (e.g. alphabetically) as the perfect hashing function is built. Chang's algorithm is unique in this respect.

Sager [8], also trying to improve on Cichelli's algorithm, used a graph-theoretic approach to create minimal perfect hashing functions of word sets as large as 512. Sager's work was later improved by Fox et al. [1] to form minimal perfect hashing functions of lists containing up to 1000 words. Both of these algorithms expand on Cichelli's table size considerably to handle large word sets. To increase the number of words hashed, both suggest segmentation.

Further discussion of perfect hashing functions can be found in [9] and [10].

Several issues have become prominent in perfect hashing research. A primary issue is the hash function's ability to form minimal item lists. In a minimal list, the items addressable by the perfect hashing function are arranged contiguously. Minimal lists save considerable storage if the items addressed by the perfect hashing function are large records of information. All the algorithms discussed previously produce minimal functions.

The order of the item list is sometimes important, for example if the item list needs to be accessible using several different techniques. Perfect hashing algorithms that allow arbitrary and specific orderings (e.g. alphabetically) of the item list once the perfect hashing function is formed are called ordered functions. Chang's algorithm [7] creates ordered perfect hashing functions, while most others do not.

Another issue important to perfect hashing functions is the amount of memory required by the function at retrieval time. All perfect hashing functions use a pre-calculated set of values to handle retrieval. The values are determined when the perfect hashing function is built. For example, Sprugnoli [2] calculates four constants \(c_1, c_2, c_3\) and \(c_4\) that are used at retrieval. Cichelli [4] calculates a table of 26 values that is accessed for retrieval. In Cichelli's algorithm, the first letter and last letter of the word to be retrieved are used as indexes into the table. The two values found in the table are summed together with the length of the word to form the hash address.

The purpose of Cichelli's algorithm is to find values for the table that cause no address collisions for all words in the static item list. The amount of space required for the table tends to grow with the number of items hashed. For example, Sager's algorithm [8] requires 4 bytes of storage in the table for each item in the static list of items.

Each perfect hashing algorithm has two completely different time efficiency aspects that must be monitored. One aspect is the retrieval efficiency. All perfect hashing algorithms have \(O(1)\) retrieval efficiency by definition. Perfect hashing functions take constant time to retrieve an item once the function has been created. Perfect hashing algorithms are also affected by the efficiency involved in actually building the perfect hashing function itself. For any algorithm, this cost for constructing a perfect hashing function occurs only once for a given set of items. For example, Cichelli's algorithm has an \(O(c^n)\) build time, while Sager's minimal cycle algorithm has an \(O(r^4)\) build time.

Finally, the complexity of the functions used for retrieval is important. Although all perfect hashing functions have an \(O(1)\) retrieval time, the amount of CPU time required for each retrieval should be minimized. Cichelli's algorithm has an extremely simple and efficient retrieval function. Sager and Fox use somewhat more complicated functions. As pointed out by Sager [8], Chang [7] uses retrieval functions that are extremely time-consuming, making them impractical.

In this article, a minimal perfect hashing function with an \(O(r^2)\) build time is presented. This algorithm uses a simple \(O(r^2)\) sparse matrix packing algorithm developed after extensive study of the run-time behavior of Cichelli's algorithm [11].

### 2. A SIMPLE PERFECT HASHING ALGORITHM

A typical application of a perfect hashing function is the \(O(1)\) retrieval of Pascal reserved words by a Pascal compiler. The Pascal reserved words are static and known in advance; this knowledge is used to build a specific perfect hashing function for the reserved word set. When a word is retrieved, the perfect hashing function is applied to the word. The use of perfect hashing functions can make token recognition for a compiler more efficient.

Several techniques are available to build perfect hashing functions. These algorithms cover a wide range of complexities, although the simpler algorithms often have significant disadvantages. For example, an extremely simple perfect hashing function for Pascal reserved words could be implemented using a 2-D array of integers as shown in Fig. 1. The first and last characters of the word being retrieved are used as indexes into the 2-D array, and the integer at the indexed location points to the word in the word list (normally the word list would contain information needed by the compiler to process the given reserved word).

In Fig. 1 a list of seven Pascal reserved words has been hashed. To retrieve the word *and* using this arrangement, the first letter *a* is used as the row index into the array, while the last letter *d* is used as the column index. At the location addressed by *a* and *d* is the value 1, which is used as a pointer to the record...
Hashing using sparse matrix packing

Fig. 1. A perfect hashing function using a 2-D array.

referenced by the word and in the word list. The retrieval time for any word in the word list is constant, regardless of the number of words in the array, giving this perfect hashing function an $O(1)$ retrieval time. This particular perfect hashing algorithm also has an $O(n)$ build time.

This arrangement works well to provide $O(1)$ access of the seven words in the example list. Unfortunately the memory usage for such a perfect hashing function is inefficient because many entries in the 2-D array are empty. In the example of Fig. 1, 659 of the 676 entries in the 2-D array are unused. One way to solve the space inefficiency of this perfect hashing algorithm is to pack the sparse 2-D array into a 1-D array. In order for the packing process to be acceptable, the retrieval time for the 1-D array must remain $O(1)$, as it is in the original 2-D array.

Cichelli's algorithm attempts to solve this problem by packing the elements of the 2-D array into a minimal (i.e. no empty spaces) 1-D array using a recursive backtracking procedure that acts on each element of the 2-D array individually. Because of its behavior, Cichelli's algorithm is essentially a 2-D sparse matrix packer. Unfortunately, Cichelli's algorithm is $O(c^e)$ during the packing process, and this time complexity makes the algorithm unusable except for extremely small item lists.

3. AN EFFICIENT SPARSE MATRIX PACKER

Investigating the run-time activities of Cichelli's packing algorithm [11] showed that the process could be significantly improved by treating the 2-D array as a set of complete rows rather than as individual entries, and then attempting to pack these rows as whole units. This approach yields a simple sparse matrix packing algorithm that is $O(r^4)$. This section describes the sparse matrix packing algorithm (SMP algorithm) developed from this insight. Although developed independently, the SMP algorithm is similar to Tarjan and Yao's row displacement algorithm [12], but their algorithm can be simplified when applied to perfect hashing. Fredman and Komlos [13] have also demonstrated $O(1)$ matrix packing techniques, but their time for construction of the function is $> O(r^4)$.

The SMP algorithm begins by separating the 2-D array into its individual rows. The rows are then copied into a 1-D array one at a time, starting with the row containing the most elements. The spacing of the elements in each row is preserved when the row is placed into the 1-D array, but each particular row can go anywhere. The starting location chosen for storing a particular row in the 1-D array is recorded in a row lookup table. The only constraint on the packing process is that no two values from different rows can occupy the same position in the 1-D array. A first fit algorithm determines row placement in the 1-D array.

The SMP algorithm starts with: (1) a sparse 2-D array such as the one shown in Fig. 1; (2) a 1-D array that will contain the packed array; and (3) a row lookup table (RLT) that will be used later to access the 1-D array during retrieval. The 1-D array holds the contents of the packed 2-D array, while the RLT holds the starting position of each row within the 1-D array. Packing begins by separating the array into its separate rows and rearranging them in decreasing order by the number of items in each row. This process is shown in Fig. 2. In Fig. 2, the E row 8 contains three entries, the C row contains two entries, etc.

Once the rows are sorted, the packing process begins. The row containing the most elements is placed into the 1-D array so that the first non-null value in the row is placed into the first location of the 1-D array. This process is shown in Fig. 3.

To allow the 1-D array to be accessed later, the RLT records how it was packed. The RLT contains one integer location for each row of the original 2-D array. The value $-3$ goes into this table for row $E$ because that is the offset for placing row $E$ in the 1-D array ($-3$ is used so that the first element in the row fills the first available slot in the 1-D array). The 1-D array and the RLT at this stage appear in Fig. 4.

Next the second most populated row of the original 2-D table is placed into the 1-D array—in this example, row $C$. This placement proceeds by determining the leftmost position in the 1-D array at which row $C$ can be placed into it without any of row $C$'s filled entries colliding with any of the filled entries already packed into the 1-D array. See Fig. 5. Row
The original array is separated into its individual rows, and the rows are sorted in order of decreasing number of items in each row.

C can be placed into the 1-D array without collision if its value 3 is placed at location 4 of the 1-D array. This placement requires that the row be offset by \(-14\) as shown in Fig. 5. The RLT and 1-D array are then updated as shown in Fig. 6. The remaining two rows are added in a similar manner. The result of the process is a 1-D array containing the non-empty entries of the 2-D array, along with the RLT. See Fig. 7.

To retrieve a word, its first letter is used to access a value from the RLT. This value points to the starting location of the row in the 1-D array. The value of the last letter of the word is then used to access the actual integer stored in the 1-D array, which in turn is used to access the word. For example, to retrieve the word _begin_ in the packed array, the letter \(b\) is used as an index into the RLT to get the value \(-7\). The letter \(n\) is converted to the integer 14.
Hashing using sparse matrix packing

Row E

\[
\begin{array}{cccccccc}
6 & 5 & 7 & 1 & 1 & 1 & 1 & 1 \\
6 & 5 & 7 & 1 & 1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

1-D array

\[ \Rightarrow \]

Offset placed in RLT

Fig. 8. The placement of row E if the word enter is included.

3.1. Sparse matrix packing algorithm

- Fill the 2-D array with values.
- Sort the rows of the array. The rows with the most entries should be at the top of the sorted list, and other rows should fill the list in decreasing order.
- Loop through all rows containing one or more values.
  - Attempt to place the current row at location one of the 1-D array.
  - While the new row's values collide with values already in the 1-D array, move the row down one position in the 1-D array and check for collisions again.
  - Once a non-colliding position for the new row is found, place the row in the 1-D array and update the RLT.
- Store the final RLT.

Because this example is small, it results in a 1-D array that is minimal (perfectly packed). As will be seen in Section 4, the 1-D array is not minimal when using the SMP algorithm on more realistic data lists. For example, if the word enter were added (as an eighth word) to the 2-D array shown in Fig. 1, then the 1-D array is no longer minimal because the entry for the R column in the E row would be stored at location 15 in the 1-D array. The locations between 7 and 15 in the 1-D array would be unused, as shown in Figs 8 and 9. The solution to this problem is addressed in Section 5.

Figure 10 summarizes the operation of the algorithm by illustrating the packing of all four rows from Fig. 1.

4. ANALYSIS OF THE SMP ALGORITHM

Determining the computational complexity of the SMP algorithm is straightforward. Assume an \( r \times r \) array contains \( r \) elements. Because the 2-D array contains \( r \) values, a minimally packed 1-D array containing the contents of this 2-D array will also require \( r \) elements. In an \( r \times r \) array containing \( r \) elements, each row will contain an average of one element. As each row is packed, every element in the row must be examined to determine if it collides with any existing values in the 1-D array. Also, in the worst case, there are \( r \) possible positions for each row to occupy in the 1-D array. Since each row can be considered to contain a single element, and since in the worst case \( r \) position for this row in the 1-D array must be examined, \( r \) comparisons must be made. This same operation must be performed for all \( r \) rows in the 2-D array. Therefore, \( r^2 \) comparisons must be performed to pack the 2-D array. The algorithm is \( O(r^2) \) in the worst case.

To test the SMP algorithm, randomly filled sparse 2-D arrays of varying dimensions were packed using the algorithm. Table 1 summarizes the results of these tests. In the table, a 2-D array of the size indicated was sparsely filled with the number of values indicated in the number of entries column. For example, for the 1000 \( \times \) 1000 array size with 1000 entries stored in it, the 1000 values between 1 and 1000 were

\[
\begin{array}{cccccc}
A & B & C & D & E & \ldots \\
1 & -7 & -14 & -3 & \ldots & \text{Row Lookup Table} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
6 & 5 & 4 & 1 & 3 & 2 & 1 & \ldots \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

1-D array

Fig. 9. An example of non-minimal packing.

\[
\begin{array}{cccccccc}
\end{array}
\]
placed into randomly chosen locations in the array. The time required to pack the array was recorded. The array was then cleared and again randomly filled. Ten runs were performed for each test set shown in the table, and the average of the times and spaces were taken. The average times indicated were produced using a PS/2 Model 70 using Turbo Pascal Version 5.0. The average 1-D array size column indicates the size of the 1-D array once the 2-D array was packed into it. The percentages at the end of each line show the percentage of unused values in the 1-D array. In all cases, the size of the RLT is equivalent to the length of one row of the 2-D array (e.g., in a 1000 x 1000 array, the RLT would contain 1000 integers—one integer for each row packed).

It may not be immediately obvious that 1000 x 1000 or 5000 x 5000 arrays have any application to perfect hashing. It will be shown that large arrays (e.g., 5000 x 5000) can be used for perfect hashing by using not just the first and last letter of the word to index the array, but instead using a function based on the first and last two or three letters of the word. For example, when the first and last pair of letters are used, 2-D arrays as large as 676 x 676 can be indexed.

Table 1 demonstrates several points about the SMP algorithm. First, the packing of the 2-D array down
Hashing using sparse matrix packing

1. IMPROVING THE SMP ALGORITHM

The SMP algorithm can be modified slightly to produce minimally packed 1-D arrays in cases where the number of elements in the array is less than or equal to the number of rows in the array. This revised algorithm is called the minimal SMP algorithm (MSMP algorithm). Instead of allowing values to be placed as far down as they would naturally fall in the 1-D array, the 1-D array is limited in size and out-of-bounds entries are moded back into that limited table size [14].

The example 6 x 6 array in Fig. 11 uses integer indexes for simplicity, but could just as easily be indexed using characters as shown in Fig. 1.

Using the SMP algorithm, the 2-D array of Fig. 11 when stored into a 1-D array would be non-minimal as shown in Fig. 12. Location 4 in the 1-D array is unused, and therefore represents wasted space and non minimal packing. This problem can be solved by limiting the size of the 1-D array to 6 (the number of values being packed), and by adding a mod function to the hashing function:

\[ \text{hash-value} = (\text{row_lookup_table}[\text{row_index}] + \text{column_index}) \mod \text{1-D_array_size} \]

By modifying the hashing function, the 1-D array and RLT would appear as shown in Fig. 13. For example, to locate the value in row 2, column 6, using Fig. 13, the value stored for row 2 in the RLT would be referenced. The value found at this position in the RLT is 3. This value in turn would be added to the column number desired (in this case 6) to obtain the value, 3 + 6 = 9, which would then be moded by

5. IMPROVING THE SMP ALGORITHM

The SMP algorithm can be modified slightly to produce minimally packed 1-D arrays in cases where...
the 1-D array size of 6 to obtain the result of 3. At location 3 in the 1-D array is the value 5, as expected.

Table 2 shows the effect of this modification on packing time and space efficiency. Its column headings are the same as those for Table 1 and the runs to generate the data were performed in the same manner as for the SMP algorithm. It can be seen in Table 2 that as long as the number of elements remains approximately equal to or less than the number of rows in the array, the array can be minimally packed. In terms of space, this outcome means that one integer (2 bytes) of storage (from the RLT) is needed to store a word with the MSMP algorithm when it is used for minimal perfect hashing. For example, if 5000 words are stored in a 5000 x 5000 array, the RLT will have 5000 entries (integers) in it, or 2 bytes per word being hashed.

The space requirements for the MSMP algorithm at build time are $O(r)$. The row and column index plus the value of each element suffice to represent the 2-D array.

### 6. APPLICATION TO PERFECT HASHING

As has been shown, these algorithms can be easily used for perfect hashing. Minimal or non-minimal 1-D arrays can be produced, depending upon whether time or space is more important. Words can be placed into a 26 x 26 element array, and the 2-D array can be packed down and later accessed by retaining only the 26-element RLT.

Unfortunately, the probability of finding a set of words that contain no pattern collisions under this algorithm decreases rapidly as the number of words increases, as it does for many other perfect hashing algorithms. For example, if the word list to be perfectly hashed in Fig. 1 had contained both been and begin, then a pattern collision would occur because both words start with a b and end with an n, and therefore map to the same location of the 2-D array. Table 3 shows the probability of pattern collision vs the number of words being placed into a 26 x 26 element array. Knuth [13] calls this phenomenon the birthday paradox [2]. For example, the array in Fig. 1 contains 676 positions. If two random words are to be placed into the array, the probability of them not colliding at the same location is $675/676 = 99.85\%$. Therefore, the probability of collision is 0.15\%. If three random words are placed in the array, then the probability of no collision decreases to $(674/676) \times (675/676) = 99.55\%$. Therefore, the probability of collision is 0.45\%.

The pattern collision problem occurs in many perfect hashing algorithms. For example, Sprugnoli [2] refers to the birthday paradox in his discussion of segmentation. Cichelli’s algorithm [4] cannot handle more than about 70 words (without resorting to segmentation) because of pattern collisions. Sager’s algorithm [8] is more immune to pattern collisions because of the increased complexity of the indexing functions used, but Sager shows that those pattern collisions that do arise must be handled by arbitrary manipulation of constants in the indexing functions.

One solution to the pattern collision problem in the algorithms described in this article is to use, for example, the first letter pair and the last letter pair (or the first and last triplet, etc.) to increase the size of the 2-D array used to hash the words (it is also possible to use even more complicated techniques, as advocated by Sager [8]—any pair of functions can be used to index into the 2-D array). Using letter pairs allows words to be placed into a 676 x 676 (26 x 26 = 676) array, which allows perfect hashing of 200 words with only a 4.3\% probability of a pattern collision, or 676 words with a pattern collision probability of 40\%.

When using even larger lists, letter triplets can be employed. For example, to perfectly hash 5000 words, the first and last letter triplet can be used. This strategy produces an extremely large 2-D array however, since $26 \times 26 = 17,576$. To solve this problem, mod functions can be used to reduce to size of the 2-D array. For example, if the first and last letter triplet's values are moded by 5000, then the 2-D array size becomes $5000 \times 5000$.

### 7. COMPARISONS

Table 4 compares the MSMP algorithm to other algorithms. The build order is the computational complexity of the algorithm at build time. The list size
is the example list size in the referenced article. The CPU time is the required to build the perfect hashing function for the word list on the machine specified in the machine column. The space column contains the number of bytes required (per item being hashed) to store the table needed to do retrieval using that perfect hashing function.

To create the entry for the MSMP algorithm in Table 4, a list of 5000 words (with three or more letters) containing no pattern collisions was chosen at random from the standard Unix word list (/usr/dict/words), and placed into a 5000 x 5000 element 2-D array. The two functions used as indexes to the 2-D array were:

row_index = (26 x 26 x (ord(s[length(s))] - 1)) + (26 x ord(s[length(s)] div 2)) + ord(s[1]) mod array_size.  
(1)

column_index = (26 x 26 x ord(s[length(s)])) + (26 x ord(s[length(s) - 2])) + ord(s[length(s) - 3]) 
× mod array_size.  
(2)

The actual first and last letter triplets of the words are not used because letter triplets such as ing and con are common and cause non-uniform distribution of values in the 2-D array. Other indexing functions were also successfully used.

The words were minimally packed, using the MSMP algorithm, in 283 s of CPU time on an IBM PS/2 model 70. Because a 5000 x 5000 element 2-D array was used, the RLT contains 5000 integers—one word (2 bytes) of storage for each row in the list. For comparison, Cichelli’s algorithm uses a 26-value table of bytes for the 40 word list, or 0.65 bytes per word hashed. Sager [8] requires two words of 2 bytes each in the table formed for each item hashed.

In comparison to other current algorithms, the MSMP algorithm is superior both in terms of the CPU time required at build time and the storage required at retrieval time.

8. LIMITATIONS

The ability of the MSMP algorithm to form minimal perfect hashing functions has been tested on word lists containing up to 5000 words. To handle 5000 words, the indexing functions of equations (1) and (2) were used. When a 10,000 word list was attempted using these same indexing functions, minimal packing was not achieved. Analysis showed the problem to involve a significant number of rows containing large numbers of entries. For example, if 10,000 values are randomly placed in a 10,000 x 10,000 array (as done to form the results in Tables 1 and 2), the row containing the most entries contains 10 filled elements. When actual words fill the 2-D array, however, many rows contain more than 50 elements when using the indexing functions of equations (1) and (2). This situation occurs because certain letter combinations are common in English words.

It is possible to improve the situation by using letter quadruplets, or by using more complicated indexing functions like those chosen by Sager [8]. The increased complexity of the indexing functions has the effect of distributing the words more evenly throughout the 2-D array. This solution may be impractical in some cases because it increases the amount of CPU time required for each retrieval.

As with other perfect hashing algorithms, the preferred solution for handling word lists of over 5000 words is segmentation of the word list. Segmentation also resolves pattern collision problems by placing colliding words into different segments. If space is not a factor, the SMP algorithm discussed can also be used to quickly produce non-minimal functions.

9. CONCLUSIONS

The MSMP algorithm can be used to replace other minimal perfect hashing algorithms, and has much better build time performance on extremely large word lists.

REFERENCES