MINIMAL AND ALMOST MINIMAL PERFECT HASH FUNCTION SEARCH WITH APPLICATION TO NATURAL LANGUAGE LEXICON DESIGN

NICK CERCONE, MAX KRAUSE and JOHN BOATES
Computing Science Department, Simon Fraser University, Burnaby, British Columbia, Canada

Abstract—New methods for computing perfect hash functions and applications of such functions to the problems of lexicon design are reported in this paper. After stating the problem and briefly discussing previous solutions, we present Cichelli's algorithm, which introduced the form of the solutions we have pursued in this research. An informal analysis of the problem is given, followed by a presentation of three algorithms which refine and generalise Cichelli's method in different ways. We next report the results of applying programmed versions of these algorithms to problem sets drawn from natural and artificial languages. A discussion of conceptual designs for the application of perfect hash functions to small and large computer lexicons is followed by a summary of our research and suggestions for further work.

1. INTRODUCTION

Skilful lexical design can engender good results in automated natural language understanding endeavours. Lexicons are an integral part of natural language systems (as well as many other systems, e.g. compilers). Although different authors writing about implementations of such systems have had different criteria for using particular lexicons, nevertheless similarities become evident in both the use and function of the lexical component of their systems. In particular, one manifest similarity ensues since lexical items serve as access to (corresponding) meaning representations. The experimental program reported in Cercone[4] explores the nature and computational use of meaning representations for word concepts in the context of a natural language understanding system. Word meanings are represented as extended semantic networks and are accessed via a tiny (approximately 200 words) lexicon.

It is interesting to note that many natural language system designers make extensive use of morphological analysis to avoid explicit storage of regularly inflected words and some irregular forms. Most systems rely on LISP (Allen[1]) to access word meanings through the word's property list. Small dictionaries with few syntactic features and simple semantic features are not penalised with this use of LISP. However, when the size of the dictionary and especially the complexity of lexical entries becomes great, more explicit control over the dictionary is desirable.

Advantages accrue when processing natural or artificial languages with fixed vocabularies of frequently used words by direct random access to items in the database. Perfect hash functions, a deterministic refinement of key-to-address transformation techniques, provide this single probe retrieval of keys from a static table. Given a set of \( N \) keys and a hash table of size \( r \geq N \), a perfect hash function maps the keys into the hash table with no collisions since the function locates each key at a unique table address. The loading factor (LF) of a hash table 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. A perfect hash function with LF \( \geq 0.8 \) is called an "almost minimal" perfect hash function.

Criteria for a good hash function include: (1) the hash address is easily calculated; (2) the loading factor of the hash table is high for a given set of keys; and (3) the hash addresses of a given set of keys are distributed uniformly in the hash table. A perfect hash function is optimal with respect to the uniform distribution of hash addresses in the hash table; adding minimality to the perfect hash function makes it also optimal with respect to the LF.

Perfect hash functions are difficult to find, even when almost minimal solutions are accepted. Knuth[16] 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.

Greniewski and Turski[12] used a non-algorithmic method for finding a perfect hash function to map the operation codes of the KLIPA assembler into an almost minimal hash table. Their
216 hash function took the form:

\[ H(k_i) = A \times k_i + B \]

where \( k_i \) is the \( i \)th key in the set of \( N \) keys and \( A, B \) are constants.

Sprugnoli[26] fashioned algorithmic methods to produce perfect hash functions of the form:

\[ H_q(k_i) = \text{floor} \left( \frac{(k_i + A)}{B} \right) \]  
\[ H_r(k_i) = \text{floor} \left( \frac{(A + k_i \times B)}{C} \mod D \right) \]  

where \( k_i \) is the \( i \)th key in the set of \( N \) keys and \( A, B, C, D \) are constants.

Jaeschke[15] devised Reciprocal Hashing to generate perfect hash functions of the form:

\[ H_j(k_i) = \text{floor} \left( \frac{A}{B \times k_i + C} \mod D \right) \]

where \( k_i \) is the \( i \)th key in the set of \( N \) keys and \( A, B, C, D \) are constants.

Sprugnoli's and Jaeschke's methods will only produce minimal perfect hash functions for limited sets of keys. Keysets larger than 15 keys must be partitioned into smaller segments for each of which a perfect hash function is computed. Both Sprugnoli's and Jaeschke's solutions are machine dependent since the number-theoretic properties of the machine character code representations of keys are used to guide the search for appropriate values of the hash addresses.

Cichelli[6] devised an algorithm (Algorithm 0) 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 hash function is machine independent because the character code used by a particular machine never enters into the hash calculation. The algorithm incorporates a two stage ordering procedure for keys which effectively reduces 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.

The objective of this research was to develop faster and more general algorithms for finding perfect hash functions of the general form of Cichelli. The cost of the combinatorial search for acceptable integer assignments to letters dictates the maximum size of key sets which Cichelli's method can process. Several heuristic search methods were investigated to accelerate the search, yet produce nearly optimal hash tables. Faster and more general algorithms of the form of Cichelli's could be used to find perfect hash functions to organise large dictionaries (50-70,000 items). Such lexicons would be useful in computational studies of natural language and artificial languages for programming and conversational terminal interactions.

Three algorithms for finding perfect hash functions were developed (Algorithms 1, 2 and 3). The data objects utilised in the algorithms were strings of characters of length \( P \) drawn from alphabet \( A \), the 26 lower-case English letters. The alphabetic ordering of \( A \) was accepted as the basis for lexicographic ordering of sets of keys. The algorithms were implemented in APL and Pascal, which store the keys as character arrays. The performance of each algorithm was evaluated.

2. DEVELOPMENT OF THE PERFECT HASH ALGORITHMS

The problem of finding faster and more general algorithms derived from the method of Cichelli was divided into three subproblems: (1) choosing a hash identifier which will uniquely identify members of a lexical key; (2) efficiently finding an assignment of integer values to letters which will map keys into a hash table without collisions; and (3) finding ways of enforcing or attaining a reasonable degree of the minimality of the solution.
2.1 Choosing a hash identifier

Optimally a hash identifier uniquely identifies each key so that each may be placed in a unique hash table location. The set of formal properties of lexical keys which could be used in a hash identifier include: the letters of the alphabet, their position of occurrence in the key, and key length. A given maximum key length \( P \) and alphabet \( A \) determine a space \( T \) of possible keys. If \( T' = \text{card}(T) \) and \( A' = \text{card}(A) \), then we can express the cardinality of the set of all possible keys as the sum of the number of keys of length \( P \) plus the number of keys of length \( P - 1 \) plus the number of keys of each of the lengths \( P - 2, P - 3, \ldots, 1 \).

\[
T' = A'^P + A'^{P-1} + \ldots + A'
= \sum_{i=P} A'^i
= A' \times (A'^P - 1)/(A' - 1)
= \Theta(A'^P)
\]

When \( A' \) becomes arbitrarily large, the limit of \( A'/(A' - 1) \) approaches 1. The resultant factor, \( (A'^P) - 1 \), reduces to \( A'^P \). Thus the key space grows at a rate polynomial in \( A' \) and exponential in \( P \).

When the occurrence of an alphabetic symbol, \( a_{ij} \), in one position, \( a_{i1} \), is treated differently from the occurrence in a different position, \( a_{i2} \), of the same symbol, the number of keys which can be distinguished is exactly the number of keys, \( T' \), in the space of keys, \( T \). When the hash function assigns the same value to a letter independent of the letter’s position in the key, the number of keys which can be distinguished is given by the expression:

\[
CH(A' + i - 1, i), 1 \leq i \leq P
\]

where \( CH(m, m) \) is the familiar “choose” function, defined as \( CH(n, m) = n!/(m! \times (n - m)! \). If \( A' = 26, P = 6 \), then the size of the key space is:

"Letter value dependent on position"

\[
\sum_{i=1}^{6} 26^i, 1 \leq i \leq 6
= \text{approx } 3.2 \times 10^8
\]

distinguishable keys

"Letter value independent of position"

\[
CH(26,1) + CH(27,2) + \ldots + CH(31,6)
= \text{approx } 9 \times 10^5
\]

distinguishable keys

Including a consideration for letter position in the hash identifier improves its representational power. There is no lexical key which cannot be uniquely represented by such a hash identifier.

Three procedures for specifying the hash identifier were implemented, including: (1) a previously defined hash identifier (Algorithms 0 and 2); (2) a hash identifier determined by an automatic procedure in the algorithm (Algorithm 1); and (3) a hash identifier specified by the user interactively (Algorithm 3).

2.2 Assigning associated letter values

Once a hash identifier has been defined, an efficient search must be organised to find an assignment of integer values to the letters which will map the keys into the hash table with no collisions.

A series of integer values must be chosen for assignment to the letters of the lexical keys. An easily generated integer series which guarantees distinct sums would have advantages but assignment of such values tends to decrease the hash table loading factor. An integer series which grows slowly and produces distinct addresses would map the hashkeys into a compact address space. Since no naturally occurring integer series investigated, e.g. powers of two or modified Fibonacci series, satisfies the requirements for assignment values, procedures for searching for an acceptable assignment of integers were developed.

In a search space viewed as a tree with three keys \( (N = 3) \), two letter positions included in the hash identifier \( (S = 2) \) and a maximum associated value of two \( (m = 3, M = \{0, 1, 2\}) \), the number of different assignments of integers to letters is \( (m^S)^N \), the number of leaf nodes in the tree (Fig. 1).
The number of integer values which is tested as an acceptable integer assignment during the search, \( m \), is the branching factor of the tree. It controls the extent of the search and whether or not a solution is possible. Procedures used to determine the upper bound of the search variable include: the user deciding on an upper bound, the algorithm deciding on a upper bound, and no upper bound being placed on the search variable.

An efficient search will lead to an acceptable assignment of associated values while generating as little as possible of the search tree. Backtracking (Algorithms 0, 1 and sometimes 3) and non-backtracking (Algorithms 2 and sometimes 3) search methods were used to find an acceptable assignment of integer values to letters in lexical keys. A non-backtrack search is preferable if the integer assignment made at any stage in the search is certain to not ultimately cause collisions. During a backtrack search the validity of all partial solutions is tested against the search predicate. If a partial solution \( (x_1, x_2, \ldots, x_i) \), \( 1 \leq i \leq N \), fails to satisfy the predicate, the subtree with this value of \( x \) as its root can be pruned. This pruning processes, called preclusion, avoids generating, for a value rejected at level \( i \) in the search tree, \( \sum_{j=1}^{i-1} m_j \), \( 1 \leq j \leq s - i \) full and partial solutions which have \( (x_1, x_2, \ldots, x_i) \) as an initial segment, where \( m \) is the branching factor of the search tree and \( s \) is the depth.

For each potential value eliminated, an exponentially growing subtree will be pruned. This justifies application of any polynomial-cost analysis that can be performed dynamically in the depth-first search which excludes values in the domain of the search variable from consideration. Preclusion is most effective when failure to satisfy the search predicate is discovered at a minimal tree depth. Fortunately, the frequency of occurrence of letter \( a_i \) is an excellent heuristic for predicting how likely it is that \( a_i \) occurs in a key which may collide with other keys. The sum (or product) of letter occurrences for one key is likewise an excellent predictor of how likely it is that a key will collide with other keys.

An ideal heuristic ordering strategy for the letters 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. This arrangement tends to occur when the keys are first ordered by sum of letter frequencies, then for each key the letters which have not occurred before are chosen in decreasing order of the frequency of occurrence. However, ordering the keys by sum of letter frequencies does not in general produce a strict ordering of letters to be assigned values, so between when an integer is assigned to a letter and when the hash address is tested, other hash addresses may have been added to the table. When collisions do occur, the program may have to backup and find new addresses for several keys. For example: the set of keys \( K = \{aa, ab, ac, ad, ae, af, ag, bd, cd, ce, de, fg\} \) after first ordering yields \( \{aa, ad, ac, ae, ab, af, ag, cd, de, bd, ce, fg\} \). The letter frequencies for \( K \) can be ordered into decreasing frequency of occurrence as \( (a, d, c, e, b, f, g) \). Although the key "cd" is determined as soon as key "ac" is placed in the hash table, the placement of "cd" is not tested until four intervening keys, ("ae", "ab", "af", "ag"), have been assigned hash addresses.

Given a permutation \( B = (a_1, \ldots, a_s) \) of the search variables, \( D(B) \) represents the number of keys, \( d_i \), whose hash addresses are newly determined when \( a_i \) is assigned a value. \( C(B) \) [essentially \( D(B) \) with \( c_i = d_i + 1, 0 \leq i \leq s \), so that \( c_i \) is the cost of visiting any node at level \( i \)] is a vector of coefficients, \( \langle c_1, \ldots, c_s \rangle \), for the series of terms \( \langle m^i, 0 \leq i \leq s \rangle \), which constitute the
Minimal and almost minimal perfect hash function search

weighted tree cost [WTC]:

\[
WTC = \sum_{i=1}^{s} c_i \times m^i, 0 < i < s
\]

\[
= c_0 + c_1 \times m^1 + \ldots + c_s \times m^s
\]

where the \( c_i \)'s are the associated costs at level \( i \).

The \( m \) factor in each of the terms in the total cost grows exponentially from the root. Cost will be minimised when the smallest possible values are assigned to the coefficients in the order \( c_s, c_{s-1}, \ldots, c_2, c_1 \), where \( c_s \) is as small as possible and \( c_1 \) is as large as possible. The smallest possible \( c_s \) is a value of two, which would derive from a letter, \( a_n \), having a frequency count of one so that it determines the value of only one key.

In the polynomial describing the size of the search tree, \( (m^i) \) is larger than the sum of the remaining terms:

\[
(1) \quad \sum_{i=1}^{s-1} m^i = m \times (m^{i-1} - 1)(m - 1), \quad 1 \leq i \leq s - 1;
\]

\[
(2) \quad m^s = m \times (m^{s-1})
\]

\[
(3) \quad m \times (m^{s-1}) > m \times (m^{s-1} - 1)/(m - 1); \quad \text{and}
\]

\[
(4) \quad m^{s-1} > (m^{s-1} - 1)/(m - 1).
\]

Since \( (m^i) \) will contribute most of the cost of generating the search tree, its coefficient (in the tree of minimal cost) MUST be the smallest which occurs in any of the \( s! \) possible permutations of the variables. A key is therefore sought which has at least one unique letter occurrence since only such a letter will come last in the ordering and still place a single key in the hash table. As a consequence, following from the necessity of placing letters which have a unique occurrence at the end of the ordering of letters, the optimal ordering will have all the keys which contain a unique letter occurrence at the end of the key ordering.

After the keys are ordered by the sum of the chosen letter frequencies, a second ordering, which rearranges the yet to be determined keys in the keyset depending on the partial assignment of integers to letters to this point, has the effect of making the coefficients of the \( m \) factors of the cost equation increase for smaller factors and decrease for the larger \( m \) factors. This process of moving the key forward in the ordering may be visualised as shifting the key's "weight" toward the root of the search tree, reducing the WTC for that tree if the order of letters induced by the new key order has larger weights nearer the root. [Recall that the sum of these coefficients along any root-to-leaf path in the search tree is the sum of the number of keys and the number of letters which occur in chosen positions, \( N + s \); this is a constant for a given problem instance.]

The optimal ordering of the search variables, \( B_{min} = (a_1, a_2, \ldots, a_s) \), is that for which WTC is a minimum. Generating all \( s! \) permutations of the letters would demonstrate that the optimal ordering is that for which \( D(B) \), and therefore \( C(B) \), has the largest lexicographic sort value.

Refining the second ordering (Slingerland and Waugh [25]) permits examination of fewer than \( s! \) permutations, speeding the computation of solutions. The key reordering process, and ultimately the ordering of the \( s \) letters which occur in chosen positions is modified 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".

Ordering procedures were used which arranged the keys by sum of letter frequencies, then reordered the keys so that any key whose hash value is determined by assigning the associated letter values already determined by previous keys is placed next (Algorithms 0, 1 and 2). In the example, the key "cd" would be placed next after "ce" is assigned a value so it would follow the key "ac". The ordering procedures were further refined to include ordering by product frequencies, key grouping, and unique letter appearance within key groups. These procedures are embodied in Algorithm 3 and details will be explained in the discussion of Algorithm 3.
2.3 Ensuring minimality of hash tables

Various methods were used to ensure a minimum loading factor 0.8. One heuristic which is applied in Algorithms 0, 1, 2 and 3 assigns the smallest associated values to those letters which occur most frequently in chosen letter positions, promoting small hash addresses for many keys. All hash addresses fall within the range \([\text{least} \ldots \text{least} + (N/0.8)]\); keys can always be mapped into addresses \([0 \ldots N/0.8]\).

When backtrack search is used to assign associated values, as in Algorithms 0 and 1, a LF of at least \(L\) can be achieved by simply limiting the size of the hash table to \(r = N/L\). The search procedure will then be forced to backtrack upon encountering any combination of letter values \((x_1, x_2, \ldots, x_i)\) such that for some \(k_i \in K: H(k_i) > N/L\). All values smaller than \(x_i\) in the domain of \(a_i\) have been excluded and any larger values of \(x_i\) will surely make \(H(k_i)\) greater than \(N/L\).

When backtracking is not used in the search, as in Algorithms 2 and 3, the LF is maximised by careful ordering of search variables and selection and testing of assigned integer values. Though non-backtracking algorithms produce solutions quickly, as the number of keys increases the LF tends to diminish. Algorithm 3 uses backtracking whenever the hash table is becoming too sparsely populated with keys.

3. DESCRIPTION OF THE ALGORITHMS

Cichelli's algorithm (Algorithm 0) 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 \(P \times CH(A', 2)\) 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 associated latter values.

Algorithm 0 employs a two-step ordering heuristic which first arranges the keys in decreasing order of the sum of frequencies of occurrence of the first and last letters. This ordering simply sorts the letters so that letters which occur most frequently are assigned integer values first. During the second step of the ordering any key whose hash value has already been determined, because its first and last letters have both occurred in keys previous to the current one is placed next in the list. This double ordering strategy arranges the static set of keys in such a way that hash value collisions will occur and be resolved as early as possible during the backtracking process.

Algorithm 0 produces perfect minimal hash functions.

Step 1. Compare each key against the rest. If two keys have the same first and last letters and the same length then report conflict and stop, otherwise continue.

Step 2. Reorder the keys by decreasing sum of frequencies of occurrence of first and last letters.

Step 3. 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.

Step 4. 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.

Algorithm 1, the first of the three new algorithms we developed, partitions the original set of keys into subsets according to their length, calculates perfect hash functions for each subset, then combines them to form one perfect hash function for the entire set. The complexity of each subproblem is at least linearly and often exponentially smaller than that of the overall problem, while the increase in the number of problem sets is linear, resulting in a marked reduction in computation. Which letter positions are chosen to identify each subset is recorded in a vector of Boolean values and a table is constructed which associates an integer value with each letter which occurs in a chosen position. For each subset an offset is maintained which keeps the subsets' hash addresses separate from those of any of the other hash functions, so that the subset hash functions can be fitted into a single hash table.

A procedure in Algorithm 1 automatically chooses, for each subset of keys of the same length, the smallest set of letter positions which will distinguish each key without regard for the
order of occurrence of letters within a key. The number of different subsets of letters is much smaller than the key space with the same maximum length. The algorithm which makes the choice of letter positions generates trial combinations of one position, then two positions, up to all \( P \) positions. When each trial combination is generated, it is tested for its ability to discriminate members of the set of keys. If no two keys have the same letter occurrences in the \( p \) selected positions, then the algorithm returns this trial combination as the solution and terminates; otherwise, the next combination is generated. The use of the cardinality of a subset as the upper bound on the range of associated values for the letters in chosen positions is a refinement of the choice of \( m \) in Algorithm 0.

Algorithm 1 order the keys in the same manner as Algorithm 0, but ties between keys with equal sums of letter frequencies are broken using Slingerland and Waugh's refinement of the second ordering.

Algorithm 1 incorporates a method of precluding the generation and testing of inadmissible combinations of associated values during the backtrack search process.

Algorithm 1 doesn't guarantee minimality, but a high loading factor is obtained by using a good heuristic choice of domain size for the associated letter values and by allowing the ranges of hash addresses for the subsets to overlap slightly.

**Step 1.** Sort the keys into order of increasing length and partition the keyset into subsets of keys which share the same length. Upon each of these subsets, perform the following steps.

**Step 2.** Choose the smallest set of letter positions such that no two keys of a subset have the same set of letters in the chosen positions.

**Step 3.** Order the keys in each subset using Cichelli's two ordering strategies with Slingerland and Waugh's refinement, to produce an approximately optimal ordering of the keys and of the letters which occur in the chosen positions.

**Step 4.** For each subset define the range of values associated with letters in the chosen positions using zero as the lower bound and the cardinality of the subset as the upper bound.

**Step 5.** Using a modified Cichelli backtrack search assign integer values to letters such that each key is mapped into a unique hash address in the subrange of the hash table defined by \([\text{offset} \ldots (p \times m)]\), where \( \text{offset} \) is the integer offset for the current subset of keys, \( p \) is the number of chosen positions, and \( m \) is the upper bound of the range of associated letter values.

**Step 6.** Add remaining unprocessed subsets sequentially to the hash table, allowing the different subsets to overlap somewhat by initialising the next offset by the number of keys which have been placed in the hash table (\( n \)), then finding the first open position (\( r, r \neq n \)).

**Step 7.** If any unprocessed subsets remain, return to Step 2; otherwise all keys have been placed in the hash table.

**Step 8.** Combine the subset perfect hash functions to make one perfect hash function for the entire keyset.

Algorithm 2 uses the key length and the first and last letters for the hash identifier, and the value of each letter is independent of position. If any two keys have all characteristics in common, this algorithm cannot be applied to the keyset. If the maximum key length is \( P \), then \( P \times CH(A', a) \) keys can be distinguished using Algorithm 2.

Algorithm 2 chooses an upper bound for each search variable from a naturally occurring integer series which has no pairwise sums among its elements. The upper bound is a solution, though not necessarily the best solution overall, and all lower potential associated values are tested in hopes of finding a better assignment of integer values. This method of assigning integer values avoids all backtracking yielding a rapid search algorithm. Algorithm 2 is not a backtracking algorithm in the classical sense, but an intelligently controlled enumerative one. Unlike backtracking search, this method never "undoes" partial results. Once a key is placed in the table, its address never changes.

Using the upper bounds for the associated values ensures that a solution will be found, but makes no guarantee that the loading factor of the resulting hash table will be acceptable (though it usually is for sets of a small number of keys), the search relies entirely on the good effects of the ordering of search variables to achieve a compact solution.

**Step 1.** Count the frequency of occurrence of each letter which appears in the first and last positions in the set of keys. Order the letters by decreasing frequency of occurrences.

**Step 2.** To each \( a \), in the ordered set of letters assign the upper bound \( F(i) \), where \( F(n) \) is a
series like a modified Fibonacci or powers of two. This assigns the smallest limiting values of the most frequently occurring letters, promoting the minimality of the resulting hash table.

**Step 3.** For each key $k_i$, $0 < i \leq N$, calculate $tval[i]$, the sum of the temporary values of the first and last letters plus the length of the key. Sort the keys on the sum, producing a list of keys ordered by the sum of the frequencies of their first and last letters; keys with the same combination of letters in these positions will be ordered by increasing length. During this ordering, Algorithm 2 ceases to function if any two keys have all characteristics in common;

**Step 4.** For each key $k_i$, $0 < i < N$, do the following: (a) if both the first and last letters in $k_i$ have been assigned values, continue with the next key; $k_i$ has been placed in the hash table previously; (b) if neither letter has been assigned a value, set the most frequent of them to zero; and (c) if only one letter $a_j$ has no assigned value, vary its associated value from zero to the upper bound until all the keys whose hash addresses are determined by this letter have been placed in open hash table locations. Each time the associated value is incremented, the function “check” is called which first changes all hash values which are affected by the current letter, then makes an $O(N^2)$ pass through the set of keys to determine whether any pair of keys have the same hash address; and

**Step 5.** Mark the current letter “tried” and continue at step 4 with the next key in the ordering. If there is none, we have a solution and the algorithm terminates.

The user interactively specifies a set of letter positions and whether or not to include the key length in the hash identifier when using Algorithm 3. The program then tests the user’s selection for key discrimination, inviting the user to try again if any two keys cannot be distinguished. Algorithm 3 takes into account the position of occurrence of letters and therefore has the greatest possible discriminatory power of the three algorithms developed. There is no set of distinct lexical keys which cannot be distinguished by Algorithm 3.

No upper bound is placed on the size of associated letter values.

Algorithm 3 incorporates a development and refinement of the non-backtracking enumerative search procedure of Algorithm 2. An improved version of search which performs a limited amount of backtracking when a solution has a low loading factor has been implemented. This promises to retain much of the speed of the current version while reducing the size of the hash tables. By analysing the occurrence of letters shared amongst the keys, Algorithm 3 eliminates many doomed choices of associated values, streamlining the search process.

**Step 1.** The user is prompted to supply two specifications: (i) the set of letter positions to be used in the hashing, and (ii) whether or not the key length is to be a part of the hash identifier.

**Step 2.** If any two words cannot be distinguished with the hash function as specified by the user, then report conflicting keys and return to step 1; if the hash identifier is acceptable, then continue with step 3.

**Step 3.** Count the number of occurrences of each letter in each position, then subtract one from each total. For each key, assign a value which is the product of the occurrence counts for the selected letters in this key. Those keys whose assigned values are zero must have at least one unique occurrence of a letter in some chosen position. Place these keys at the head of a list of keys with unique occurrences. Repeat step 3 for the non-zero keys until no more keys with unique letter occurrences are found. Keys selected in this process will follow all keys with no unique letter occurrences in the final ordering.

**Step 4.** Order the remaining keys, those with no unique letter occurrences, by decreasing product of their letter frequency counts.

**Step 5.** Form a group by first choosing the key nearest the head of the list which has the fewest letters with no assigned value ("new" letters); next, find all the keys whose hash addresses will be determined when the chosen key’s new letters are assigned integer values. Repeat step 5 until all keys have been chosen.

**Step 6.** Order the keys within each group so that for any two keys $k_i$ and $k_j$, if we calculate the set differences between the letters from chosen positions in each key $[D_{ij} := L(k_i) - L(k_j); D_{ji} := L(k_j) - L(k_i)]$, where $L(k_m)$ is the set of letters in chosen positions for $k_m$, then if $k_i$ precedes $k_j$ in the ordering, all letters in $D_{ij}$ will be assigned values before the last letter in $D_{ji}$ is assigned a value.

**Step 7.** For each key, determine which of its chosen letters will be the last to be assigned a value in the search. This letter’s value can be manipulated to place the key into an open hash
address. In cases where length is the only difference between neighboring keys, the distance back to a key which differs in letters is noted.

**Step 8.** Taking these noted letters in order, determine for the next letter which of its possible values are precluded by conflict with the hash addresses assigned to previous keys.

**Step 9.** Assign letter values. If a single key is being placed, its determining letter value is just the one which places it in the lowest possible open hash address. If the key is part of a group whose hash addresses are determined by assigning the current key's hash address, then choose the smallest possible value that maps all the keys into open hash addresses.

**Step 10.** If no two letters (and therefore no keys) remain, then the algorithm terminates. Otherwise, continue at step 8.

A second version of this algorithm has been written which allows backtracking whenever it is discovered, at step 10, that the hash table has become too sparse. In that case, step 10 is replaced by the following two steps.

**Step 10.** The loading factor \([LF]\) of the partial solution generated to this point is calculated. If \(LF\) is too small and the number of allowable backtracks (set by the user in step 1 in response to a prompt) is not exceeded, then proceed to step 11; otherwise continue with step 8; and

**Step 11.** For the latest group added to the table, determine which keys have the highest and lowest hash address; call them \(k_{\text{max}}\) and \(k_{\text{min}}\). Choose a letter from \(k_{\text{min}}\), say \(a_i\), which does not affect the hash address of \(k_{\text{max}}\, and increment the associated value of \(a_i\) by one. Reset to zero the associated values of all letters which were assigned after \(a_i\). Remove from the table all keys which were placed after the value of \(a_i\) was assigned, and adjust the sum of assigned letter values of each affected key. Place these keys and the head of the list of keys which have not yet been assigned a hash address. Adjust the order of letters which determine the hash addresses of groups of keys and return to step 8.

4. PERFORMANCE OF THE ALGORITHMS

Analytic comparison of the relative performance of backtracking algorithms is difficult (Knuth[17]). The execution time, maximum number of keys which can be processed in a reasonable time, and the loading factor of the hash tables produced by Algorithms 0, 1, 2 and 3 were compared. The number of basic operations of the algorithm and the memory requirements should be considered in algorithm expense. Krause[18] reports the number of times basic operations are performed by Algorithms 0, 1, 2 and 3.

The effective loading factor \([ELF]\) of a perfect hash function is defined as:

\[
ELF = \frac{N}{r + t}
\]

where \(N\) is the number of keys, \(r\) is the range of calculated hash addresses and \(t\) is the number of associated letter values stored. ELF is a realistic measure of the amount of memory needed for implementing each algorithm since associated value tables are an essential part of each algorithm and they differ in each algorithm. Utility \([U]\) is proposed as a further measure of relative performance

\[
U = N \times \frac{LF}{T}
\]

where \(N\) represents the number of keys to be hashed, \(LF\) is the loading factor and \(T\) is the search time in milliseconds. Larger values of \(U\) imply a greater degree of utility. This is an arbitrary measure; it does, however, reward compact solutions to large problem sets and penalises algorithms using excessive execution time.

Algorithms 0, 1 and 2 were implemented as Pascal programs [Pascal/UBC]; Algorithm 3 was written in APL. All programs were run on an IBM 4341 computer under the Michigan Terminal System [MTS] time-sharing operating environment. Timing comparisons were gathered via MTS system subroutines which return \(CPU\) time, in milliseconds. Time-sharing overhead, such as swapping, was excluded from totals so that time is time actually used to execute the algorithms.

Cichelli[6] reported the results of applying his algorithm (Algorithm 0) to five sets of keys: (1) the 12 three-letter month abbreviations; (2) the 31 most frequently occurring English words;
(3) the 34 ASCII control codes; (4) the 36 Pascal reserved words (including ‘OTHERWISE’); and (5) the 39 Pascal predefined identifiers (excluding ‘ODD’). The three recently developed algorithms were also tested on: (6) the 33 Basic keywords; (7) the 42 Algol-W reserved words; (8) the 64 most frequently occurring English words; (9) the 76 Pascal reserved words plus predefined identifiers; and (10)–(12) the 100, 200 and 500 most frequently occurring English words.

Table 1 presents the search times and loading factors obtained by running each algorithm on the test problem sets.

For Algorithm 1 the worst case computation of the procedure for choosing a set of letter positions which produce unique hash identifiers requires calculation exponential in key length. In the tested examples the execution time for this procedure was linear with respect to N, the number of keys in the keyset. The worst case analytic estimate for the ordering of keys to produce a beneficial ordering of the letters which appear in the chosen positions for Algorithm 1 was $O(N^3)$. For the small problem sets tested the times required for this ordering fell between $5 + 2 \times N$ and $3 \times N$. Surprisingly, 30,735 milliseconds were required to order 42 keys of length 3 in the 200 most frequently occurring English words [MFEW] compared to 42 milliseconds required to order 61 keys of length 4 in the same example. We hypothesize that the complexity of these problems is dependent on the way in which letters are shared among the keys.

Although Algorithm 1 does not guarantee minimal hash tables, it almost always produces minimal results for the set of keys as a whole. Allowing the ranges of the subsets to overlap promotes minimality even though the combined subsets are non-minimal. The most demanding test, the 200 MFEW produced a hash table with an LF = 0.97.

Algorithm 1 performs well as long as the largest subset contains approximately 50 or fewer keys. At this point the pattern of sharing letters among the keys of the subsets starts to affect the number of nodes in the backtrack search tree which must be examined. We conclude that the procedure for choosing a set of letter positions which produce unique hash identifiers is of marginal utility.

Algorithm 2 gives acceptable results for sets of up to 100 keys. Two major problems with Algorithm 2 are: (i) the loading factors of the solutions produced degenerate quickly for keysets

Table 1. Comparison of time ($T$) (in milliseconds) and loading factor [LF] for all four algorithms on some representative key sets

<table>
<thead>
<tr>
<th>Key Set</th>
<th>Algorithm 0</th>
<th>Algorithm 1</th>
<th>Algorithm 2</th>
<th>Algorithm 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 Three Letter</td>
<td>459</td>
<td>45</td>
<td>672</td>
<td>84</td>
</tr>
<tr>
<td>Month Abbrev.</td>
<td>LF=1.0</td>
<td>LF=1.0</td>
<td>LF=1.0</td>
<td>LF=1.0</td>
</tr>
<tr>
<td>31 Most Frequent</td>
<td>290</td>
<td>23</td>
<td>2466</td>
<td>1763</td>
</tr>
<tr>
<td>English Words</td>
<td>LF=0.97</td>
<td>LF=1.0</td>
<td>LF=0.94</td>
<td>LF=1.0</td>
</tr>
<tr>
<td>33 Basic Keywords</td>
<td>no</td>
<td>results</td>
<td>no</td>
<td>results</td>
</tr>
<tr>
<td>34 ASCII Control Codes</td>
<td>1833</td>
<td>41</td>
<td>6916</td>
<td>1993</td>
</tr>
<tr>
<td>LF=1.0</td>
<td>LF=1.0</td>
<td>LF=1.0</td>
<td>LF=1.0</td>
<td>LF=1.0</td>
</tr>
<tr>
<td>36 Pascal Reserved Words</td>
<td>579</td>
<td>29</td>
<td>5712</td>
<td>2609</td>
</tr>
<tr>
<td>LF=1.0</td>
<td>LF=1.0</td>
<td>LF=0.88</td>
<td>LF=1.0</td>
<td>LF=1.0</td>
</tr>
<tr>
<td>40 Pascal Predefined IDs</td>
<td>360441</td>
<td>30</td>
<td>242</td>
<td>3049</td>
</tr>
<tr>
<td>LF=1.0</td>
<td>LF=1.0</td>
<td>LF=0.89</td>
<td>LF=1.0</td>
<td>LF=1.0</td>
</tr>
<tr>
<td>42 Algol-W Reserved Words</td>
<td>no</td>
<td>results</td>
<td>no</td>
<td>results</td>
</tr>
<tr>
<td>44 Most Frequent</td>
<td>383</td>
<td>26619</td>
<td>2933</td>
<td></td>
</tr>
<tr>
<td>English Words</td>
<td>LF=1.0</td>
<td>LF=0.69</td>
<td>LF=1.0</td>
<td></td>
</tr>
<tr>
<td>76 Pascal Reserved + Predefined IDs</td>
<td>no</td>
<td>results</td>
<td>no</td>
<td>results</td>
</tr>
<tr>
<td>100 Most Frequent</td>
<td>10063</td>
<td>129073</td>
<td>5100</td>
<td></td>
</tr>
<tr>
<td>English Words</td>
<td>LF=1.0</td>
<td>LF=0.70</td>
<td>LF=0.96</td>
<td></td>
</tr>
<tr>
<td>200 Most Frequent</td>
<td>42055</td>
<td>1505228</td>
<td>6096</td>
<td></td>
</tr>
<tr>
<td>English Words</td>
<td>LF=0.97</td>
<td>LF=0.42</td>
<td>LF=0.70</td>
<td></td>
</tr>
<tr>
<td>500 Most Frequent</td>
<td>no</td>
<td>no</td>
<td>results</td>
<td>results</td>
</tr>
</tbody>
</table>
of more than 60 keys; and (ii) the mechanism used for distinguishing keys, like Cichelli's (Algorithm 0), is not adequate for many problem sets.

The refinements to Algorithm 2, which led to the development of a substantially different Algorithm 3, addressed these problems directly with moderate success with respect to problem (i) and complete success with problem (ii). Algorithm 3 outperforms the other algorithms generally and shows the greatest promise for further development. Algorithm 3 does require additional storage to maintain separate associated value tables for each letter position selected.

Table 2 summarises the relative utility for the four algorithms. Algorithm 1 produces spectacular utilities for keysets up to about 100 keys. When the sizes of the subsets produced by partitioning reach around 50 keys, however, these values decline rapidly. The utility of Algorithm 3 remains relatively constant for all test keysets.

5. APPLICATION OF PERFECT HASH FUNCTIONS TO NATURAL LANGUAGE LEXICON DESIGN

Retrieval methods usually assume equal likelihood of retrieval for each data item (Knuth[16]). Cichelli[6] pointed out the utility of perfect hash functions for use in compilers. It is well documented in the literature of lexicography (Carroll et al.[3], Dewey[10]) that this is not the case for the English language (or, presumably, for any natural language). We propose to make use of information about the frequency of occurrence of English words and judicious mix of common search and hash encoding techniques to provide an efficient organisational strategy for a natural language lexicon.

If the dictionary is formed by putting properties on LISP atoms (as is done in many natural language systems), the entire search is performed by a LISP system. Most implementations of LISP (Allen[1], pp. 275-277) use an "object list" to access atoms, usually implemented as hash buckets. A built-in general purpose hash function is provided which distributes the hash values of the complete set of keys in the dictionary (hopefully equally) among the hash buckets, each of which is searched sequentially. The access time is therefore dependent on the number of buckets and on bucket size. [The retrieval time is dependent on the actual distribution of the keys among the buckets. For any hash function, there exist some set of keys which will produce very uneven distributions. In the worst case, all keys will have the same hash value, so the average cost of a successful search would be N/2; for an unsuccessful search, the cost would be N (where N is the number of keys).]

In addition to this search for the atom name, the property list must be scanned for dictionary properties. If, for the majority of items in the lexicon, this is the only property on the property list, the time required for any lexical access is approximately equal to the hash encoding scheme time. Comparatively, the number of words with many properties remains insignificant and will not be considered.

Any desirable search technique can be imposed on an explicitly-stored dictionary. When we attempt to organise the lexicon in a way that minimises retrieval time, many factors affect our choices, such as the size of the lexicon and the need for secondary storage. Some design criteria, however, will improve the access time for any linear search algorithm of a natural language lexicon. One such design feature is to order the dictionary according to the relative frequency of the use of the letters in words (Cercone and Mercer[5]).

<table>
<thead>
<tr>
<th>N</th>
<th>Algorithm 0</th>
<th>Algorithm 1</th>
<th>Algorithm 2</th>
<th>Algorithm 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>1303.23</td>
<td>2000.00</td>
<td>65.93</td>
<td>50.42</td>
</tr>
<tr>
<td>31</td>
<td>1034.69</td>
<td>3100.00</td>
<td>16.71</td>
<td>70.42</td>
</tr>
<tr>
<td>50</td>
<td>1042.51</td>
<td>1042.51</td>
<td>49.33</td>
<td>49.33</td>
</tr>
<tr>
<td>34</td>
<td>6.64</td>
<td>1797.47</td>
<td>7.00</td>
<td>59.96</td>
</tr>
<tr>
<td>36</td>
<td>41.22</td>
<td>3571.43</td>
<td>7.50</td>
<td>60.50</td>
</tr>
<tr>
<td>39</td>
<td>1.14</td>
<td>10.00</td>
<td>9.90</td>
<td>10.00</td>
</tr>
<tr>
<td>40</td>
<td>10.00</td>
<td>10.00</td>
<td>10.00</td>
<td>10.00</td>
</tr>
<tr>
<td>42</td>
<td>10.00</td>
<td>1899.49</td>
<td>10.00</td>
<td>1899.49</td>
</tr>
<tr>
<td>44</td>
<td>1466.67</td>
<td>1466.67</td>
<td>53.04</td>
<td>53.04</td>
</tr>
<tr>
<td>44</td>
<td>10.00</td>
<td>10.00</td>
<td>10.00</td>
<td>10.00</td>
</tr>
<tr>
<td>44</td>
<td>344.09</td>
<td>2.33</td>
<td>59.73</td>
<td>2.33</td>
</tr>
<tr>
<td>76</td>
<td>2533.25</td>
<td>2533.25</td>
<td>62.97</td>
<td>62.97</td>
</tr>
<tr>
<td>100</td>
<td>20.26</td>
<td>0.78</td>
<td>66.10</td>
<td>0.78</td>
</tr>
<tr>
<td>200</td>
<td>6.26</td>
<td>0.00</td>
<td>42.25</td>
<td>0.00</td>
</tr>
<tr>
<td>500</td>
<td>20.62</td>
<td>20.62</td>
<td>20.62</td>
<td>20.62</td>
</tr>
</tbody>
</table>
One proposal we will explore here is to divide the dictionary into two or more parts to form dictionary hierarchies. This feature is most interesting when one considers the very high frequency of use of a very small number of words, but it is also important when one needs to consider how to divide a dictionary over different storage media. For example, 732 items comprise 75% of the words used in representative text. A possible three-level hierarchy would be 64 items that account for 50% of the words in the text, 688 items that comprise another 25% and the remainder that provide the final 25%. A hash into the first level of 64 words followed by a binary search of the second level (which on the average would require about 9 accesses), followed by a trie search of the third level would provide a very efficient search.

Lexicon storage is as crucial an issue as the retrieval of lexical information. Common structure sharing and morphological analysis contribute towards efficient space utilisation; certain dialects of LISP use various techniques, such as CDR-encoding, to reduce the representational overhead. The dictionary represented as a trie, Knuth[16], requires less space because letters are not repeated unnecessarily in successive words. Some representational overhead is incurred, however, by the required pointers.

The previous discussion has considered how to minimise the space required by the lexicon. We now present a short synopsis of some typical lexicon designs. For the purposes of this discussion we will consider lexicons that contain large quantities of information in three representative sizes: (i) small—500 entries or less; (ii) medium—1000-5000 entries; and (iii) large—10,000 entries or more.

Typically, a small lexicon gains little from complex organisation schemes. Our interactive implementation of Algorithm 3, however, can compute almost-minimal hash functions for most lexicons of small size. One drawback is that we have to store $26^S$ associated values when $S$ letter positions are selected, making this table's size the same order of magnitude as the dictionary itself. Of course, search time would be cut considerably, so the storage overhead might still be found acceptable.

Medium size lexicons need to be analysed differently; if the dictionary can fit in random access memory, a binary search would provide efficient access of items, supplemented by hash encoding into a mini-dictionary of the most common words. There is no space advantage using a trie structure because the overhead in associated pointers is high and there is little common spelling among so few words.

Another approach which utilises Algorithm 3 is illustrated in Listing 5.1. Satisfactory experimental results have shown that 500 words can be placed in a noncolliding hash table in under 20 seconds. Nevertheless the LF is only about 0.68 which we feel is unsatisfactory; increasing the LF results in a substantial increase in computation. When more than one hash function is used, an offset can be manipulated to start the next group of 500 words in the sparse part of the table occupied by the previous group of 500 words, typically resulting in a loss of about only 10% of storage space. In this example the medium sized lexicon is divided into group of 500 lexical items (more or less) and Algorithm 3 is applied successively, manipulating the OFFSET to interleave the 500 word pieces to effectively increase the LF to an acceptable level. Listing 5.1 illustrates this method using the 500 most frequently occurring English words divided into 5 groups of 100 words each (rather than the 2500 MPFW divided into 5 groups of 500 words due to space limitations). In Listing 5.1 the first 100 word chunk is fitted into a table of size 133; the first offset is set to 98 (because the first unused space is table location 100 and length is part of each hash function, thus 98 plus a length of 3 would place the next word in location 101), the index where the application of Algorithm 3 to the second 100 word lexical chunk begins to place items. This technique effectively makes use of unused spaces from previous applications of Algorithm 3. If the lexicon cannot fit into memory, it is appropriate to treat the medium size lexicon as a large lexicon.

Large lexicons typically require secondary storage, and the number of retrievals from secondary storage should be minimised. The favorable results from Algorithm 3 suggest including the 732 most frequent words in a single almost-minimal hash table, giving one-probe retrieval in 75% of the cases. The remaining 50,000 words could be mapped by a second hash function into 50 subsets of about 1000 words each and stored in secondary memory. In order to preserve the machine-independence of the algorithm, this second hash function could be based on the ordinal positions of letters in the alphabet rather than on the machine character code.
For each of these subsets, an almost-minimal perfect hash function could be computed, storing the associated values in the same secondary memory location as the lexical information itself. If the key sought is not in the table of most-frequent words, then a hash would be performed to select the proper second-level table from a secondary storage medium; this table would then be searched using its own perfect hash function. This organisation would allow retrieval of any key with three hash calculations and one probe of secondary memory.

6. CONCLUSIONS

Cichelli’s algorithm provides a useful alternative to numerical approaches to the search for perfect hash functions. We have found methods of extending the application of this “simple” approach to larger problem sets.

We have considered improvements along the lines of (i) hash identifier choice; (ii) partitioning of the problem set; and (iii) improved search methods. In our empirical study of Algorithms 0, 1, 2 and 3 we conclude that the heuristics employed in Algorithm 3 appear the most promising for future research although both Algorithms 1 and 2 perform significantly better than Algorithm 0 (Cichelli’s algorithm). The introduction of a limited backtrack in Algorithm 3 is an important contribution leading to almost minimal hash tables for large keysets (> 500 keys). Further analysis of the problem may reveal a better way of performing this limited backtracking.

Finally, we found three mathematical problems closely related to the search for perfect hash functions including: (i) harmonious labelling of graphs; (ii) graceful labelling of graphs; and (iii) additive bases. Krause[18] describes these problems.

Acknowledgements—We wish to thank Venkatakasi Kurnala and Paliath Narendran of Rensselaer Polytechnic Institute for their work on Algorithm 2. Thanks are also due Josie Backhouse for reading an earlier draft of this paper and to Carol Murchison for her extensive editing. This research was supported by the National Science and Engineering Research Council of Canada under operating Grant no. A4309 and by the Office of the Academic Vice-President, Simon Fraser University.

REFERENCES


```
*********** Listing 5.1: Interleaved Lexicon - 500 Most Frequently Used English Words. ***********

<table>
<thead>
<tr>
<th>RUN #</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*EXCEPTION SETTINGS 99999;
 clear ws
 wsize is 46172
LOAD PERFECT 300000
 saved 12125110 01/26/62
 wsize is 300124

HASHT (preorders data for associated value calculation)

WORDS TO BE MAPPED: c1
 (c1 is a variable containing the 1st 100 MFEN)

Letters to be used: 1 2 3 4
 (for assignment of associated values)

IS BLANK TO BE A CHARACTER (Y/N): N
 (or use last letter of word if appropriate)

IS LENGTH TO BE PART OF FUNCTION (Y/N): Y

ORDER BY PRODUCT OR MAXIMUM (P/N): P
 (product of letter frequencies or like Cicelli's)

THE DATA IN CORRECTED PREORDER FORM:

... the then these when she they there they made her more not been then
... than that what war was has has men man their this him him have
... say no for first shell would come can could not one on i in an
... any are well will would but out into from who to so not no its is at
... should be = is as your you who can any by the now now of it us a over
... soon with little do us all two have like such very about every great
... other which people

RUN 0
 (invoking the second ordering part - nonbacktracking)

BINDING STARTED AT 1982 7 20 14 4 36 19

TIME DURATION WAS 0 0 0 0 11 51 52

CPU SECONDS USED IN BASH IS 1.573

NUMBER OF TIMES THROUGH BASH MAIN LOOP IS 75

TERMINATION AFTER BACKTRACK 0

LETTERS USED 1 2 3 4

OFFSET USED 0

LETTER VALUES

<table>
<thead>
<tr>
<th>Letter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A'</td>
<td>4 14 54</td>
</tr>
<tr>
<td>'B'</td>
<td>9 20 0 0</td>
</tr>
<tr>
<td>'C'</td>
<td>10 0 25 72</td>
</tr>
<tr>
<td>'D'</td>
<td>21 0 16 39</td>
</tr>
<tr>
<td>'E'</td>
<td>18 0 40 0</td>
</tr>
<tr>
<td>'F'</td>
<td>19 0 33 41</td>
</tr>
<tr>
<td>'G'</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>'H'</td>
<td>6 0 29 12</td>
</tr>
</tbody>
</table>

HASH TABLE

<table>
<thead>
<tr>
<th>Letter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'J'</td>
<td>2 12 21 24</td>
</tr>
<tr>
<td>'K'</td>
<td>0 0 24 0</td>
</tr>
<tr>
<td>'L'</td>
<td>4 11 19 2</td>
</tr>
<tr>
<td>'M'</td>
<td>5 0 14 12</td>
</tr>
<tr>
<td>'N'</td>
<td>79 57 76 0</td>
</tr>
<tr>
<td>'O'</td>
<td>8 7 9 0</td>
</tr>
<tr>
<td>'P'</td>
<td>45 44 16 5</td>
</tr>
<tr>
<td>'Q'</td>
<td>0 0 6 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Letter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'R'</td>
<td>0 38</td>
</tr>
<tr>
<td>'S'</td>
<td>14 2 5 7</td>
</tr>
<tr>
<td>'T'</td>
<td>0 41 19 2</td>
</tr>
<tr>
<td>'U'</td>
<td>20 17 15 0</td>
</tr>
<tr>
<td>'V'</td>
<td>79 57 76 0</td>
</tr>
<tr>
<td>'W'</td>
<td>14 2 5 7</td>
</tr>
</tbody>
</table>

CPU SECONDS USED IN HASH IS 2.187

THE DATA IN CORRECTED PREORDER FORM:

... this then these when she they there they made her more not been then
... than that what war was has has men man their this him him have
... say no for first shell would come can could not one on i in an
... any are well will would but out into from who to so not no its is at
... should be = is as your you who can any by the now now of it us a over
... soon with little do us all two have like such very about every great
... other which people

RUN 0
 (invoking the second ordering part - nonbacktracking)

BINDING STARTED AT 1982 7 20 14 4 36 19

TIME DURATION WAS 0 0 0 0 11 51 52

CPU SECONDS USED IN BASH IS 1.573

NUMBER OF TIMES THROUGH BASH MAIN LOOP IS 75

TERMINATION AFTER BACKTRACK 0

LETTERS USED 1 2 3 4

OFFSET USED 0

LETTER VALUES

<table>
<thead>
<tr>
<th>Letter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A'</td>
<td>4 14 54</td>
</tr>
<tr>
<td>'B'</td>
<td>9 20 0 0</td>
</tr>
<tr>
<td>'C'</td>
<td>10 0 25 72</td>
</tr>
<tr>
<td>'D'</td>
<td>21 0 16 39</td>
</tr>
<tr>
<td>'E'</td>
<td>18 0 40 0</td>
</tr>
<tr>
<td>'F'</td>
<td>19 0 33 41</td>
</tr>
<tr>
<td>'G'</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>'H'</td>
<td>6 0 29 12</td>
</tr>
</tbody>
</table>

HASH TABLE

<table>
<thead>
<tr>
<th>Letter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'J'</td>
<td>2 12 21 24</td>
</tr>
<tr>
<td>'K'</td>
<td>0 0 24 0</td>
</tr>
<tr>
<td>'L'</td>
<td>4 11 19 2</td>
</tr>
<tr>
<td>'M'</td>
<td>5 0 14 12</td>
</tr>
<tr>
<td>'N'</td>
<td>79 57 76 0</td>
</tr>
<tr>
<td>'O'</td>
<td>8 7 9 0</td>
</tr>
<tr>
<td>'P'</td>
<td>45 44 16 5</td>
</tr>
<tr>
<td>'Q'</td>
<td>0 0 6 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Letter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'R'</td>
<td>0 38</td>
</tr>
<tr>
<td>'S'</td>
<td>14 2 5 7</td>
</tr>
<tr>
<td>'T'</td>
<td>0 41 19 2</td>
</tr>
<tr>
<td>'U'</td>
<td>20 17 15 0</td>
</tr>
<tr>
<td>'V'</td>
<td>79 57 76 0</td>
</tr>
</tbody>
</table>

CPU SECONDS USED IN HASH IS 2.187

THE DATA IN CORRECTED PREORDER FORM:

... this then these when she they there they made her more not been then
... than that what war was has has men man their this him him have
... say no for first shell would come can could not one on i in an
... any are well will would but out into from who to so not no its is at
... should be = is as your you who can any by the now now of it us a over
... soon with little do us all two have like such very about every great
... other which people

```
Minimal and almost minimal perfect hash function search

Listing 5.1. U-kiwd...r)

<table>
<thead>
<tr>
<th>Hash Function Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>(binding the next 100 MFEM into the hash table offset 98 places)</td>
</tr>
<tr>
<td>Binding started at 1982 7 20 14 7 33 114</td>
</tr>
<tr>
<td>Time duration was 0 0 0 0 0 22 926</td>
</tr>
<tr>
<td>CPU seconds used in hash is 1.46</td>
</tr>
<tr>
<td>Number of times through hash main (loop is 77)</td>
</tr>
<tr>
<td>Termination after backtracking 0</td>
</tr>
<tr>
<td>Letters used 1 2 3 4</td>
</tr>
<tr>
<td>Offset used 98</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hash Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash values</td>
</tr>
<tr>
<td>'A' 21 0 2 22</td>
</tr>
<tr>
<td>'B' 0 0 11 0</td>
</tr>
<tr>
<td>'C' 5 0 12 45</td>
</tr>
<tr>
<td>'D' 14 0 21 31</td>
</tr>
<tr>
<td>'E' 44 1 5 8</td>
</tr>
<tr>
<td>'F' 4 53 9 76</td>
</tr>
<tr>
<td>'G' 7 43 17 64</td>
</tr>
<tr>
<td>'H' 6 23 36 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Words to be hashed: c3</td>
</tr>
<tr>
<td>Letters to be used: 1 2 3 4</td>
</tr>
<tr>
<td>Is blank to be a character (Y/N): N</td>
</tr>
<tr>
<td>Is length to be part of function (Y/N): Y</td>
</tr>
<tr>
<td>Used by method: UK maximum (Y/N): P</td>
</tr>
<tr>
<td>CPU seconds used in hash is 2.256</td>
</tr>
<tr>
<td>The data in corrected preorder form:</td>
</tr>
<tr>
<td>set let letter cent went want water battle general spot something men</td>
</tr>
<tr>
<td>several gave done certain forty conditions given five find situation</td>
</tr>
<tr>
<td>both side mind morning soon seen means coal steel state statement stand</td>
</tr>
<tr>
<td>call either early tell him believe military nation small brought front</td>
</tr>
<tr>
<td>city fighting night doing whole though children thing taken gun full</td>
</tr>
<tr>
<td>during books took hand high amount among every dear and enough course</td>
</tr>
<tr>
<td>four large big until asked use used saw law why whose eyes eyes left</td>
</tr>
<tr>
<td>next order interest ago off six it's known labor young almost himself</td>
</tr>
<tr>
<td>really capital purpose service necessary themselves</td>
</tr>
</tbody>
</table>

Listing 5.1. U-kiwd...r)

<table>
<thead>
<tr>
<th>Hash Function Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>(binding the next 100 MFEM into the hash table offset 195 places)</td>
</tr>
<tr>
<td>Binding started at 1982 7 20 14 7 33 114</td>
</tr>
<tr>
<td>Time duration was 0 0 0 0 0 22 926</td>
</tr>
<tr>
<td>CPU seconds used in hash is 1.497</td>
</tr>
<tr>
<td>Number of times through hash main (loop is 79)</td>
</tr>
<tr>
<td>Termination after backtracking 0</td>
</tr>
<tr>
<td>Letters used 1 2 3 4</td>
</tr>
<tr>
<td>Offset used 195</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hash Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash values</td>
</tr>
<tr>
<td>'A' 20 2 24 0</td>
</tr>
<tr>
<td>'B' 3 0 39 0</td>
</tr>
<tr>
<td>'C' 0 0 59 0</td>
</tr>
<tr>
<td>'D' 3 0 75 0</td>
</tr>
<tr>
<td>'E' 20 0 32 5</td>
</tr>
<tr>
<td>'F' 0 0 79 0</td>
</tr>
<tr>
<td>'G' 2 0 13 50</td>
</tr>
<tr>
<td>'H' 50 24 0 15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Words to be hashed: c4</td>
</tr>
<tr>
<td>Letters to be used: 1 2 3 4</td>
</tr>
</tbody>
</table>
********** Listing 5.1. (Continued...)

> IS BLANK TO BE A CHARACTER (Y/N): N
> IS LENGTH TO BE PART OF FUNCTION (Y/N): Y
> ORDER BY PRODUCT OR MINIMUM (P/M): P
> CPU SECONDS USED IN HASH IS 2.294
> THE DATA IN CORRECTED PRECEDING FORM:

```
  file sent send cents seems seems need held soldiers hands told months
  fell called land head help lost past party common committee complete
  moment free save love truly true pounds miles lines times hear nor hour
  hours keep kind face further turned perhaps point makes having making
  saying cause real result value rather future building boys better view
  knew know wish wrong thousand show whether show flour anything already
  along became broken afternoon price special often were able more back
  wheat dinner office ask boy 'm girl idea above enemy quite action
  itself number twenty subject beginning different yesterday impossible

BIND 395
  ( binding the next 100 HPFH into the hash table offset 395 places )
BINDING STARTED AT 1992 7 20 14 42 107
TIME DURATION WAS 0 0 0 0 3 984
CPU SECONDS USED IN HASH IS 1.580
NUMBER OF TIMES THROUGH HASH MAIN LOOP IS 82

TERMINATION AFTER BACKTRACK 0
LETTERS USED 1 2 3 4
OFFSET USED 395

LETTER VALUES
-'A' 0 5 13 18 'I' 0 25 11 15 'S' 0 0 18 43
-'B' 94 40 0 46 'K' 26 0 31 49 'T' 2 81 14 3
-'C' 1 52 30 32 'L' 6 51 6 9 'U' 0 25 0 11
-'D' 24 70 0 4 'M' 7 0 15 6 'V' 30 0 24 18
-'E' 53 0 0 1 'N' 5 30 1 29 'W' 0 63 0 2
-'F' 0 36 9 36 'O' 0 0 2 31 'X' 0 83 0 25 1
-'G' 43 0 44 0 'P' 2 27 75 12 'Y' 0 45 0 8
-'H' 0 59 0 27 'R' 52 0 0 0

HASH TABLE
  0 THE 4 THEN 5 THESE 6 WHEN 7 SHE .......
  149 WHICH 192 HOOT 193 MONEY 184 POWER 145 PART
  131 MAKE 132 FROM 133 PEOPLE 134 SAYS .......
  361 THEMSELVES 362 FEET 363 SENT 364 SEND 365 CENTS
  366 SEEM 367 SEEMS 369 NEED 369 HELD 316 SENDS 311 TOLD
  312 MONTHS 313 SOLDIERS .....
  327 TRUE 328 OFF 329 SAVE 330 LOVE .......
  401 BOYS 446 ASK 451 IMPOSSIBLE

HASH
WORDS TO BE HASHED: c5
  ( c5 is a variable containing the 5th 100 HPFH )
LETTERS TO USE: 1 2 3 4
IS BLANK TO BE A CHARACTER (Y/N): N
IS LENGTH TO BE PART OF FUNCTION (Y/N): Y
OFFICERS CONFLICTS WITH OFFICIAL
WOULD YOU LIKE TO TRY A DIFFERENT ROUTE: #
  ( we use the same hash function )
ORDER BY PRODUCT OR MINIMUM (P/M): P
CPU SECONDS USED IN HASH IS 2.728
THE DATA IN CORRECTED PRECEDING FORM:
road read reason tea sea heard near hold needs news poor political
products position pretty that's cost change thank change mean loan
coming remember comes together ten month clear plan truth court pound
third thirty child drive dollars following received glad primer company
trouble ground note modern alone among began family national hero
received care wanted worth hundred guns built some list million didn't
air force looking weak street behind department second reports flight
electric eight others effort official offensive house indeed yes cars
room can't opinion except living school taking unless evening however
o'clock perfect question important knowledge

BIND 395
  ( binding the next 100 HPFH into the hash table offset 395 places )
BINDING STARTED AT 1992 7 20 14 12 42 107
TIME DURATION WAS 0 0 0 0 3 672
CPU SECONDS USED IN HASH IS 1.598
NUMBER OF TIMES THROUGH HASH MAIN LOOP IS 80
KRYPTOMIN AMUX KRIPTOMIN
LETTERS USED 1 2 3 4
OFFSET USED 395
```
Listing 5.1. (Continued...)

> LETTER VALUES
> 'A'  8  5  0  6  'I'  8  22  28  3  'R'  8  2  43  7
> 'B'  71  0  8  8  'K'  40  0  87  44  'S'  7  8  11  7
> 'C'  3  13  33  0  'L'  13  20  0  6  'T'  0  10  40  9
> 'D'  24  0  30  3  'M'  11  31  19  77  'U'  34  32  22  33
> 'E'  40  1  6  5  'N'  12  0  8  12  'V'  0  34  40  12
> 'F'  23  0  49  51  'O'  19  0  8  38  'W'  20  0  88  3
> 'G'  21  0  21  0  'P'  0  9  27  33  'X'  0  6  0  0
> 'H'  0  9  36  0  'Q'  55  0  0  0  'Y'  78  0  0  33
> ...  0  36  0  26

> HASH TABLE
> THE  1 THEY  5 THESE  6 WHICH  7 SHE  ... .
> . . . 100 WHICH 102 MOST 103 MONEY 104 POWER 105 PART
> . . . 401 BOYS 402 ROAD 403 READ 404 REASON 405 TEA
> . . . 406 ASK 407 SEA 408 HEARD 409 NEAR 410 HOLD 411 IMPOSSIBLE
> 412 NEEDS 413 HEAVY  ... .
> . . . 501 KNOWLEDGE 502 EFFORT

TOTAL TIME IS: 5 HASHES - 11.275 SECONDS
5 RENDS - 7.636 SECONDS
TOTAL - 18.911 SECONDS

LOADING FACTOR IS: 540/528 = .997

OFF
EXECUTION TERMINATED 16:10:51 PM

Listing 5.1. Interviewed Lexicon - 500 Most Frequently Used English Words.++++++++++