# LEXICON DESIGN USING PERFECT HASH FUNCTIONS

Nick Cercone, Max Krause, John Boates

Department of Computing Science
Simon Fraser University
Burnaby, British Columbia, CANADA V5A 1S6

## 1. Introduction

The research reported in this paper derives from the recent algorithm of Cichelli (1980) 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

A minimal perfect hash function is one which provides single probe retrieval from a minimally-sized table of hash identifiers [keys]. Cichelli's hash function is machine-independent because the character code used by a particular machine never enters into the hash calculation.

Cichelli's algorithm uses a simple backtracking process to find an assignment of non-negative integers to letters which results in a perfect minimal hash function. Cichelli employs a twofold ordering strategy which rearranges 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. This double ordering provides a necessary reduction in the size of the potentially large search space, thus considerably speeding the computation of associated values.

In spite of Cichelli's ordering strategies, his method is found to require excessive computation to find hash functions for sets of keys with more than about 40 members. Cichelli's method is also limited since two keys with the same first and last letters and the same length are not permitted.

Alternative algorithms and their implementations will be discussed in the next section; these algorithms overcome some of the difficulties encountered when using Cichelli's original algorithm. Some experimental results are presented, followed by a discussion of the application of perfect hash functions to the problem of natural language lexicon design.

## 2. Search Strategies for Perfect Hash Functions

### 2.1. Perfect Minimal Hash Functions

According to Knuth (1973) a good hash function should satisfy two basic requirements: (i) the calculation of the hash value for each key should be very fast; and (ii) the function should minimize collisions. The type of hash function used by Cichelli in indeed calculated quickly since it consists of the sum of three positive integers (assumming that we find the length of each key as part of the recognition process). Because Cichelli's algorithm finds a perfect hash function, collisions are eliminated entirely, but not without expending considerable computational effort to find such a hash function for a given set of keys. Knuth (1973) estimates that only one in ten million functions is a perfect hash function for mapping 31 keys into 41 locations. The minimality requirement further complicates this computation.

The basic algorithm consists of a backtrack search of the solution space of assignments of integers to the letters which occur as first or last letters of keys. The complexity of this problem grows exponentially in the number of keys which share letters in the chosen positions. Cichelli therefore uses a two-step ordering heuristic which limits the size of the search space. The static set of keys is first arranged in decreasing order of the sum of the frequencies of occurrence of their first and last letters. This ordering ensures that the most frequent letters are the first to be assigned integer values. In the second step, the order of the key list is modified so that any key whose hash value is determined (because its first and last letters have both occurred in keys which precede the current one) is placed next in the list. This double ordering has the effect of forcing conflicts to occur early in the backtrack search, thereby pruning large branches of the potential search tree.

We have extended Cichelli's efforts in two major ways. First we have devised methods which find solutions significantly faster, enabling us to accommodate the much larger number of keys necessary for application of this method to natural language lexicons. The second extension is to make the method more generally applicable by including a procedure which finds the minimum set of letter positions which distinguishes each word from the rest.

The following is an outline of Cichelli's algorithm:

ALGORITHM 0:

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.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

©1981 ACM 0-89791-064-8/81/0500-0069 $00.75

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.

Two basic alternatives to Cichelli's algorithm, with several variations of each, are reported, followed by a discussion of the performance of the algorithm on sample sets of keys.

One approach is to adapt Cichelli's algorithm to find minimal perfect hash functions for each subset of keys of the same length. This eliminates the use of the key length as part of the hash function. In order to combine the separate hash tables for each subset into one table for the entire set, we need only add an offset to the hash function for each subset. A separate table of associated values is kept for each subset of the keys. The reduction in execution time for N keys partitioned into S equal-sized subsets is

O((C exp N) - (S * (C exp (N/S))))

for some constant C. Additional overhead includes storing the value of an offset and an assignment of integers for each subset.

A useful addition to this method, and to any of the others, would be to generalise the selection of letter positions, eliminating the need to delete one of any pair of words which have the same length and the same first and last letters. The process of finding a minimal set of letter positions which distinguishes each key is exponential in the maximum key length, i.e., O(2 exp M), where M is the maximum key length. There are (2 exp M) possible combinations of letter positions to choose from, and each time we consider one of the possible combinations we must ensure that none of the words have the same set of chosen letters. This calculation adds a cost factor of (N exp 2)/2 (where N is the number of keys), giving us

O((N exp 2) * (2 exp M)).

This selection process is clearly an expensive addition to any of our algorithms and might be better left to the specification of the user. In either case, there is certainly some set of positions which distinguishes each key (if we take into account the ordering of the letters), ensuring that we can place all the keys in the hash table.

We now give an informal outline of this modified version of Cichelli's algorithm.

ALGORITHM 1:

Step 1: sort the keys into ascending order, by length, in order to partition the set of keys into subsets of keys which share the same length. With each of these subsets do the following steps.

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

Step 3: Employ Cichelli's two ordering strategies, with Slingerland and Waugh's refinement, to produce an (approximately) optimal ordering of the keys and, therefore, of the letters which occur in chosen positions.

Step 4: make the upper bound of the range of associated values equal to the number of keys in the current subset. The lower bound of this range is zero for every subset.

Step 5: use Cichelli's backtracking procedure to place each key of the current subset in the subrange of the hash table defined by [offset...(p*m)], where each offset is the 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: if any unprocessed subsets remain, adjust the offset of the next subset: initialise the offset to the number of keys which have already been placed in the hash table, say n, then find the first open position r, r >= n. This is the new offset for the next subset..

Step 7: if any unprocessed subsets remain, return to step2; otherwise all keys have been placed in the hash table and the algorithm terminates.

If p letter positions are chosen for the subset of keys of length i, then the hash value is of the form:

```
hash value = offset(i) +
        assoc value(i, key(1st selected letter pos) +
        assoc value(i, key(2nd selected letter pos) +
                              :
                              :
        assoc value(i, key(pth selected letter pos)
```

For each key length a record is kept of the offset, the selected letter positions, and the associated letter values.

Algorithm 1 will produce the maximum cost reduction in a situation where the set of keys is partitioned by length into several subsets, each of a relatively small size. This algorithm's complexity is now dominated by the size of its largest subset, rather than by the cardinality of the set of keys as a whole; the minimum size of the largest subset occurs when the keys are most evenly distributed by length. Among the 200 most frequently-occurring English words, 80% have lengths of 3, 4, or 5. The largest of these subsets is words of length 4, containing about one third of the words in this rank. The maximum word length in the rank is 9.

Although our experimental results are inconclusive thus far for this algorithm, we expect that it will be no more than a factor of S (the number of different word lengths which occur in a set of keys) faster than Cichelli's original algorithm. This represents a significant improvement but not enough to allow large sets of keys (500-1000) to be accommodated.

This algorithm generalises the process of choosing letter positions to searching for a minimal set of letter positions which will distinguish all keys - a maximally

independent set of letters. For keys of only one letter, only one position is used. Keys of two letters will often need to use both letters, etc. This problem reduction approach requires additional storage for indexing tables which indicate letters used in associated value computation. The cost to maintain this information is bounded by O(N*M) storage locations for M letter positions used in the hash function and N keys.

Associated value assignments need to be made for each length subset of hash keys. If we have S subsets, the increase in storage allocations will be S*26.

Two related difficulties in Cichelli's algorithm are eliminated with this first algorithm. Keys such as 'his' and 'has' would never be permitted to have 'h' as the key's first selected letter and 's' as the key's last selected letter, since their key lengths are the same (this would also be true of 'label' and 'level'). Also, a key such as 'loop' could be kept distinct from a key such as 'pool' (as could 'on' and 'no') through the use of a constant multiplier for the key's last selected letter position (see below), or by maintaining a separate vector of associated letter values for each position.

### 2.2. Almost-Minimal Perfect Hash Functions

In practise, almost-minimal perfect hash functions compare favourably with minimmal perfect hash functions. An almost-minimal perfect hash function is one which leaves no more than 20% of the hash table locations unused. The next algorithm searches effectively for almost-minimal perfect hash functions which prove to be particularly well-suited for application to the natural language lexicon.

The essential part of the problem of finding a perfect hash function is choosing a method of assigning associated values to symbols of the alphabet. To arrive at a perfect hash function, the assignment should be done efficiently and should not lead to collisions.

As a simple example, consider the following assignment of associated values. The symbols of the alphabet A, B, ..., Z are assigned the values 1, 2, ..., 26 respectively. The hash function applied to the word 'program', for example, returns

hash value = length('program') + av('p') + av('m')
= 7+16+13 = 36

It is clear that if the word 'program' appears in a set of less than 30 keys and this hash function is used, then the hash table cannot be minimal or even almost-minimal, by our standards. Note also that there are many combinations of associated values and word length which give us a hash value of 36: an 8 letter word whose first and last letters are 'n', or a 4 letter word whose first letter is 'r' and last letter is 'n', for instance. We are led to the conclusion that a randomly chosen simple assignment of a series of distinct values to letters will not generally provide a good solution to our problem.

We have investigated some natural series which produce distinct values and whose elements, when added, produce distinct sums. One such series is the following:

$$F(1) = 0$$
$$F(2) = m$$
$$F(3) = 3m$$
$$\vdots$$
$$F(n) = 2*F(n-1) + F(n-2),\ n>3$$

The series is 0, m, 3m, 7m, 17m, 41m, .... If we choose m to be the length of the longest word and assign the values to letters in decreasing order of their frequencies of occurrence in the list of keys, we obtain all desired distinct values. This assignment method ensures that all pairwise sums of associated values are distinct as well. The difficulty in assigning associated values in this way is that the magnitude of the values rises very rapidly, even if the multiplicative factor m is small; with m=1, the tenth most frequent letter will get a value of 1731, which obviously defeats our purpose of producing almost-minimal sized hash tables.

Series of this nature are useful, however, in limiting the magnitude of associated values. Small hash values are convenient and lead to minimality. In order to keep hash values as small as possible, we must allow letters to share associated values. This possible duplication of values reduces the advantage of assignment of series of distinct associated values, since we again have to check for conflicts with each new vector of assigned values. Nevertheless, because we have assigned the smallest limiting values to the most frequently-occurring letters (those which are most likely to be involved in collisions), hash values do tend to remain small in the first 85-90% of the list of keys. After this there usually appears a series of increasingly large gaps which reduce the loading factor to about 42% for 200 keys.

Two series which produce distinct pairwise sums have been used in an implementation of Algorithm 2 [outlined below] based on the preceding ideas. The first series utilised is the powers of 2, which has the property that the sums of any two different sets of distinct members of the series are different. The series of this type which grows most slowly is clearly the one with base 2:

$$F(1) = 2 \exp 0$$
$$F(2) = 2 \exp 1$$
$$\vdots$$
$$F(n) = 2 * F(n-1)$$

Another candidate series is the Fibonacci series, suitably modified. We add one to the term at each step in the production of the series, thus:

$$G(1) = 1$$
$$G(2) = 2$$
$$\vdots$$
$$G(n) = G(n-2) + G(n-1) + 1$$

There are no pairwise sums here, and this series grows slower than the powers of 2.

The algorithm which uses one of these series to assign limiting values can be described (informally) as follows:

ALGORITHM 2:

Step 1: initialise all associated values to the temporary values computed according to some natural series and assign them to the letters with the highest frequency of occurrence, thus obtaining the lowest values. Apply step2 to each key, Ki (0<i<# of keys), in turn.

Step 2: if Ki(1), the first character, has been 'tried' already, see whether Ki(length Ki), the last character, has been 'tried'. If both have been tried, proceed to the next key.

Step 3: for an 'untried' character, vary the associated value from 0 to the temporary value in increments of 1 until no collision occurs in the hash values. [A boolean function, CHECK, makes the necessary alterations of the hash values of the key list and then makes an exhaustive search for collisions].

Step 4: once a letter has been assigned an associated value, mark this letter 'tried'.

Algorithm 2 is not a backtracking algorithm in the classical sense, but an intelligently controlled ennumerative one. Since no associated value can exceed the temporary value given to the letter in step1 (because the temporary values are guaranteed to avoid collisions!), there is an upper bound to the search time. [If we are using powers of 2 as temporary values, adding one letter to the set of those which will be assigned values doubles the number of possible combinations; this upper bound may become quite large]. The time complexity of the CHECK function is O(N exp 2) because we need to compare each key against all the rest. Each time we try a new assignment we invoke CHECK, so if s letters of the alphabet occur in selected positions then the function will be called, in the worst case, the sum over all i, 0<i<s+1, of F(i). If we add a multiplicative factor m, such as the word length, the worst case cost is multiplied by m. The overall time complexity, in the worst case, of the execution of step3 is O((N exp 2)*m*SUM(F(i):0<i<s+1)). Using the powers of 2, the third factor is no greater than 2*(2 exp 26) = 2 exp 27, about 128 million.

In practise, this algorithm tends to find a solution long before the worst case occurs; when the set of keys is small enough we often get an almost-minimal hash function. When the number of keys approaches 200, the loading factor tends to fall to about 50% or less.

A third algorithm has been written to produce almost minimal hash functions quickly, enabling us to apply the method to some large sets (>500) of keys with good results.

Algorithm 3 assigns a set of associated values for each selected letter position. In effect we distinguish the occurrence of, say, an 'e' in the first position from one in the third position. Words such as 'on' and 'no', 'was' and 'saw', and 'live' and 'evil' are now easily distinguished. If we have p-selected letter positions, then the hash function has the form:

hash value = assoc value of letter in 1st selected pos'n +
assoc value of letter in 2nd selected pos'n +
:
assoc value of letter in p-th selected pos'n

This algorithm utilises a pre-processor which first orders the keys by the product of the frequencies of letters in selected positions. The words are then reordered as in previous algorithms so that words which share letters are grouped together. Any word which has a letter in a selected position which occurs in that word only will be one of the last keys added to the hash table; this unique letter occurrence makes it possible to place this key in any open location in the hash table without disturbing the values of the remaining letters which might affect the hash values of other keys.

Keys are placed in the hash table in groups. Groups have two properties: (i) one letter is common to all keys in the group; and (ii) all other letters which occur in the group have already been assigned values. Grouping the words is determined as part of the preprocessing. We may find that some pair of words in a group have equal sums of associated values previously assigned to the respective sets of letters which the words do not have in common. When this situation occurs, the value of the letter held in common cannot affect the existance of a collision, so we must change some previously assigned associated value(s). In order to minimise the number of letters (and keys) which need to be re-considered, we choose to change the value of the letter which is in only one of the two words and most recently was assigned a value. That letter's group and all that were added after it are removed from the hash table, and placed at the head of the list of groups not yet added. We then continue adding groups to the hash table until all the keys are included. Clearly, the last keys added, those with unique letter occurrences, can be placed anywhere in the table beyond the sum of letters which already have values. These keys tend to fill gaps in the hash table, promoting the minimality of the size of the hash table.

A brief outline of the algorithm follows:

ALGORITHM 3:

Step 1: the user is prompted to select a set of letter positions.

Step 2: check for conflicts (two keys having the same characteristics). If conflict occurs, continue with step1; otherwise continue with step3.

Step 3: order the keys so that the most frequent letters occur earliest in the list of keys. Words with a unique letter occurrence in some position are placed at the end of the list.

Step 4: add the next group of keys to the hash table by varying the value of any letter which hasn't occurred before. If an unresolvable conflict occurs, return to an earlier stage in the calculation by removing conflicting groups from the hash table. If no groups remain, stop; otherwise performs step4 again.

The APL implementation of this algorithm, running under the Michigan Terminal System [MTS] on an IBM 4341 computer, has produced excellent results.

## 3. Experimental Results

We now illustrate some basic programming results. The programs were originally written in UCSD Pascal and run on a North Star Horizon II micro-computer. Since we decided to try additional experiments in APL, we translated the programs to APL and Pascal/UBC and ran them on an IBM 4341 computer under the MTS operating system.

We first present the results of using Cichelli's algorithm, algorithm 0, and our algorithm 2 on several test cases, including those given by Cichelli. These algorithms are both implemented in Pascal programs. The final pages of our examples illustrate an interactive version of algorithm 3 with most of the variations mentioned in the text incorporated. The outpuut of algorithm 3 was produced by an APL program.

ALGORITHM 0

Starting at 14:50:48 on FEB 19, 1981
Solving for 1 solution
Placing 31 words

FIND PERFECT HASH FUNCTIONS FOR:
THIRTY-ONE MOST FREQUENT WORDS

THE, OF, AND, TO, A, IN, IT, THAT, I, IS, FOR, BE, WAS, YOU, AS, WITH, HE, HAVE, ON, BY, NOT, AT, THIS, ARE, WE, HIS, BUT, THEY, ALL, WILL, OR

NO CONFLICTS
Starting search at 14:50:49

| | KEY | HASH VALUE | FIRST ASSOC VALUE | SECOND ASSOC VALUE |
|---|---|---|---|---|
| 1 | I | 1 | 0 | 0 |
| 2 | IT | 2 | 0 | 0 |
| 3 | YOU | 3 | 0 | 0 |
| 4 | THAT | 4 | 0 | 0 |
| 5 | AT | 5 | 3 | 0 |
| 6 | IS | 6 | 0 | 4 |
| 7 | A | 7 | 3 | 3 |
| 8 | THIS | 8 | 0 | 4 |
| 9 | AS | 9 | 3 | 4 |
| 10 | THE | 10 | 0 | 7 |
| 11 | TO | 11 | 0 | 9 |
| 12 | HIS | 12 | 5 | 4 |
| 13 | ARE | 13 | 3 | 7 |
| 14 | HE | 14 | 5 | 7 |
| 15 | FROM | 15 | 9 | 2 |
| 16 | HAVE | 16 | 5 | 7 |
| 17 | BY | 17 | 15 | 0 |
| 18 | BUT | 18 | 15 | 0 |
| 19 | AND | 19 | 3 | 13 |
| 20 | OF | 20 | 9 | 9 |
| 21 | HAD | 21 | 5 | 13 |
| 22 | IN | 22 | 0 | 20 |
| 23 | NOT | 23 | 20 | 0 |
| 24 | BE | 24 | 15 | 7 |
| 25 | WAS | 25 | 18 | 4 |
| 26 | HER | 26 | 5 | 18 |
| 27 | WITH | 27 | 18 | 5 |
| 28 | WHICH | 28 | 18 | 5 |
| 29 | OR | 29 | 9 | 18 |
| 30 | FOR | 30 | 9 | 18 |
| 31 | OF | 31 | 9 | 20 |

PRINTING AT 14:51:06 FEB 19, 1981
addword called 7264 times.
try called 23467 times.
16775 milliseconds of CPU time elapsed.

ALGORITHM 2

THIRTY-ONE MOST FREQUENT WORDS

THE, OF, AND, TO, A, IN, IT, THAT, I, IS, FOR, BE, WAS, YOU, AS, WITH, HE, HAVE, ON, BY, NOT, AT, THIS, ARE, WE, HIS, BUT, THEY, ALL, WILL, OR

Calling 'try' at 19:43:04 on FEB 12, 1981
Finished at 19:43:06
2781 milliseconds of CPU time elapsed.

| | KEY | HASH VALUE | FIRST ASSOC VALUE | SECOND ASSOC VALUE |
|---|---|---|---|---|
| 1 | I | 1 | 0 | 0 |
| 2 | IT | 2 | 0 | 0 |
| 3 | THAT | 4 | 0 | 0 |
| 4 | AT | 5 | 3 | 0 |
| 5 | IS | 6 | 0 | 4 |
| 6 | A | 7 | 3 | 3 |
| 7 | THIS | 8 | 0 | 4 |
| 8 | AS | 9 | 3 | 4 |
| 9 | THE | 10 | 0 | 7 |
| 10 | TO | 11 | 0 | 9 |
| 11 | HIS | 12 | 5 | 4 |
| 12 | ARE | 13 | 3 | 7 |
| 13 | HE | 14 | 5 | 7 |
| 14 | OF | 15 | 9 | 4 |
| 15 | HAVE | 16 | 5 | 7 |
| 16 | IN | 17 | 0 | 15 |
| 17 | NOT | 18 | 15 | 0 |
| 18 | WAS | 19 | 12 | 4 |
| 19 | FROM | 20 | 4 | 12 |
| 20 | WITH | 21 | 12 | 5 |
| 21 | WHICH | 22 | 12 | 5 |
| 22 | FOR | 23 | 4 | 16 |
| 23 | HER | 24 | 5 | 16 |
| 24 | BUT | 25 | 22 | 0 |
| 25 | ON | 26 | 9 | 15 |
| 26 | OR | 27 | 9 | 16 |
| 27 | BY | 28 | 22 | 4 |
| 28 | YOU | 29 | 4 | 22 |
| 29 | AND | 30 | 3 | 24 |
| 30 | BE | 31 | 22 | 7 |
| 31 | HAD | 32 | 5 | 24 |

ALGORITHM 0
-----------

Starting at 15:31:29 on FEB 20, 1981
Solving for 1 solution
Placing 36 words

FIND PERFECT HASH FUNCTIONS FOR:
THIRTY-SIX PASCAL RESERVED WORDS
-------------------------------

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

NO CONFLICTS
Starting search at 15:31:30

| | KEY | HASH VALUE | FIRST ASSOC VALUE | SECOND ASSOC VALUE |
|---|---|---|---|---|
| 1 | DO | 2 | 0 | 0 |
| 2 | END | 3 | 0 | 0 |
| 3 | ELSE | 4 | 0 | 0 |
| 4 | TO | 5 | 3 | 0 |
| 5 | DOWNTO | 6 | 0 | 0 |
| 6 | TYPE | 7 | 3 | 0 |
| 7 | WHILE | 8 | 3 | 0 |
| 8 | OTHERWISE | 9 | 0 | 0 |
| 9 | OF | 10 | 0 | 8 |
| 10 | OR | 11 | 0 | 9 |
| 11 | FILE | 12 | 8 | 0 |
| 12 | NOT | 13 | 7 | 3 |
| 13 | THEN | 14 | 3 | 7 |
| 14 | RECORD | 15 | 9 | 0 |
| 15 | PACKED | 16 | 10 | 0 |
| 16 | CASE | 17 | 13 | 0 |
| 17 | REPEAT | 18 | 9 | 3 |
| 18 | PROCEDURE | 19 | 10 | 0 |
| 19 | FOR | 20 | 8 | 9 |
| 20 | CONST | 21 | 13 | 3 |
| 21 | AND | 22 | 19 | 0 |
| 22 | FUNCTION | 23 | 8 | 7 |
| 23 | MOD | 24 | 21 | 0 |
| 24 | NIL | 25 | 7 | 15 |
| 25 | GOTO | 26 | 22 | 0 |
| 26 | DIV | 27 | 0 | 24 |
| 27 | IN | 28 | 19 | 7 |
| 28 | IF | 29 | 19 | 8 |
| 29 | SET | 30 | 24 | 3 |
| 30 | BEGIN | 31 | 19 | 7 |
| 31 | UNTIL | 32 | 12 | 15 |
| 32 | WITH | 33 | 3 | 26 |
| 33 | ARRAY | 34 | 19 | 10 |
| 34 | LABEL | 35 | 15 | 15 |
| 35 | VAR | 36 | 24 | 9 |
| 36 | PROGRAM | 38 | 10 | 21 |

PRINTING AT 15:31:31 FEB 20, 1981
addword called 1392 times.
try called 4456 times.
1625 milliseconds of CPU time elapsed.

ALGORITHM 2
-----------

THIRTY-SIX PASCAL RESERVED WORDS
-------------------------------

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

Calling 'try' at 12:08:29 on FEB 20, 1981
Finished at 12:08:51
21103 milliseconds of CPU time elapsed.

| | KEY | HASH VALUE | FIRST ASSOC VALUE | SECOND ASSOC VALUE |
|---|---|---|---|---|
| 1 | DO | 2 | 0 | 0 |
| 2 | END | 3 | 0 | 0 |
| 3 | ELSE | 4 | 0 | 0 |
| 4 | TO | 5 | 3 | 0 |
| 5 | DOWNTO | 6 | 0 | 0 |
| 6 | TYPE | 7 | 3 | 0 |
| 7 | OR | 8 | 0 | 6 |
| 8 | OTHERWISE | 9 | 0 | 0 |
| 9 | NOT | 10 | 4 | 3 |
| 10 | THEN | 11 | 3 | 4 |
| 11 | RECORD | 12 | 6 | 0 |
| 12 | NIL | 13 | 4 | 6 |
| 13 | OF | 14 | 0 | 12 |
| 14 | REPEAT | 15 | 6 | 3 |
| 15 | FILE | 16 | 12 | 0 |
| 16 | LABEL | 17 | 6 | 6 |
| 17 | WHILE | 18 | 13 | 0 |
| 18 | PACKED | 19 | 13 | 0 |
| 19 | AND | 20 | 17 | 0 |
| 20 | FOR | 21 | 12 | 6 |
| 21 | PROCEDURE | 22 | 13 | 0 |
| 22 | CASE | 23 | 19 | 0 |
| 23 | FUNCTION | 24 | 12 | 4 |
| 24 | MOD | 25 | 22 | 0 |
| 25 | IN | 26 | 20 | 4 |
| 26 | CONST | 27 | 19 | 3 |
| 27 | GOTO | 28 | 24 | 0 |
| 28 | DIV | 29 | 0 | 26 |
| 29 | ARRAY | 30 | 17 | 8 |
| 30 | UNTIL | 31 | 20 | 6 |
| 31 | SET | 32 | 26 | 3 |
| 32 | WITH | 33 | 13 | 16 |
| 33 | IF | 34 | 20 | 12 |
| 34 | VAR | 35 | 26 | 6 |
| 35 | BEGIN | 36 | 27 | 4 |
| 36 | PROGRAM | 42 | 13 | 22 |

ALGORITHM 3

SEVENTY-FIVE PASCAL KEY WORDS AND RESERVED WORDS

MASHING STARTED AT 1981 5 19 15 44 53 139
MASHING FINISHED AT 1981 5 19 15 45 12 114
CPU SECONDS USED IN MASH IS 1.936
NUMBER OF TIMES THROUGH MASH MAIN LOOP IS 66

LETTER POSITIONS USED: 1 2 4

LETTER VALUES

| LETTER | POSN 1 | POSN 2 | POSN 4 |
|---|---|---|---|
| 'A' | 18 | 15 | 22 |
| 'B' | 49 | 25 | 0 |
| 'C' | 3 | 0 | 17 |
| 'D' | 12 | 0 | 3 |
| 'E' | 0 | 0 | 0 |
| 'F' | 11 | 2 | 27 |
| 'G' | 40 | 0 | 58 |
| 'H' | 0 | 21 | 25 |
| 'I' | 19 | 24 | 9 |
| 'K' | 0 | 0 | 37 |
| 'L' | 28 | 67 | 0 |
| 'M' | 31 | 0 | 0 |
| 'N' | 31 | 11 | 1 |
| 'O' | 9 | 6 | 4 |
| 'P' | 1 | 0 | 33 |
| 'Q' | 0 | 48 | 0 |
| 'R' | 0 | 11 | 1 |
| 'S' | 3 | 0 | 20 |
| 'T' | 1 | 60 | 9 |
| 'U' | 35 | 17 | 40 |
| 'V' | 51 | 0 | 28 |
| 'W' | 20 | 0 | 35 |
| 'X' | 0 | 32 | 0 |
| 'Y' | 0 | 67 | 0 |

HASH TABLE

| | | | | | |
|---|---|---|---|---|---|
| 4 | REAL | 29 | CHAR | 54 | GOTO |
| 5 | RESET | 30 | PUT | 55 | SQR |
| 6 | REPEAT | 31 | SIN | 56 | ARRAY |
| 7 | READ | 32 | COS | 57 | PACK |
| 8 | REWRITE | 33 | IN | 58 | NIL |
| 9 | READLN | 34 | CONST | 59 | PACKED |
| 10 | RECORD | 35 | AND | 60 | UNTIL |
| 11 | EOLN | 36 | EOF | 61 | MAXINT |
| 12 | ROUND | 37 | INTEGER | 62 | BOOLEAN |
| 13 | TO | 38 | PROCEDURE | 63 | BEGIN |
| 14 | TEXT | 39 | FILE | 64 | SQRT |
| 15 | SET | 40 | OF | 65 | OUTPUT |
| 16 | TRUE | 41 | SUCC | 66 | ABS |
| 17 | END | 42 | LN | 67 | DIV |
| 18 | TRUNC | 43 | MOD | 68 | EXP |
| 19 | PRED | 44 | ARCTAN | 69 | NEW |
| 20 | PAGE | 45 | WRITE | 70 | VAR |
| 21 | FOR | 46 | WHILE | 71 | ELSE |
| 22 | CASE | 47 | WRITELN | 72 | TYPE |
| 23 | OR | 48 | LABEL | 73 | WITH |
| 24 | DO | 49 | NOT | 74 | UNPACK |
| 25 | DOWNTO | 50 | IF | 75 | INPUT |
| 26 | ORD | 51 | FALSE | 76 | DISPOSE |
| 27 | THEN | 52 | GET | 77 | PROGRAM |
| 28 | CHR | 53 | FUNCTION | 78 | OTHERWISE |

ALGORITHM 2

TWO HUNDRED MOST FREQUENT ENGLISH WORDS

Calling 'try' at 20:36:43 on FEB 19, 1981
Finished at 22:54:19
8255568 milliseconds of CPU time elapsed.

| | KEY | HASH VALUE | FIRST ASSOC VALUE | SECOND ASSOC VALUE |
|---|---|---|---|---|
| 1 | JUST | 6 | 0 | 2 |
| 2 | THAT | 8 | 2 | 2 |
| 3 | THOUGHT | 11 | 2 | 2 |
| 4 | THE | 14 | 2 | 9 |
| 5 | TIME | 15 | 2 | 9 |
| 6 | THERE | 16 | 2 | 9 |
| 7 | SET | 18 | 13 | 2 |
| 8 | THIS | 19 | 2 | 13 |
| 9 | TIMES | 20 | 2 | 13 |
| 10 | THINGS | 21 | 2 | 13 |
| 11 | SHE | 25 | 13 | 9 |
| 12 | SOME | 26 | 13 | 9 |
| 13 | SINCE | 27 | 13 | 9 |
| 14 | DONT | 28 | 22 | 2 |
| 15 | DIFFERENT | 33 | 22 | 2 |
| 16 | END | 34 | 9 | 22 |
| 17 | SOMETIMES | 35 | 13 | 13 |
| 18 | SAID | 39 | 13 | 22 |
| 19 | SOUND | 40 | 13 | 22 |
| 20 | SHOULD | 41 | 13 | 22 |
| 21 | AT | 42 | 38 | 2 |
| 22 | ABOUT | 45 | 38 | 2 |
| 23 | ALMOST | 46 | 38 | 2 |
| 24 | DID | 47 | 22 | 22 |
| 25 | THEIR | 48 | 2 | 41 |
| 26 | ARE | 50 | 38 | 9 |
| 27 | TOGETHER | 51 | 2 | 41 |
| 28 | ABOVE | 52 | 38 | 9 |
| 29 | AS | 53 | 38 | 13 |
| 30 | EVER | 54 | 9 | 41 |
| 31 | ALWAYS | 57 | 38 | 13 |
| 32 | ANIMALS | 58 | 38 | 13 |
| 33 | NOT | 60 | 55 | 2 |
| 34 | THEN | 61 | 2 | 55 |
| 35 | NIGHT | 62 | 55 | 2 |
| 36 | AND | 63 | 38 | 22 |
| 37 | ASKED | 65 | 38 | 22 |
| 38 | AROUND | 66 | 38 | 22 |
| 39 | READ | 67 | 41 | 22 |
| 40 | EVEN | 68 | 9 | 55 |
| 41 | SOON | 72 | 13 | 55 |
| 42 | A | 77 | 38 | 38 |
| 43 | DOWN | 81 | 22 | 55 |
| 44 | AIR | 82 | 38 | 41 |
| 45 | AFTER | 84 | 38 | 41 |
| . | | . | . | . |
| . | | . | . | . |
| . | | . | . | . |
| 192 | MAY | 380 | 220 | 157 |
| 193 | MANY | 381 | 220 | 157 |
| 194 | FOLLOWING | 404 | 214 | 181 |
| 195 | YOU | 405 | 157 | 245 |
| 196 | UNTIL | 428 | 245 | 178 |
| 197 | FROM | 438 | 214 | 220 |
| 198 | IF | 451 | 235 | 214 |
| 199 | I | 471 | 235 | 235 |
| 200 | UP | 476 | 245 | 229 |

ALGORITHM 3

FIVE HUNDRED MOST FREQUENT ENGLISH WORDS

```
MASHING STARTED AT  1981 5 19 14 19 55 987
MASHING FINISHED AT 1981 5 19 14 20 30 147
CPU SECONDS USED IN LASH IS 23.288
NUMBER OF TIMES THROUGH LASH MAIN LOOP IS 112

LETTER POSITIONS USED: 1 2 3 4 10
```

LETTER VALUES

| LETTER | POSN 1 | POSN 2 | POSN 3 | POSN 4 | POSN 10 |
|---|---|---|---|---|---|
| 'A' | 34 | 11 | 50 | 365 | 40 |
| 'B' | 7 | 18 | 1 | 1 | 0 |
| 'C' | 1 | 7 | 99 | 153 | 20 |
| 'D' | 15 | 0 | 107 | 122 | 47 |
| 'E' | 98 | 0 | 0 | 0 | 0 |
| 'F' | 0 | 125 | 23 | 87 | 223 |
| 'G' | 91 | 100 | 2 | 243 | 59 |
| 'H' | 3 | 8 | 51 | 324 | 106 |
| 'I' | 229 | 7 | 166 | 177 | 1 |
| 'J' | 0 | 0 | 0 | 6 | 0 |
| 'K' | 368 | 0 | 147 | 72 | 311 |
| 'L' | 39 | 113 | 127 | 32 | 188 |
| 'M' | 4 | 61 | 8 | 184 | 264 |
| 'N' | 21 | 45 | 11 | 18 | 5 |
| 'O' | 152 | 1 | 1 | 210 | 201 |
| 'P' | 60 | 230 | 24 | 301 | 2 |
| 'Q' | 59 | 0 | 0 | 0 | 0 |
| 'R' | 190 | 64 | 1 | 11 | 54 |
| 'S' | 0 | 254 | 2 | 121 | 7 |
| 'T' | 1 | 161 | 7 | 26 | 17 |
| 'U' | 71 | 14 | 204 | 251 | 0 |
| 'V' | 87 | 348 | 110 | 191 | 0 |
| 'W' | 2 | 28 | 178 | 9 | 345 |
| 'X' | 0 | 2 | 26 | 7 | 0 |
| 'Y' | 198 | 144 | 203 | 286 | 9 |
| ''' | 0 | 0 | 2 | 132 | 0 |

HASH TABLE

```
  3 SEE        4 WE         5 HE        6 ME
  7 WERE       8 HERE       9 BE       10 MORE
 11 SHE       12 THE       13 SOME     14 COME
    :            :            :           :
    :            :            :           :
    :            :            :           :
688 I'M      689 IF       696 INTO    703 ALWAYS
768 OPINION  771 KNEW     772 KNOW    821 ASK
```

## 4. Application to Natural Language Lexicon Design

Retrieval methods usually assume equal likelihood of retrieval for each data item (Knuth, 1973). It is well documented in the literature of lexicography (Dewey, 1923; Carroll et al., 1971) 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 a 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, 1978, 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, 1980).

The 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 (see Table 4.1), 668 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.

| WORD | REL. FREQ. | TOTAL |
|---|---|---|
| THE | 7.310 | 7.310 |
| OF | 3.998 | 11.308 |
| AND | 3.280 | 14.588 |
| TO | 2.924 | 17.512 |
| A | 2.120 | 19.632 |
| IN | 2.116 | 21.748 |
| IT | 1.488 | 23.236 |
| THAT | 1.367 | 24.603 |
| I | 1.236 | 25.839 |
| IS | 1.224 | 27.063 |

| WORD | REL. FREQ. | TOTAL |
|---|---|---|
| FOR | 1.035 | 28.098 |
| BE | 0.956 | 29.054 |
| WAS | 0.850 | 29.904 |
| YOU | 0.808 | 30.712 |
| AS | 0.782 | 31.494 |
| WITH | 0.727 | 32.221 |
| HE | 0.687 | 32.908 |
| HAVE | 0.658 | 33.566 |
| ON | 0.643 | 34.209 |
| BY | 0.600 | 34.809 |
| NOT | 0.589 | 35.398 |
| AT | 0.585 | 35.983 |
| THIS | 0.572 | 36.555 |
| ARE | 0.549 | 37.104 |
| WE | 0.537 | 37.641 |
| HIS | 0.517 | 38.158 |
| BUT | 0.504 | 38.662 |
| THEY | 0.495 | 39.157 |
| ALL | 0.467 | 39.624 |
| WILL | 0.464 | 40.088 |
| OR | 0.458 | 40.546 |
| WHICH | 0.454 | 41.000 |
| FROM | 0.433 | 41.433 |
| HAD | 0.414 | 41.847 |
| HAS | 0.390 | 42.237 |
| ONE | 0.389 | 42.626 |
| OUR | 0.357 | 43.983 |

| WORD | REL. FREQ. | TOTAL |
|---|---|---|
| AN | 0.330 | 43.313 |
| BEEN | 0.329 | 43.642 |
| MY | 0.329 | 43.971 |
| THERE | 0.329 | 44.300 |
| NO | 0.321 | 44.621 |
| THEIR | 0.319 | 44.940 |
| WERE | 0.307 | 45.247 |
| SO | 0.300 | 45.547 |
| HIM | 0.285 | 45.832 |
| YOUR | 0.283 | 46.115 |
| CAN | 0.277 | 46.392 |
| WOULD | 0.267 | 46.659 |
| IF | 0.263 | 46.922 |
| THEM | 0.262 | 47.184 |
| WHAT | 0.260 | 47.444 |
| ME | 0.257 | 47.701 |
| WHO | 0.248 | 47.949 |
| DO | 0.239 | 48.188 |
| WHEN | 0.237 | 48.425 |
| HER | 0.234 | 48.659 |
| TIME | 0.232 | 48.891 |
| WAR | 0.217 | 49.108 |
| ANY | 0.210 | 49.318 |
| MORE | 0.210 | 49.528 |
| NOW | 0.210 | 49.738 |
| UP | 0.207 | 49.945 |
| OUT | 0.206 | 50.151 |

Table 4.1. The 64 Most Frequently Used Words.
(adapted from Dewey, 1923)

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, 1973) 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 - 200 entries or less; (ii) medium - 2000 to 4000 entries; and (iii) large - 30,000 to 60,000 entries. Typically, a small lexicon gains little from complex organisation schemes. Our 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. If the lexicon cannot fit into memory, it is appropriate to treat the medium size lexicon as a large lexicon.

Large lexicons are easier to analyse because they typically require secondary storage media. Our major concern in this case is to ensure that the number of retrievals from secondary memory is minimised. The favourable results we have obtained from Algorithm 3 lead us to consider including the 732 most frequent words in a single almost-minimal hash table, giving us one-probe retrieval in 75% of the cases. The remaining, say, 50,000 words could be mapped by a second hash function into 50 subsets of about 1000 words each. [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]. These could be stored separately in secondary memory. For each of these subsets, compute an almost-minimal perfect hash function, storing the associated values in the same secondary memory location as the lexical information itself. If the key we are searching for is not in the table of most-frequent words, then this scheme would perform a hash 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 us to retrieve any key with three hash calculations and one probe of secondary memory.

## 5. Concluding Remarks

We have tried ordering strategies other than the ones we have reported above, which were found to be inferior to Cichelli's original method. These include ordering the keys in terms of their length, which resulted in a hash table which did not meet the minimality criteria, and ordering the keys by increasing frequency of the letters (rather than by decreasing frequencies).

Discussion with Professor Krishnamoorthy of Rensselaer regarding the theoretical limitations of Algorithm 2 resulted in the following investigation. Consider the subproblem in which all the keys have the same length, hence the hash function is simplified to the addition of two associated values. If we take the set (AT, IT, IN, ON, TO, AN) and represent it by the undirected graph of Figure 5.1 below, the problem reduces to the assignment of integer weights to the nodes of the graph, such that the weights of the edges (we define the weight of an edge(u,v) as the weight of u plus the weight of v) are all distinct. We tried Algorithm 2 for the commplete graph illustrated in Figure 5.2.

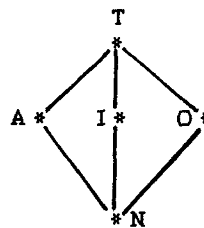

Figure 5.1.

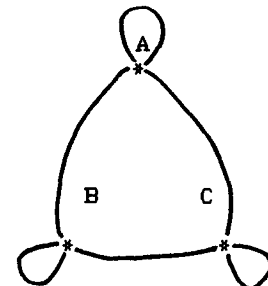

Figure 5.2.

We ran the program for the words (AA, AB, AC, BB, BC, CC). Here we found a limitation of our method. For a complete graph with ten nodes, viz., for the keys formed from (A, B, C, D, E, F, G, H, I, J), our implementation of Algorithm 2

returned {1, 2, 4, 8, 13, 21, 31, 45, 66, 81} respectively as the associated values. But the minimum set is given by {0, 1, 6, 10, 23, 26, 34, 41, 53, 55}. This sequence is called a B2 sequence and has been exhaustively studies in Number Theory. The problem of finding the absolute minimum (or optimum) sequence may well prove NP-complete but this proof appears to be a non-trivial problem.

We are currently undertaking a major comprehensive study of these perfect hashing schemes with all the variations we mentioned and a number of other variations we are formulating now. We anticipate the modularised Pascal and APL programs, which are written, will be available for distribution for all those interested by Spring, 1982.

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. 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

Allen, J. (1978). THE ANATOMY OF LISP, McGraw-Hill, New York.

Carroll, J., Davies, P., and Richman, B. (1971). THE AMERICAN HERITAGE WORD FREQUENCY BOOK, American Heritage Publishing Company, Inc., New York.

Cercone, N. (1975). "Representing Natural Language in Extended Semantic Networks", PhD Thesis, Technical Report TR75-11, Department of Computing Science, University of Alberta, Edmonton, Alberta.

Cercone, N., and Mercer, R. (1980). "Design of Lexicons in Some Natural Language Systems", ALLC Journal, 1, pp 37-51.

Cichelli, R. (1980). "Minimal Perfect Hash Functions Made Simple", Communications of the ACM, 23, pp 17-19.

Cichelli, R. (1980). Author's response to technical correspondance, Communications of the ACM, 23, pp 729.

Dewey, G. (1923). RELATIV FREQUENCY OF ENGLISH SPEECH SOUNDS. Harvard University Press, Cambridge, Mass.

Graham, R. (1980). "On Additive Bases and Harmmonious Graphs", SIAM Journal on Algebra and Discrete Methods, 1 (4), pp 382-404.

Halberstam and Roth. (1966). SEQUUENCES. volume 1, Oxford University Press.

Knuth, D. (1973). THE ART OF COMPUTER PROGRAMMING, volume 3: Sorting and Searching, Addison Wesley, Reading, Mass.

Moon, D. (1974). MACLISP REFERENCE MANUAL, Project MAC, MIT, Cambridge, Mass.

Morris, R. (1968). "Scatter Storage Techniques", Communications of the ACM, 11, pp 38-44.

Schank, R., Goldman, N., Rieger, C., and Riesbeck, C. (1973). "MARGIE: Memory Analysis, Response Generation and Inference in English", Proceedings of IJCAI3, pp 255-261.

Schwartz, E. (1963). "A Dictionary for Minimum Redundancy Encoding", Journal of the ACM, 10, pp 413-439.

Wilks, Y. (1973). "Preference Semantics", Stanford AI Project, memo AIM-206, Stanford University, Stanford, California.

Winograd, T. (1972). UNDERSTANDING NATURAL LANGUAGE. Academic Press, New York.

Woods, W., Kaplan, R., and Nash-Webber, B. (1972). "The Lunar Sciences Natural Language Information System: Final Report", Bolt, Beranek and Newman, Inc., Cambridge, Mass.