Implementation of a Perfect Hash Function Scheme

Sanjeev Datta and Edward A.Fox

TR 89-9

IMPLEMENTATION OF A PERFECT HASH FUNCTION SCHEME

Project report submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Master of Science
in
Computer Science and Applications
by
Sanjeev Datta
March 1988
Committee Chairman: Edward A. Fox
Committee Members: Lenwood Heath, H. Rex Hartson

Department of Computer Science
VPI&SU, Blacksburg, VA 24061-0106

ABSTRACT (Prepared for Technical Report, March 1989)

This report surveys the recent development in computing perfect hashing functions, and in particular, closely examines an algorithm proposed by Thomas Sager. An implementation of that algorithm in C has been done to demonstrate and verify the behavior of the algorithm for various settings of parameters.

Experimentation has shown that:

- the execution time of the algorithm is polynomial to $O(n^4)$; thus it is practical for small word sets, such as $n<300$;
- the ratio of size of the search graph to that of the word set affects the performance - the best choice of the ratio appeared to be around 0.75;
- expansion of the size of the target hashing tables improves the performance.

The report also suggests using binary tree indexing or multiple-level hashing approaches to augment the algorithm when it is applied to word sets of size more than 300, such as occur for CD-ROM applications.

CR Categories: G.2.2 [**Discrete Mathematics**]: Graph Theory - *graph algorithm* ; H.2.2 [**Database Management**]: Physical Design - *access methods* ; H.3.1.[**Information Storage and Retrieval**]: Content Analysis and Indexing - *indexing methods* ; H.3.3. [**Information Storage and Retrieval**]: Information Search and Retrieval - *retrieval models* .

General Terms: Algorithm, Design.

Key Words and Phrases: searching, hashing methods, minimal perfect hashing, direct addressing, hashing function.

ACKNOWLEDGEMENTS

I am deeply grateful to Dr. Edward Fox and Dr. Lenwood Heath for their constant encouragement and support throughout the entire period of the project. They have generously shared their time, knowledge and experience which was most invaluable. My special thanks are due to Dr. Rex Hartson for agreeing to be in my committee.

TABLE OF CONTENTS

1.0 INTRODUCTION

1.1 Motivation

The motivation for this project is to investigate various algorithms for developing minimal or almost minimal perfect hash functions (described in the next section), and then to implement one such hashing scheme intended for a large collection of lexical items. The *lexical items* are word entries in a dictionary and have been further described below.

In section 1.2 some basic concepts of *hashing* are discussed, and section 1.3 describes the project context.

1.2 Basic Concepts

Large amounts of data are stored on many computer systems, mainly on secondary storage units, like disks. One commonly used organization for such files of data supports *hashed access* [KNUT73], a scheme based on the fact that disk units and similar devices provide means of directly accessing any block, given a known address. Some portions of each record in a hashed file is selected as the primary key and is used to compute the record's address and locate it in the file. Hashed access is illustrated in Figure 1. Basically there are two essential characteristics of any hashed file:

1. The file space is organized into M slots. Each slot has the capacity to hold at least one of the N records of the file.

2. A function F which yields a slot address for a record given its key, i.e., F(key), provides a direct entry into a hash table, which stores the actual address of the corresponding record or records in the file space.

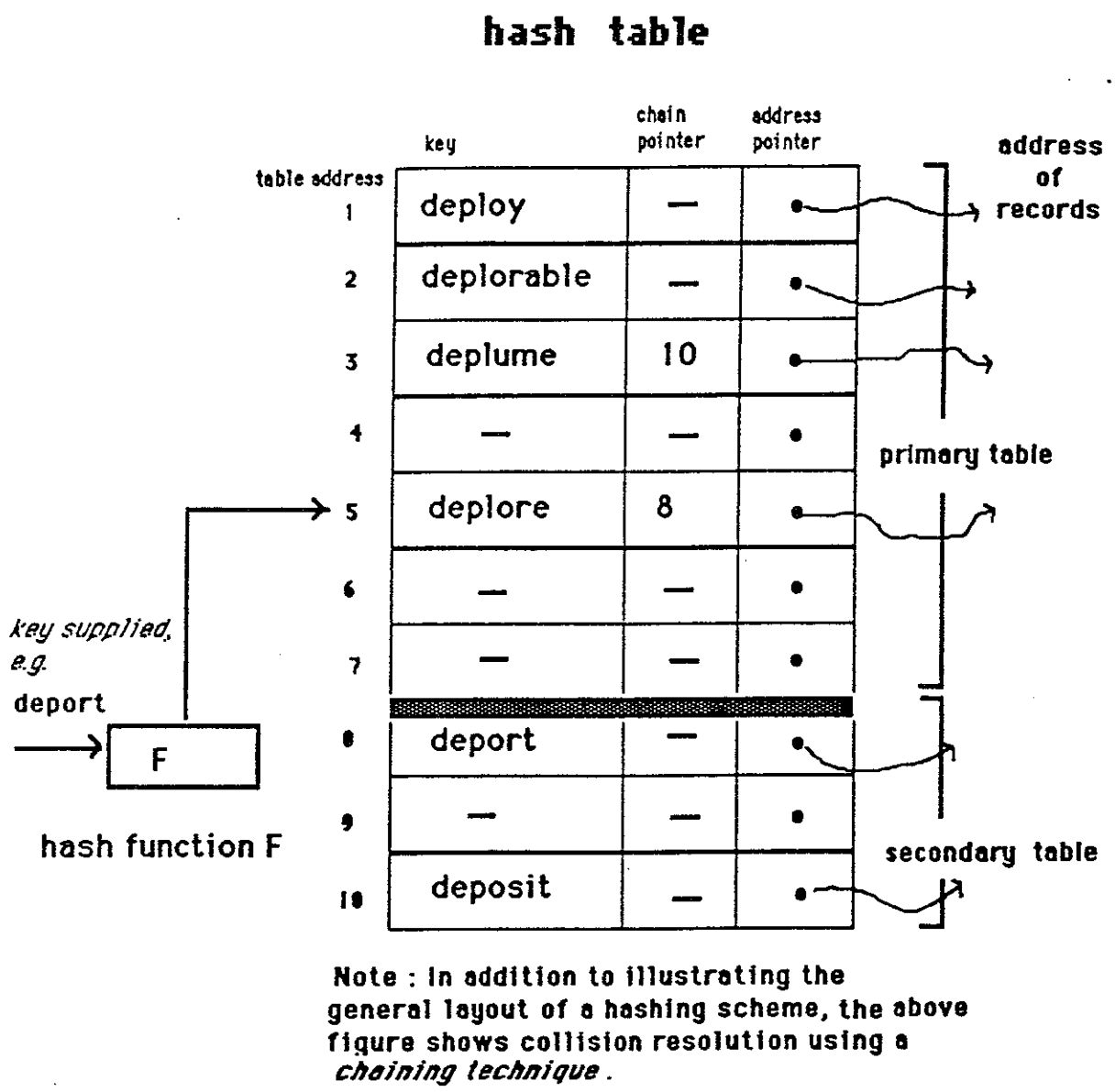


Figure 1. Basic layout of a hashing scheme

The focus in this project is to determine a computation F (for direct calculation of hash table address from key); little attention is given to other issues regarding the organization of hashed files.

A good hash function must have the following characteristics:

- The hash address should be easily computable, in other words the computation F should take a minimal amount of time.

- The LF should be as large as possible, optimally 1, where LF is defined as the ratio M/N, given a set of M keys and N slots in the hash table.

- The hash addresses of a given set of keys must be uniformly distributed, i.e., the number of collisions should be close to 0.

There are various schemes for handling collisions, broadly falling into two categories: *chaining techniques* and *open hash addressing techniques* [KNUT73]. In chaining techniques an additional field is provided in the hash table (See Figure 1): the chain pointer. This field links all the keys which have the same hash value into a linear chain. To locate a key, the hash function helps locate the chain head, and the chain is traversed until the key is found or a null pointer is encountered. In open hash addressing techniques, if a collision occurs, an empty slot in the table is located, and the new key is inserted. There are various policies for locating an empty slot. For example: in *linear probing* the table is searched sequentially in a circular fashion; in *double hashing,* a new randomizing key-to-address transformation computes the new slot address.

Collision handling is a very important aspect of most randomizing hashing schemes. However, since we are trying to develop ***perfect hash functions*** (described below), solutions for collisions will not concern us.

A ***perfect hash function*** assigns each lexical key to a unique slot in the hash table, thus avoiding collisions. A minimal perfect hash function makes LF one. Perfect hash functions

(henceforth PHF's) are found by deterministic refinement of key-to-address transformations. This is possible only with a static set of keys, i.e., the process that determines the perfect hash functions assumes knowledge of all the possible keys that the PHF will encounter.

It is not easy to find a PHF with certainty for any given set of words in a reasonable amount of time. As Knuth [KNUT73] observes, only one in 10 million functions is a perfect hash function for mapping the 31 most frequently used English words into 41 addresses.

A PHF with LF > 0.8 is called an *almost minimal* perfect hash function. In practice, the LF ranges from 0.6-0.95 for typical hash functions [CERC83].

1.3 Project Context

This project was undertaken for an information retrieval system (for the CODER project [FOXE86]) which requires rapid access to lexical items in a dictionary. The lexical items, which serve as meaning representations, can be viewed as records with various fields, e.g., the part-of-speech field, the definition field, etc., with the lexeme serving as the primary key. A ***lexeme*** is the string of characters that represents each entry in the dictionary, i.e., the ***headword.*** The database of these lexical items was built [WOHL86] by processing the magnetic tape for the *Collins Dictionary of the English Language* (CDEL).

Currently the dictionary database is in the form of a large collection of ***Prolog facts.*** The MU-Prolog (version 3.2) external database package [NAIS83] is used for storing the collection of Prolog facts. The database package uses a two-level superimposed coding scheme [RAMA85]. It will be interesting to see how the retrieval times of such a scheme compares with that of a perfect hash function scheme. But, to be able to use the PHF scheme, a new file organization will first be required. A simple and effective file organization for dictionary items is described in [AMSL87] and [PETE82].

Implementing a perfect hash function scheme is appropriate for a dictionary since:

- The set of keys is static.

- The *lexeme* forms a natural primary key for every item in the dictionary.

The properties of a lexical key that can be used to build perfect hash functions are the length of the lexical key and the positional occurrence of the letters of the alphabet.

The disadvantage of using a hashing scheme of course is that it will generally not allow sequential processing of the records. But since adjacent items in a dictionary need not have any logical relationship, the disadvantage might not be serious. In any case, if sequential processing is required, it can quite easily be incorporated using chaining or by maintaining an auxiliary table of address-pointers in sequential order.

1.4 Overview

Chapter 2 gives a brief review of some of the different methods that have been used to build PHF's. Chapter 3 discusses one such scheme, *the mincycle algorithm,* in greater detail. This scheme has been implemented. Chapter 4 describes some extensions made to *the mincycle algorithm* to handle a large collection of lexical items. In Chapter 5, some implementation issues have been touched upon and finally, Chapter 6 describes some of the performance measurements made on the algorithm and discusses the results.

2.0 LITERATURE REVIEW

Perfect hashing algorithms are an interesting and useful class of hashing algorithms and have been the subject of some active research lately. This chapter gives a brief review of some of the approaches that have been undertaken.

The building of PHF's typically requires an exponential search of the lexical key space, so the size of the set of words considered has usually been very limited. One approach has been to employ heuristics by exploiting the lexical properties of the keys to cut down on the search space. These heuristics attempt to make the average performance of the algorithm polynomial, rather than exponential.

2.1 Cichelli's Method

One such relatively early approach is discussed in Cichelli [CICH80]. The form of his hash function is:

Hash value = key length +
associated value of the key's first letter +
associated value of the key's last letter

The associated values for the letters are computed using a backtracking procedure which is potentially exponential in the size of the word set. Cichelli uses two ordering heuristics on the set of words that cause the inevitable hash value conflicts to occur during the search as early as possible, thus effectively pruning the search tree and speeding the computation. The first ordering heuristic is:

- Order the keys in descending order by the sum of the frequencies of the occurrences of each key's first and last letter in the list.

The seconding ordering heuristic is:

- Modify the order of the list of keys such that any word whose hash value is determined by assigning the associated character values already determined by previous words is placed next.

The backtracking procedure attempts to build the table of associated values by picking the words that have been ordered using the above heuristics. The table of associated values is an array of integers (ranging from 0 to a preassigned maximum) which is indexed by letter (i.e., the ordinal value of the letter in the alphabet).

The advantage of the above scheme is that it is simple, efficient and machine independent because the character code used by a particular machine never enters into the hash calculation.

The main disadvantages of this scheme are:

- No two keys can have the same length and the same first and last letters.
- The scheme works well only when the cardinality of the set of words is fairly small, e.g., 40.

Each of the disadvantages stated above automatically disqualifies this scheme for use in hashing the lexical items in the dictionary. According to Cichelli, though, this appears to be a good scheme for a small set of words, and the ordering heuristic seems to be quite effective. For example, the above procedure finds a solution for Pascal's 36 research words in about 7 seconds on a DEC PDP-11/45; without the second ordering heuristic, the search takes 5.5 hours.

2.2 *Cercone et al.'s Method*

Cercone et al. [CERC83] attempted to extend and generalize Cichelli's scheme. One of the key features in their effort has been to improve the representational power of the hash function by giving positional value to letters, e.g., an ***a*** in the 1st position of a word has an integer representation that

is different from an *a* in the 2nd position. This allows unique identification of each key.

To illustrate the gain in representational power they observe the following: Given an alphabet A and a maximum key length P, the key space (set of all possible keys) grows at a rate polynomial in card(A), and exponential in card(P). If card(A) = 26 and card(P) = 6, then if the letter value is independent of position, there are approximately 9×10^5 distinguishable keys, while if the letter value is dependent on position then there are approximately 3.2×10^8 distinguishable keys. This is a gain of on the order of 10^2 in the representational power. The price for this advantage is the overhead in maintaining a number of alphabet-to-integer tables, one for each position.

Using this scheme, any set of distinct lexical keys will be distinguishable. This scheme can be seen as an extension to Cichelli's algorithm in which the ordering heuristics were further refined to include **product frequencies, key grouping** and **unique letter appearances** within a given key group. The basis of most of the heuristics when using lexical strings is described by Cercone et al. [CERC83]:

> "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 likewise is an excellent predictor of how likely it is that a key will collide with other keys."

To begin with, the user is prompted for the set of letter positions to be used in the hashing. If n letter positions have been chosen, no two words can have identical n-tuples, where the n-tuple of a word consists of the letters in the designated n letter positions of the word.

A word with a *unique letter appearance* is a word which has a unique letter in any of the n letter positions.

To identify words of *unique letter appearances,* the number of occurrences of each letter in each position is counted and one subtracted from the total. Each key is then assigned a value which is the product of the occurrence counts for the selected letters in the key. If the key value is zero, then

it has a unique letter occurrence and is put at the bottom of the list. This is so that the word is considered last while assigning associated values, since it is impossible that this word will cause a collision.

After all the keys with unique letter occurrences have been identified and eliminated, the remaining keys are ordered by decreasing product of their letter frequency counts, termed the *product frequencies*. This is similar to Cichelli's first ordering heuristic; the product instead of the sum of the frequencies is considered. The second ordering heuristic is exactly the same as Cichelli's second ordering heuristic, *key grouping:* Find all keys whose hash addresses will be determined when the chosen key's new letters are assigned integer values.

This method does not use backtracking (to start with) but keeps track of the open hash address space at any given time. If the load factor suffers too much (the user can pre-set a tolerance level) an alternative backtracking process can be switched on to make the hash table less sparse.

This generalized approach definitely seems to work well (results are presented in the paper [CERC83]). However, the paper does not give an analysis of the algorithm nor does it compare the times for building the PHF's with and without the ordering heuristic. Therefore it is not clear how effective the heuristics really are, or how the timings will be affected if one increases the word set size. The authors claim only to have run their algorithm successfully for a maximum set size of 500.

The other point that has been left unclear by the authors is what guidelines must be followed by the user to choose the number of letter positions to be considered. It is clear that as long as the words are not uniquely identifiable (at the start of the process), one has to increment the letter positions to increase discrimination. It would have been interesting to see how the number of letter positions chosen affected the search process as well.

2.3 *Other Methods*

2.3.1 Sprugnoli's Method

Sprugnoli [SPRU78] devised two algorithms for computing perfect hash functions:

- Quotient Reduction Method

  $H_q(k_i) = \text{floor}[(k_i + A)/B]$

- Remainder Reduction Method

  $H_r(k_i) = \text{floor}\{[(A + k_i * B) \bmod C]/D\}$

where k_i is the i^{th} key in a set of N keys and A, B, C, D are constants. Algorithms for determining these constants are discussed in the paper. Sprugnoli's algorithms generate sparse tables and work only for small key sets (about 12). For example, for table sizes of about 12, the load factor is generally > 0.5 but when n = 20, load factors generally are < 1/300 [JAES81]. The algorithm is exponential in the number of words; moreover there is no guarantee that it will work for any set of words.

2.3.2 Jaeschke's Method

Jaeschke's method [JAES81] is important because the *reciprocal hashing scheme* he describes is guaranteed to work (i.e., proof of correctness is presented) but the author admits that the algorithm is efficient only for sets < 15. The algorithm is exponential in the number of identifiers.

Jaeschke proves that given a ***set of integers*** W (no description of the mapping from lexical key to the set of integers was discussed), a minimal PHF h can be found: $h : W \rightarrow I$ where I is the set of

integers [0, ..., N] where N = card(W). In other words, given W, there always exists C, D, E such that h(w) = floor (C/D * w + E) mod N, and C is such that:

$$(C/w_1 \bmod N) \neq (C/w_2 \bmod N) \neq \ldots \neq (C/w_n \bmod N)$$
where $W = \{w_1, w_2, \ldots, w_n\}$

The author shows that such a C exists only if the w_i's are pairwise relatively prime. According to the author, in random selections of W's there were only two cases out of 3000 in which a C defined as above did not exist. In the case when C does not exist, D and E can always be determined such that :

$(D \times w_1) + E, (D \times w_2) + E, \ldots, (D \times w_n) + E$ are pairwise relatively prime.

Since the set size is severely restricted by the complexity of the algorithm, this method could not be considered as a possible candidate for finding a PHF for the lexical items in the CDEL.

2.3.3 Chang's Method

Chang [CHAN86] modifies Jaeschke's scheme to make it directly applicable to a lexical string, and after Jaeschke's algorithm calls it a *letter-oriented reciprocal hashing scheme* . He assumes that there exist two integers i and j such that the pairs formed by the ith and jth letters of the identifiers are distinct. This assumption is obviously very restrictive: for example, there exists no solutions in the simple case of a set containing 3 words such that the first two words have the first two letters in common while the third word has only two letters:

department
depreciate
do

The form of his hash function is:

h(k) = d (first letter of k)
+ floor (C(first letter of k)/p(second letter of k))
mod n(first letter of k)

He then describes an algorithm to construct the four tables d(X), C(X), p(X), n(X) where X is the set of letters in the alphabet. Whereas Jaeschke's parameters were constants for a given set, the d, C, p, and n parameters of Chang are a function of the letters. He warns that like Jaeschke's algorithm, his algorithm works only for small sets of words. Also, his method requires that there be an injection from the set of words W to a set of prime integers, but no general algorithm for doing this is presented.

The next chapter focuses on yet another algorithm to building a PHF from a set of words. A more detailed discussion of that method is presented.

3.0 THE MINCYCLE ALGORITHM

This chapter describes yet another attempt to build PHF's, using the mincycle algorithm developed by Thomas Sager [SAGE85]. This algorithm is a fairly elaborate and interesting extension to Cichelli's algorithm [CICH80]. What makes this algorithm particularly attractive is that unlike the previous attempts [SPRUG78, JAES81, CICH80], which required execution time exponential in card(W) (where W is the set of words), the mincycle algorithm requires polynomial expected time to build the PHF (proportional to $card(W)^4$). Consequently, considerably larger sets of words (up to 500) can be handled at one time without causing prohibitively high execution times.

The algorithm can be divided into 3 parts. In part I some parameters for the PHF F are described. Part II describes an ordering heuristic for the set of words. In part III a backtracking search process is explained which attempts to build the PHF by using the ordered set of words built in part II.

3.1 Part I: Parameter Definitions

Let W be the set of words, and I the set of integers.

Let M = card(W).

Let N be the table size of the final hash table.
Since we are aiming for a minimal PHF, N = M. (Note that in the implementation, this does not have to be the case. The load factor LF (i.e., M/N) is a parameter that the user can set.)

Let R1 and R2 be two disjoint sets of integers, where each of them spans an integer interval.

Let h0, h1, h2 be three quickly computable pseudorandom functions:

- h0: $W \rightarrow I$
- h1: $W \rightarrow R1$
- h2: $W \rightarrow R2$

The PHF F is chosen to have the following form:

$$F(w) = (h0(w) + g \bullet h1(w) + g \bullet h2(w)) \bmod N$$

Since h0, h1 and h2 are determined even before entering the search process, the crucial function to discover so that F is a PHF will be the function g: $R \rightarrow [0, \ldots, N-1]$ where $R = R1 \cup R2$, $R = \text{card}(R)$.

The function h0, h1 and h2 should be able to convert any w, $w \in W$ to a unique 3-tuple: (h0(w) mod N, h1(w), h2(w)). The reason why only the first term has a *mod N* in it can be understood from section 3.8 of this chapter. It has to do with the fact that the terms g • h1(w) and g• h2(w) are always less than or equal to N. The functions suggested by Sager, and which seem to work well in most cases are:

- h0(w) = (length(w) + (Σ ord(w[i], i = 1 to length(w) by 3))
  (3 is the increment of i in the summation)
- h1(w) = (Σ ord(w[i], i = 1 to length(w) by 2) mod r)
- h2(w) = ((Σ ord(w[i], i = 2 to length(w) by 2) mod r) + r)
- R1 = [0, ..., r-1]
- R2 = [r, ..., 2r-1]

where

1. r = card(W)/2.
2. w[i] is the i^{th} character of the word w.
3. ord is the usual function that gives the integer representation of a character (this, typically, is machine dependent).

The 3-tuple described above must be unique. The process will continue only after the collisions have been resolved. Attempts to make the 3-tuples unique are made by changing the following parameters: h0, h1, h2, r, N. Description of how these parameters are changed is given in Chapter 5.

3.2 *Part II: Ordering Heuristic*

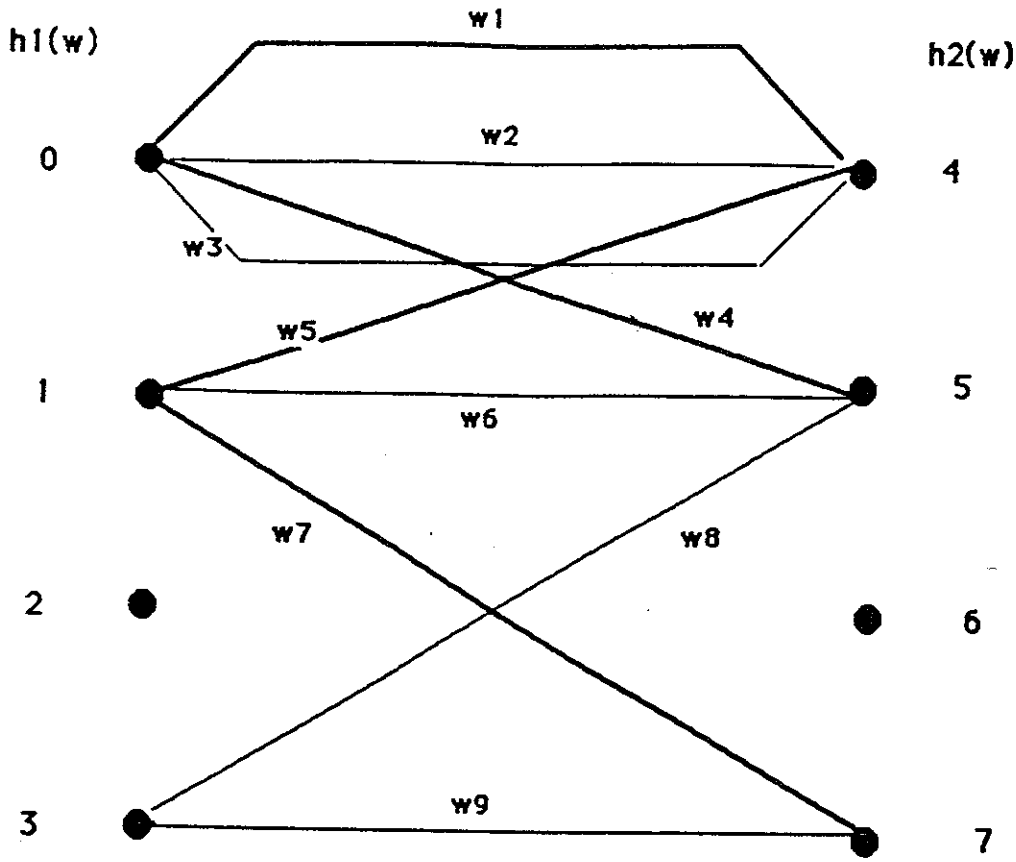
This is really the crucial part where an ordering is enforced on the set of words W. The ordering produced helps cut down on the search space in part III, and thus it is likely that part III will no longer be exponential in card(W). Part II has a complexity polynomial in card(W), proportional to $card(W)^4$. Sager has experimentally determined, and this has been observed (see Chapter 6), that when parameter r in part I is chosen to be r = card(W/2), part II seems to dominate part III in execution time. This should make the time complexity of the whole process of determining the PHF polynomial in card(W).

The end result of part II will be an ordering of the words in W: this ordering is in the form of a monotonic tower of subsets; i.e., let W_i be a subset of W, then the tower has the form:

$$\phi = W_k \supset \ldots \supset W_2 \supset W_1 \supset W_0 = W, \text{ where } \phi \text{ is the empty set.}$$

The first step is to build a bipartite graph from the set W. The h1(w) values and h2(w) values form the nodes. Each word w is an edge that connects the nodes h1(w) and h2(w). By construction, the h1 nodes are mutually nonadjacent and the h2 nodes are mutually nonadjacent (see Figure 2): any edge (u,v) in the graph is such that u is a *h1 node* and v is a *h2 node*.

The notion of **dependence** of a word w_i on a set of words Y is described next. w_i is dependent on Y if there exists a path from $h1(w_i)$ to $h2(w_i)$ in the bipartite graph formed by Y. Y must also satisfy the following property: given any $y_i \in Y$, y_i is not dependent on $Y - \{y_i\}$. This is equivalent to Y containing no *cycles*. For example in Figure 2, w_2 is dependent on $Y = \{w_1\}$ since there exists a path from $h1(w_2) = 0$ to $h2(w_2) = 4$ via the edge w_1. Similarly, w_6 is dependent on $Y = \{w_5, w_1, w_4\}$, since there exists a path from $h1(w_6) = 1$ to $h2(w_6) = 5$ via the edges w_5, w_1, w_4.



W0 = { }
W1 = {w1 w2 w3}
W2 = {w1 w2 w3 w4}
W3 = {w1 w2 w3 w4 w5 w6}
W4 = {w1 w2 w3 w4 w5 w6 w7}
W5 = {w1 w2 w3 w4 w5 w6 w7 w8 w9}

{w1 w4 w5 w7 w9} forms a spanning tree.
(Bold lines show that spanning tree in the graph.)

Figure 2. Representation of the bipartite graph

The tower subsets W_i are formed in the following way: W_0 is the empty set. W_1 is formed by including in it a word w_i (the way this word is chosen is explained below), and all other words that are dependent on w_i. The word w_i will be called a canonical element. W_2 is formed by:

1. including all the elements of W_1.
2. adding a word w_j that is not dependent on the set of canonical elements already in W_1. w_j then becomes a canonical element in W_2.
3. adding all the words $W - W_2$ which are dependent on the new set of canonical elements (w_j is the new member in the set of canonical elements).

The above process is repeated until $W_i = W$. Note from Figure 2 that the set of canonical elements $\{w_1, w_4, w_5, w_7, w_9\}$ forms a spanning tree over the bipartite graph. Typically there would be a spanning forest when there exists disconnected components in the original graph. The next thing to note is that the choice of canonical elements at each level of the tower is not random but dictated by the following criteria:

> at each level of the tower the choice of the canonical element should be such that: card(W_i) is maximized.

The proof as to why the above heuristic will reduce the search time in part III is given in Sager [SAGE84]. Intuitively it makes sense, and in fact heuristics along these lines have been suggested and explored both by Cichelli [CICH80] and Cercone [CERC83]: the larger group of words which are dependent on one another appears at the top of the tower. This results in early collision detection and therefore backtracking is minimized.

The next question is, how is the maximization of card(W_i) at each level of the tower being achieved? It is not clear if there exists an algorithm that will achieve the above in a reasonable amount of time. However an approximation to an optimal tower can be obtained by employing a ***mincycle*** algorithm and then choosing the ***maxedge*** on it. The mincycle algorithm detects the cycle of minimum length in the graph and arbitrarily picks an edge from it. In the case when there

are several mincycles present, the edge that is common to the maximum number of mincycles is chosen, called the ***maxedge.***

Going back to the example in Figure 2, there are several cycles present, to name a few (the node numbers are being used to identify the cycles): {1, 5, 3, 7} of length 4, {0, 4} of length 2, {0, 5, 1, 4} of length 4. The mincycle algorithm correctly picks {0, 4} as the cycle to be considered for maximizing card(W_1) (value 3) and chooses w_1 to be the canonical element. A choice of {1, 5, 3, 7} would have resulted in $W_1 = \{w_7\}$ so that card(W_1) would be 1.

The time complexity of the mincycle algorithm is $O(R^3)$ where R = 2r, and the complexity of the ***build tower*** algorithm is $O(R^4)$. In part I, we had chosen r = card(W)/2, which makes R = card(W). Therefore the ***build tower*** algorithm has a complexity $O(card(W)^4)$.

3.3 Part III: Determining PHF

In this part the actual PHF is built by using the tower of word subsets built in Part II. The PHF has the following form:

$$F(w) = (h0(w) + g \bullet h1(w) + g \bullet h2(w)) \bmod N$$

Let U: $Y \rightarrow [0, \ldots, N]$ be a function where Y is the set of canonical elements. U is defined to be:

$$U(w) = g \bullet h1(w) + g \bullet h2(w).$$

In the search procedure, U for the canonical element of the first tower subset is assigned a value between [0, ..., N] and consequently the F values for all the elements (in the first tower subset) are automatically determined. Similarly, U for the next canonical element in the tower is assigned, and the corresponding F values determined. After each assignment of F values, the algorithm checks for collisions. If there is a collision, the U value for the current canonical element is varied from [0, ..., N] with F values being reassigned and collisions checked for after each assignment. If none

of the values between 0 and N can avoid a collision, the procedure backtracks to the previous canonical element and varies its value from 0, ..., N. This process of backtracking could involve a very exhaustive search, but the ordering heuristic of part II minimizes the amount of backtracking. More formally, the F values are being determined in the following way:

for all $w \in X_t$ where $X_t = W_i - W_{i-1}$,

$F(w) = (h0(w) + (\sum (-1)^j \ U(y_j)) \bmod N$

where $j \in [0, \ldots, t]$ and path $(w) = y_1, y_2, \ldots, y_t$ and U is assigned the values:

$$U(x_j) = \begin{cases} (F(x_j) - h0(x_j)) \bmod N & \text{if } 0 < j < i \\ = n & \text{if } j = i \end{cases}$$

and the value of n is made to vary from 0, ..., N.

The path for any given word is unique for a given set of canonical elements (because the set of canonical elements form a spanning tree). However, the set of canonical elements need not be unique.

For example (see Figure 2), to determine $F(w_6)$:

$path(w_6) = \{w_5, w_1, w_4\}$

$$F(w_6) = (h0(w_6) + g \bullet h1(w_5) + g \bullet h2(w_5) - g \bullet h1(w_1) - g \bullet h2(w_1) + g \bullet h1(w_4)) \bmod N$$

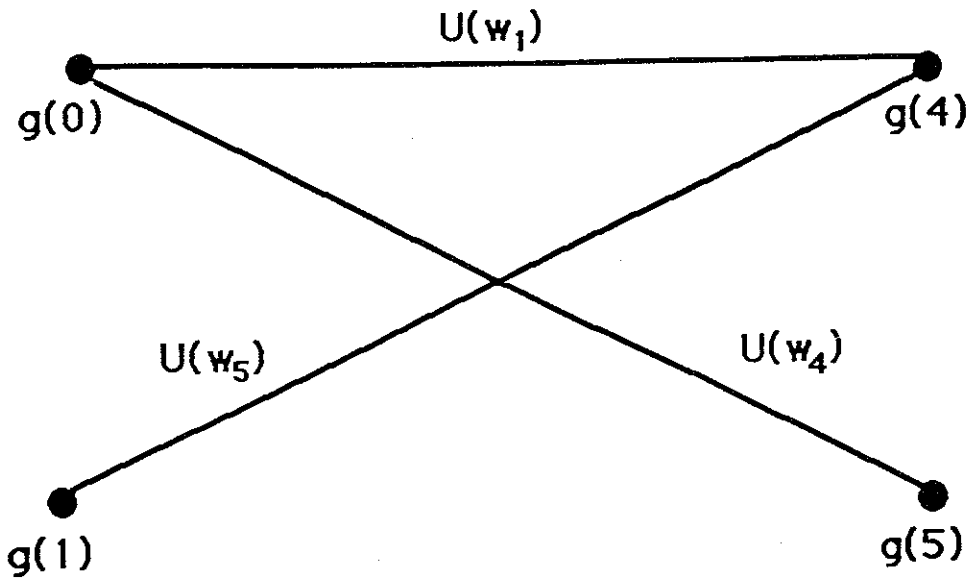
substituting the values for h1(w) and h2(w) from Figure 2:

$$\begin{aligned} F(w_6) &= (h0(w_0) + g(1) + g(4) - g(4) - g(0) + g(0) + g(5)) \bmod N \\ &= h0(w_6) + g(1) + g(5) \\ &= h0(w_6) + g \bullet h1(w_6) + g \bullet h2(w_6) \end{aligned}$$

After the backtracking process has assigned a unique value F to all the words and U values for all

the canonical elements have been determined, the next step is to determine the value of the function g: $\boldsymbol{R} \rightarrow [0, \ldots, N-1]$.

The g function is determined by traversing all the spanning trees in the original graph. The method of assigning values to g is straightforward. U can be written in the form $U = g(x) + g(x')$, where x and x' are canonical elements. For example, from Figure 3 we see: $U(w_4) = g(1) + g(4)$. One of the above g values, say g(1), is assigned 0 and g(4) is determined accordingly: $g(4) = U(w_4) - g(1)$. Similarly the value of g(4) is used to determine the value of g(0), since $U(w_1)$ is known. Similarly for g(5).



shows how g gets assigned values
once U is determined, the figure shows
a portion of the spanning tree from
Figure 2.

Figure 3. Determining the g-function

4.0 EXTENSION TO SAGER'S ALGORITHM

So far, Sager's algorithm for determining PHF's has promised the most in terms of size of the word set (≈500) that can be processed in a reasonable amount of execution time and usually has a load factor equal to 1 as well. But to use a PHF scheme on a word-set as large as the number of entries in the *Collins Dictionary of the English Language* (i.e., almost 70,000), it is necessary to adopt a multi-level scheme. In the following section, an implementation of a straightforward two-level scheme using a combination of binary search lookup and perfect hashing is described. A more interesting approach which uses a two-level hashing scheme is described later.

4.1 Approach 1: Binary Search

There are approximately 70,000 entries for the CDEL. If those were divided into sets, each of 550 words, there would be 70,000/550 ≈ 128 such sets. Each set could have its own PHF, or its own table for the g-function. Now, all the 70,000 entries can be maintained in alphabetical order and the 128 sets can be formed sequentially so that the order is maintained both within each set and from set to set. While the sets are being built, the last word of each set is stored in an array *set-arr;* thus *set-arr* contains 127 ordered strings. When looking for any word, a binary search is first made on *set-arr* to determine which set the word belongs to (see Figure 4), and then the appropriate g-function is applied for the retrieval of the corresponding record. Since for the most part there will not be an exact match with the *set-arr* entries, at every stage a comparison will have to be made with two adjacent entries in the array. The binary search would in the worst case make $\log_2 128 = 7$ comparisons. To get to the hash-table address would thus require one table look-up and a few quick computations.

A slight improvement in search time could be achieved at a considerable cost of space if instead of storing the set separator strings in an array, we stored them in a trie data structure. The binary search involves entire string comparisons whereas the trie data structure will require only a few (explained below) character comparisons. The trie data structure would have to be modified. All the letters would have to be stored at each node in the trie, and characters that do not have a pointer

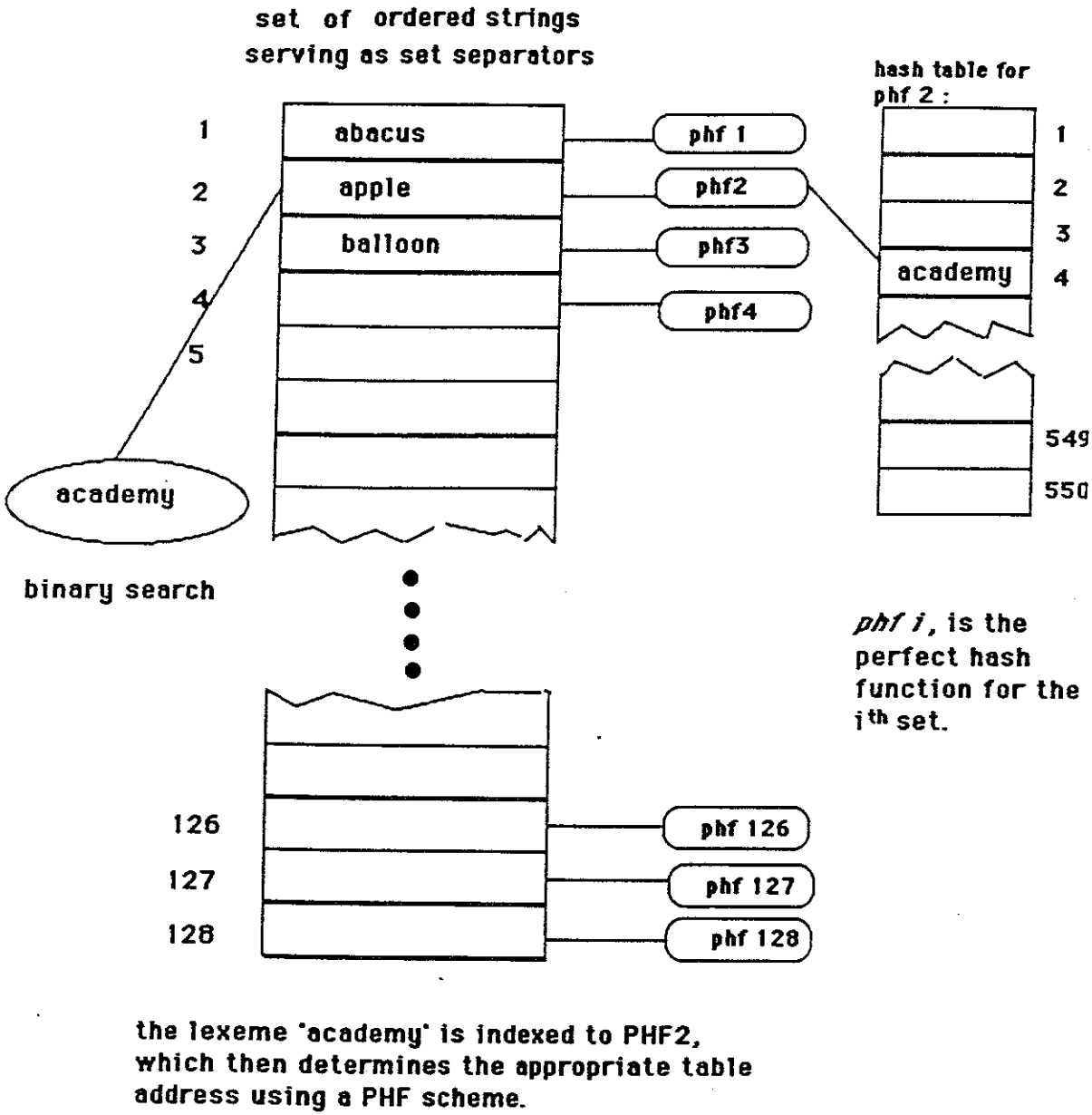


Figure 4. Access Method

to a subtrie will have to point directly at an information node (see Figure 5). The advantage of a trie in this situation is that it will be shallow and wide rather than deep. This is because the information nodes will contain set separators that are 550 words apart in the ordered word list, and it is highly unlikely that two adjacent set separators have in common, say, the first 4 characters. Therefore, usually the trie will not have a depth of more than 3. This is a good sign since the worst case search time in a trie is O(m) where m is the number of levels in the trie.

4.2 Approach 2: 2-Level Hashing

The following approach is suggested as an alternative to approach 1. This approach is more interesting and potentially vaster because a hashing process (not perfect hashing) is used even for the first level. The main criteria is that the level-1 hashing should be able to evenly distribute the 70,000 words into 128 sets of 550 words. A hash function that would do this directly was not found but an interesting way of getting sets of approximately the same size is described next. A hash table **tab** of size T is chosen such that T > (70,000/N), where N is the desired set-size used in level-2 hashing. T should preferably be a prime number. A quickly computable pseudorandom function has to be defined. A combination of the h0, h1 and h2 functions described in Chapter 3 (used for the level-2 hashing) seems to be as good a choice as any with the added benefit that they need not be recomputed for level-2. For example, the hash function in level-1 could have the form (h0 + h1 + h2) mod T. A bucket is associated with each slot in tab. The entire set of 70,000 words is then hashed and each word is assigned into its respective bucket. The buckets need to record not only the count of the words that are assigned to them, but also the associated lexical strings and their h0, h1 and h2 values. This is done to build the PHF functions at the second level as described below (see Figure 6).

Let b-count [i] denote the number of words in bucket i. A ***greedy*** algorithm is then used to partition the buckets into groups of buckets such that Σb-count[i] over any group gives a value close to N. The buckets in each group are linked together. The words in these linked buckets form the different sets used for level-2 hashing.

The larger T is, the better the granularity and therefore, the probability of a uniform distribution is higher.

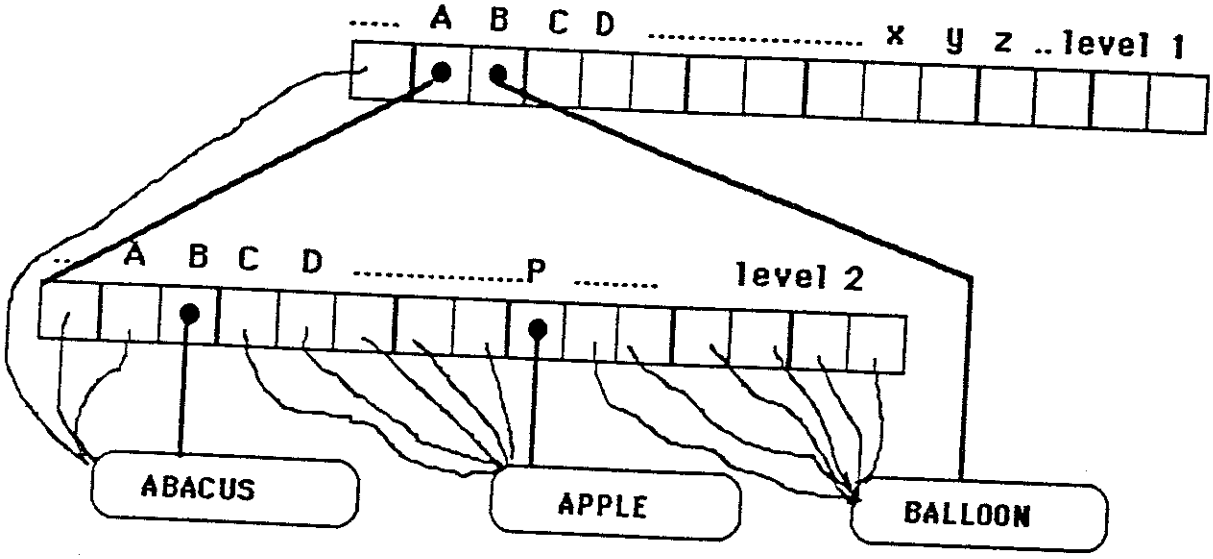


Figure 5. Modified *trie* structure

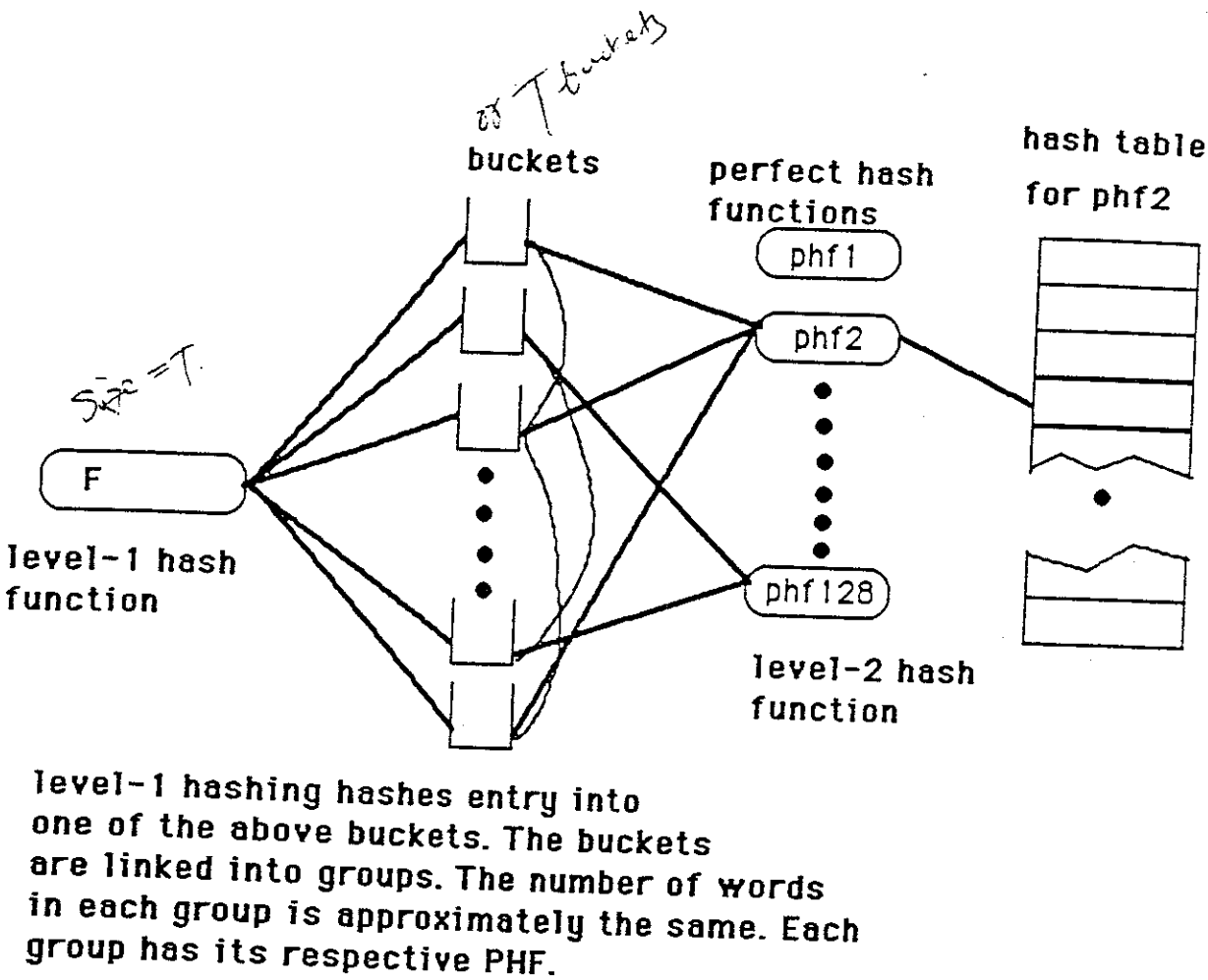


Figure 6. Scheme using a 2-level hashing scheme

Space Considerations:

The two level hashing scheme described above would need more space when building the hash functions compared to the first scheme (using binary search). Following is an estimate of the space required for some of the important data structures in the program. Both the schemes use identical methods to build the PHF for a given set of words. For the space estimates we will assume a value of $M \approx 550$. Let us also assume $R/M = 3/4$ (which seems like a good choice, based on results in Chapter 6). Therefore, $R = 550 \times 3/4 \approx 412$.

The static size of the *graph data structure* is:

> $412 \times 412 \times 4 \approx 0.68$ megabytes, (see Chapter 5; 4 bytes are used to store the pointer address to a word)

Additional space will be required at run-time:

> 550 (# of words) $\times$ 10 (average number of characters in a dictionary entry) $\times$ 2 (short integer to store array index) $\times$ 4 (pointer to next word) $\approx$ 4.4 kilobytes.

The *average* number of characters is used in the above expression because we have variable length dictionary entries. Data structures similar to the *graph structure* above are needed to store the *multiplicity* in the graph and to find the *dependencies* in the graph. Therefore the graph-related data structures would require a total of $0.68 \times 3 \approx 2$ megabytes. Once the PHF is built for a given set, a table for the *g-function* has to be stored in primary memory. The size of this table would be:

$$412 \text{ (\# of words)} \times 4 \text{ (integer size)} \approx 1.6 \text{ kilobytes.}$$

There will be a total of 128 such sets. Therefore to store the 128 tables we would need:

$$1.6 \times 128 \approx 0.2 \text{ megabytes.}$$

A total of approximately 2 + 0.2 = 2.2 megabytes of static memory would be required for the data structures for building the PHF. Once the PHF is built, we would require only 0.2 megabytes to store the tables for the g-functions.

Following is an estimate of the space required in *level-1* while building the set of PHF's. The scheme using the binary search will require an additional:

> 128 (# of sets) × 10 (average number of characters in a dictionary entry) × 4 (pointer to PHF) ≈ 5.1 kilobytes.

On the other hand, the scheme using a 2-level hashing scheme would require:

> {70,000 (total number of words) × 10 (average number of characters in a dictionary entry) × 4 (pointer to next word)}
> +
> {T (# of buckets) × 4 (integer to store bucket size) × 4 (pointer to connect bucket groups) × 4 (pointer to PHF)}
> ≈ 2.87 megabytes (assuming T = 2N = 1100).

The space required for the 2-level hashing scheme is rather high. The main reason for that is that would would like to store all the 70,000 words into a single data structure so that we may conveniently apply the *greedy algorithm*. This of course need not be the case, i.e., it should be possible to process the 70,000 words in smaller chunks (at the cost of slower time and more computations) so that building such a large data structure can be avoided.

5.0 IMPLEMENTATION ISSUES

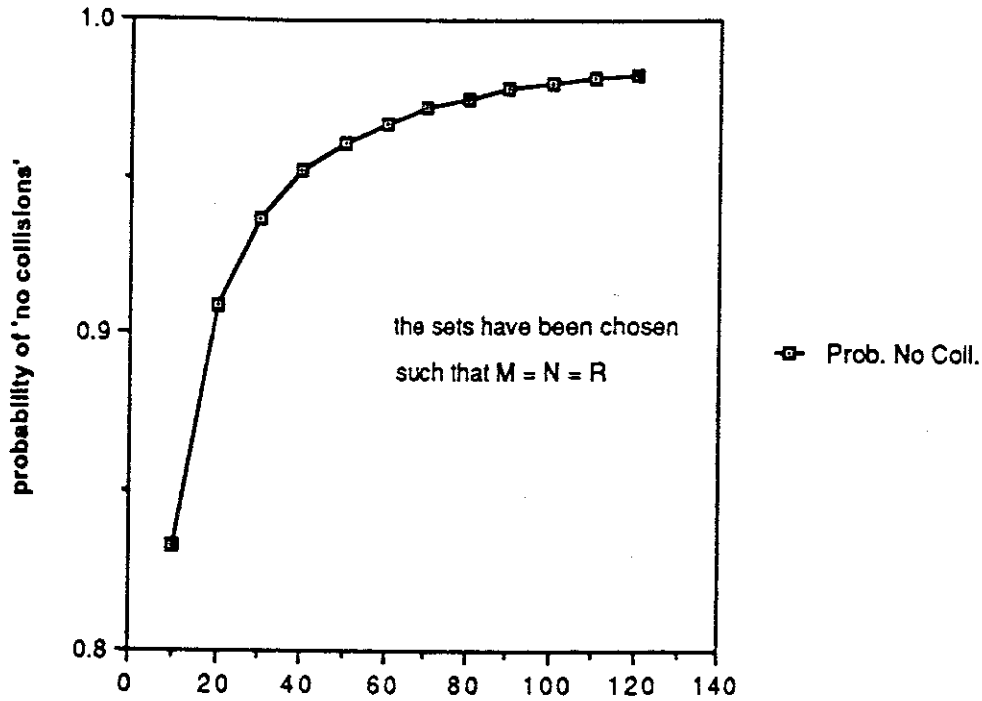
Since the implementation of Sager's algorithm is being applied to such a large collection of items for the first time, the system that develops the PHF has been kept as flexible as possible; for example, the word set size, the load factor, and the r value that determines the size of the bipartite graph are input as command line arguments; if no arguments are given, default values are used.

One of the preliminary steps in building the PHF is to make sure that the 3-tuple (h0(w) mod N, h1(w), h2(w)) is unique for all of the words in each table. In case of a collision, the user is first shown a list of the words that have collided and their respective h0, h1, h2 values. Next the user is given the option to reset any of the following parameters that would alter the h0, h1, h2 values:

- r: measure of the size of the bipartite graph
- N: size of the final hash table
- the *increments* in the *for loops* that compute h0, h1, h2.
- the *starting position* in the word, when computing h0, h1, h2 in the loop.

The user also has the option of letting the system try to solve the collision. The system has a simple collision policy: The remainder of h0(2) is taken after dividing it successively with a list of predefined numbers until there is no collision; in other words, the LF is made to vary by changing the value of N. If all the numbers in the list have been tried and there are still collisions, the process aborts. Such a simplistic approach has been taken because the probability of having collisions is very low (see Figure 7). So far, the data used have not warranted a more involved algorithm to avoid collisions.
The probability for having no collisions is given by the expression:

$$1 \times \frac{m-1}{m} \times \frac{m-2}{m} \times \ldots \times \frac{m-k+1}{m} = \frac{m!}{(m-k)!m^k} \quad where\ m = \frac{NR^2}{4}\ ,\quad k = M$$



case no.	set size	Prob. No Coll.
1	10	.833
2	20	.909
3	30	.937
4	40	.952
5	50	.961
6	60	.967
7	70	.972
8	80	.975
9	90	.978
10	100	.980
11	110	.982
12	120	.983

Figure 7. The probability of having 'no collisions'

The bipartite graph is implemented as a 2-dimensional array of linked lists. The lists contain the lexical strings that form the edges. The rows and columns of the 2-d array represent the nodes of the graph. A 2-d array also stores the multiplicity of the graph: the number of edges between two nodes. This is used to find the *mincycle* while building the tower. As the tower is being built, the graph shrinks. Every time a *mincycle* is determined, and a *maxedge {a, b}* chosen, the set of words between nodes ***a*** and ***b*** is added to the tower. Thus the tower is updated, and the two nodes are made to collapse into a single node. The collapsing of two nodes is carried out by copying all the connections from **b** to **a** (see Figure 8). The graph is made to shrink by copying the current last row and current last column to row ***b*** and column ***b*** respectively.

The backtracking process uses a *while loop* rather than recursive function calls to save on execution time. The search process also keeps track of the amount of backtracking to monitor the effectiveness of the buildtower heuristic. It is possible that after a full backtrack search, it is still not possible to determine a PHF. In this case, the system prompts the user to interactively decrease the load factor. If the user wishes to continue, he may input a lower LF, or he has the option of aborting the process.

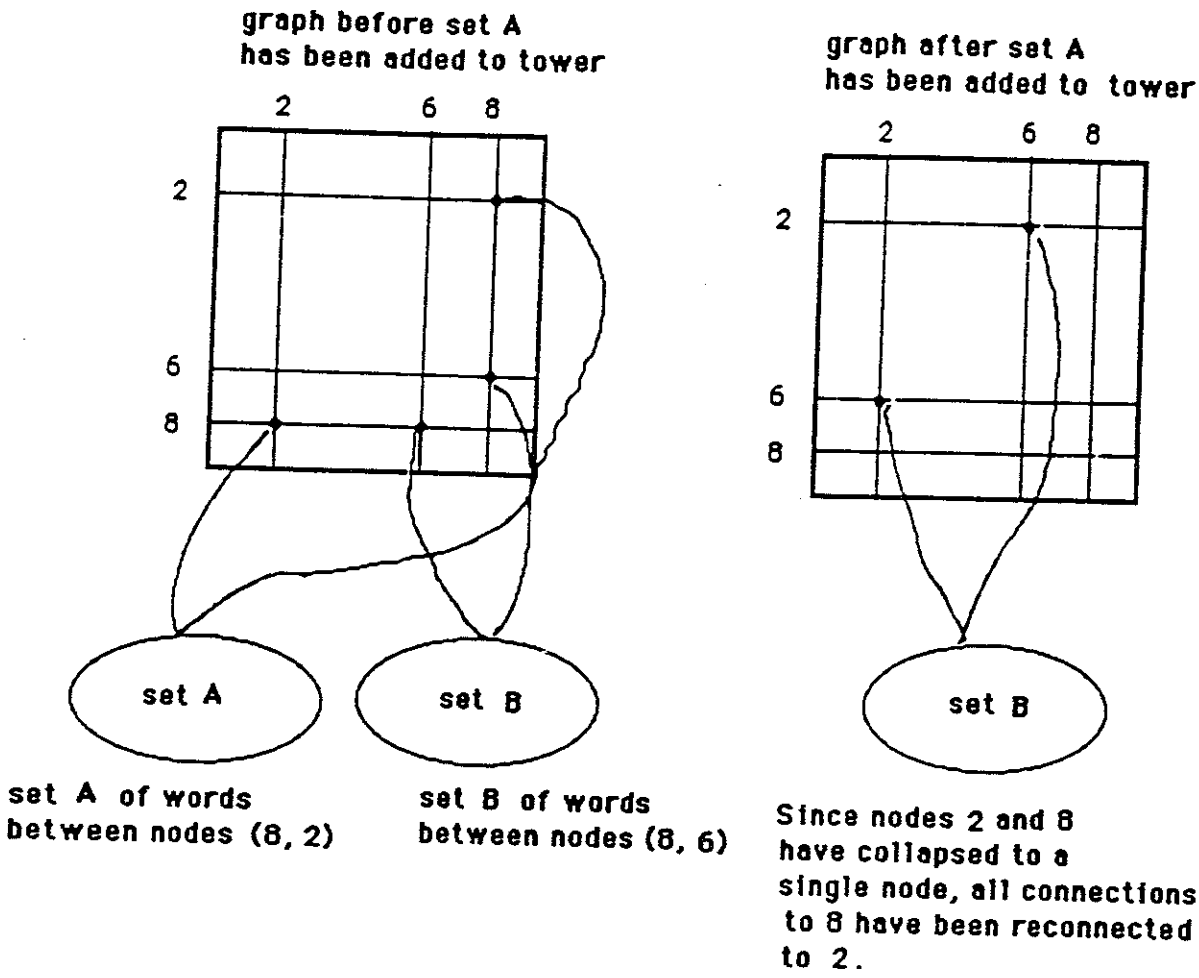


Figure 8. Collapsing of nodes

6.0 RESULTS AND DISCUSSION

Since the process for building the PHF's relies on certain heuristics intended to cut down on execution time, it is useful to study the behavior of the system by making some timing measurements. This study is also particularly useful as regards the following problem: the execution time was unusually high when using larger data sets (M = 200 - 500).

There are a number of parameters that can affect the performance:

- M: number of items in the set
- R: measure of the size of the bipartite graph
- LF: the load factor
- The ratio R/M

It was realized that the bulk of the processing time would be spent either in applying the **ordering heuristic** or in doing the **search** for the function values. Therefore the processing times (CPU time) spent inside the functions **build-tower** and **search** were measured separately, as well as the total time. Time was measured using the built-in UNIX ***time*** command. The total CPU time was measured up to an accuracy of 0.1 seconds. The time spent inside different functions was measured up to an accuracy of 1 second. All the measurements were made using an ***Apple Macintosh II system, running A/UX.***

6.1 Experiment I

Goal: To see if the execution time of the process is indeed polynomial in the size of the word set.

Set-up: The word-set size, M, was varied from 10 to 120. LF was kept at the optimum value of 1 throughout. The value of R was made M as suggested by Sager [SAGE85], who experimentally determined that when R = M, part II dominates part III in execution time. The following

measurements were made:

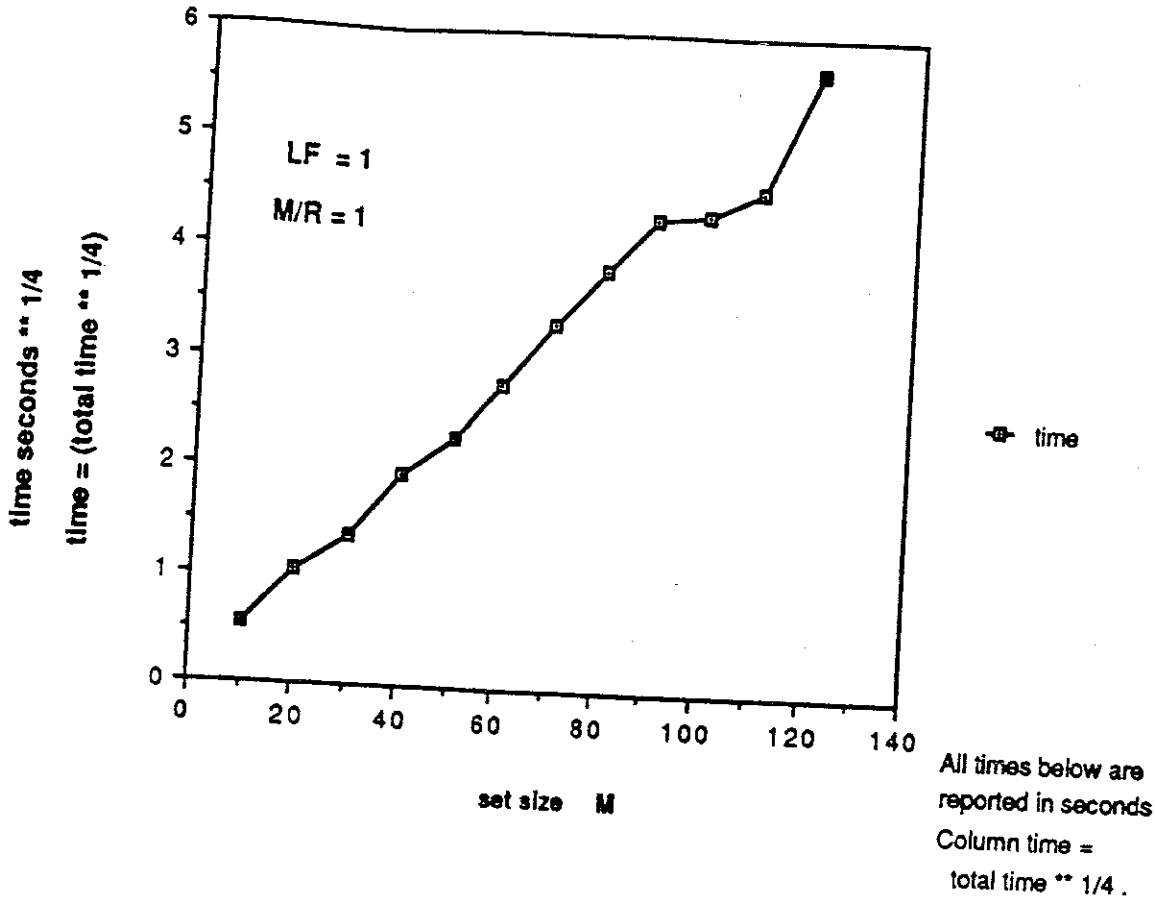
- total CPU time (in seconds)
- time spent in build-tower
- time spent in search
- number of backtracks made in search

Results: See Figures 9 and 10.

One set of results was obtained, as shown in Figure 9, with the words presorted to be in lexical order. A second set of results was obtained, as shown in Figure 10; the measurements above were repeated, with the following change: instead of ordered sets, randomly ordered word sets were used. The elements in each set was kept the same as in the previous run. The random ordered sets were created by calls to the UNIX built-in random number generator function, ***rand.***

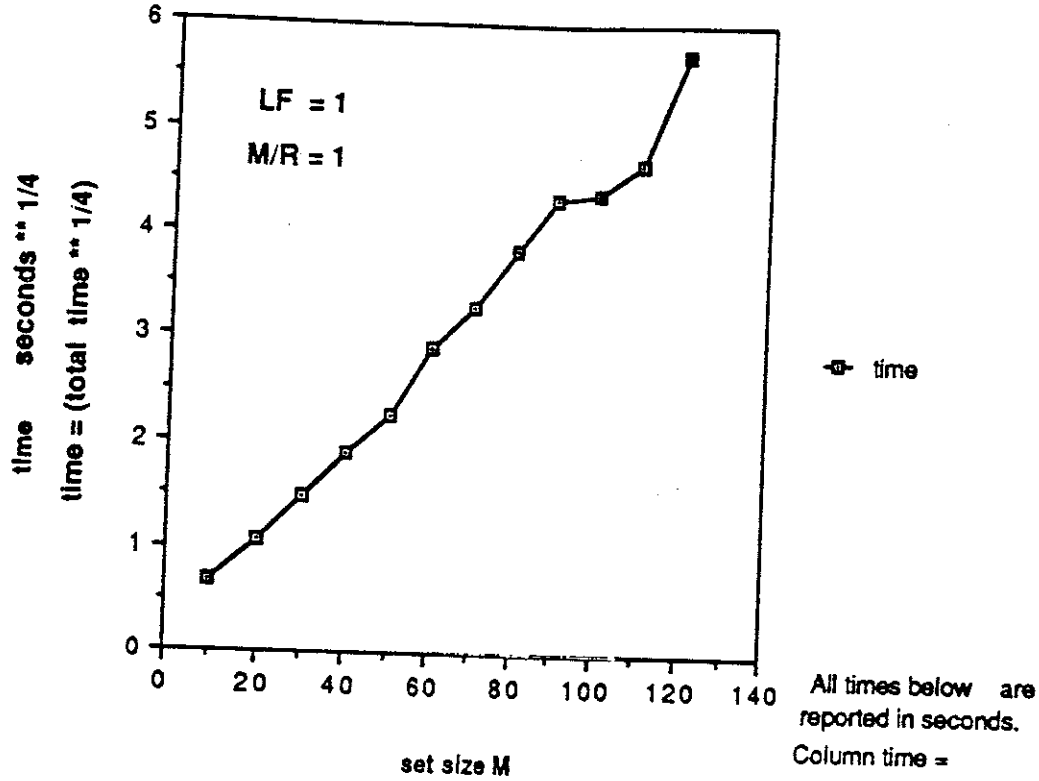
Conclusions:

1. A plot of M vs. $time^{1/4}$ was made to see if indeed the complexity of the entire process was being dominated by that of build-tower, $O(M^4)$. The $M \propto time^{1/4}$ is evident from both of the plots.
2. That no backtracking occurred in any of the sets ($R/M = 1$ for all sets), speaks well for the heuristic.
3. The ordering of the words in each set does not seem to have any significant effect on processing time, as the two graphs are almost identical.
4. Making $M = R$ does seem to take care of two important factors:
   - no collisions take place at the start of the process
   - no backtracking is observed in any of the runs above



set size	time	total time	search	build	# backtracks	tower height
10	0.56	0.1	0	0	0	8
20	1.04	1.2	0	1	0	14
30	1.36	3.5	0	3	0	23
40	1.94	14.1	0	13	0	34
50	2.3	28.4	1	27	0	41
60	2.78	59.4	0	58	0	49
70	3.36	128.0	0	126	0	59
80	3.86	222.6	0	220	0	67
90	4.32	349.4	2	346	0	75
100	4.38	368.5	4	362	0	76
110	4.60	448.2	6	439	0	76
120	5.71	1063.9	7	1053	0	97

Figure 9. Results from Experiment I



set size M	time	total time	search	build	# backtracks	tower height
10	0.67	0.2	0	0	0	8
20	1.06	1.3	0	1	0	17
30	1.48	4.8	0	5	0	24
40	1.89	12.8	0	13	0	32
50	2.27	26.4	0	26	0	39
60	2.90	70.9	1	69	0	51
70	3.30	118.6	1	117	0	56
80	3.85	220.1	1	218	0	66
90	4.34	354.5	2	350	0	75
100	4.38	369.1	4	363	0	76
110	4.67	475.8	5	467	0	78
120	5.71	1062.2	7	1052	0	97

Figure 10. Results from Experiment I

In fact, the time spent in **search** is almost negligible compared to the time spent in **build-tower**.

6.2 *Experiment II*

Goal: To check if R/M = 1 is indeed the optimum choice.

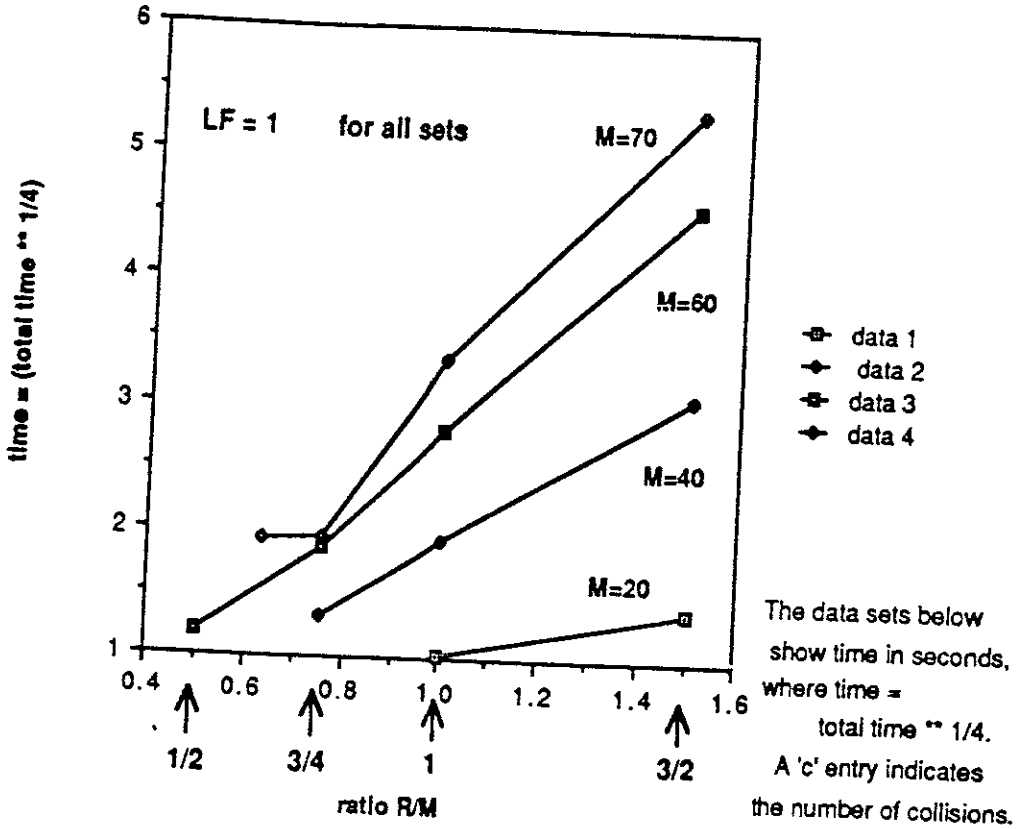
Set-up: It is not apparent from the algorithm that R/M = 1 should be the best choice. In this set of measurements, the LF was again restricted to 1, and plots were made of time vs. R/M, where R/M ranged from 1/4 to 3/2 in increments of 1/4. This was done for set sizes of M = 20, 40, 60, 70. The same quantities as in the previous experiment were measured.

Results: See Figure 11.

Conclusions: For a given M, R/M = 1 is not the best choice in terms of execution time. For example, for all the runs, a R/M = 3/4 has much better times; indeed, an order of magnitude better. In fact, as the ratio decreases, the time seems to improve for a while, in spite of backtracking in **search**, see Figure 12, data set 4, column 4. Of course, R/M cannot be arbitrarily small, since beyond a certain value, the following results:

- the collision frequency goes up (this happens because the choice of the parameters h0, h1, h2 is such that the *collision frequency* has a $1/(R^2)$ term in it).
- the time spent in backtracking takes over; for example, in data set 4, column 2, the search went on for over 5 hours without yielding values.

The conclusion that one could draw from this set of measurements is the following: For a given M and a given LF, there is a whole range of performance that depends on the choice of R. Therefore,



ratio R/M	data set 1	data set 2	data set 3	data set 4
0.25	4c	3c	2c	2c
0.5	1c	1c	1.2	over 5 hrs
0.63	-	-	-	1.92
0.75	1c	1.33	1.87	1.95
1.00	1.02	1.93	2.79	3.37
1.5	1.4	3.08	4.6	5.36

Figure 11. Results from experiment II

data set 1

ratio R/M	total time	search time	build time	# backtracks	tower height
1/4	4c	-	-	-	-
1/2	1c	-	-	-	-
3/4	1c	-	-	-	-
1	1.1	0	1	0	14
3/2	3.9	0	3	0	16

data set 2

ratio R/M	total time	search time	build time	# backtracks	tower height
1/4	3c	-	-	-	-
1/2	1c	-	-	-	-
3/4	3.1	0	3	0	28
1	14.1	0	13	0	34
3/2	(1:30.7) 90.7	1	90	0	38

data set 3

ratio R/M	total time	search time	build time	# backtracks	tower height
1/4	-	-	-	-	-
1/2	1.8	1	0	4	29
3/4	12.3	0	11	2	41
1	(1:0:3) 60.3	1	59	0	49
3/2	(7:26.6)446.6	1	445	0	57

data set 4

ratio R/M	total time	search time	build time	# backtracks	tower height
1/4	2c	-	-	-	-
1/2	over 5 hours	-	-	-	-
5/8	13.7	6	7	75	39
3/4	14.6	1	13	0	47
1	(2:07.9)128.6	1	128	0	59
3/2	(13:47.0)827.	1	825	0	66

Figure 12. Results from experiment II

to obtain optimal performance, the ratio R/M has to be fine-tuned for any given value of M.

Note that R/M = 1 does not seem to be the best choice in terms of time or space ($R \propto (space)^4$), but perhaps a safe choice in the sense that chance of collision is minimal.

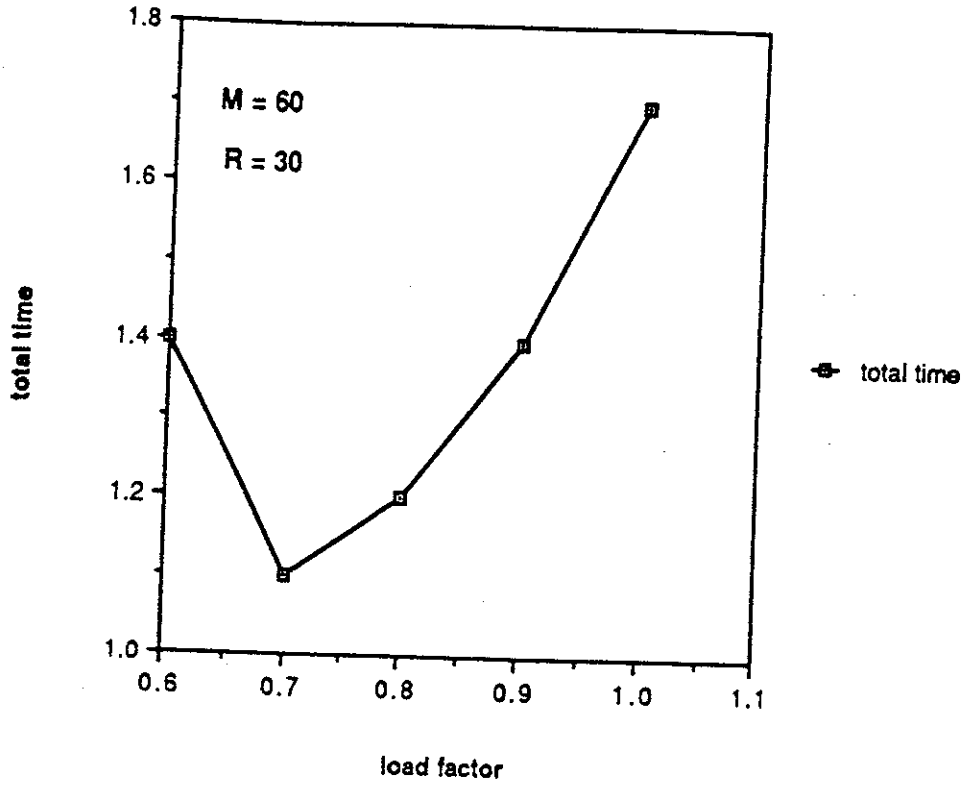
6.3 *Experiment III*

Goal: To see if a decreased LF cuts down execution time.

Set-up: For a fixed set size of M and a fixed ration of R/M, the LF was varied from 1 to 0.6 in decrements of 0.1. A typical result is shown below, and its interpretation given.

Results: See Figure 13.

Conclusions: As expected, a decrease in LF cuts down on the number of backtracks (since it is now easier to locate an empty slot), and hence on execution time. Even when the number of backtracks gets down to zero, the timing improves up to a point, because at every tower level, there are fewer assignments being made to the U value, i.e., the number of loops at a given tower level is fewer. If the LF is decreased beyond a certain point, the execution time suffers again, because of the increased table size; for example, initialization of a larger table, etc., becomes more and more significant. Therefore, even the LF has to be fine-tuned for optimal performance.



load factor	total time	# bactracks	tower height
1	1.7	4	The tower
0.9	1.4	2	height was
0.8	1.2	0	uniformly 29
0.7	1.1	0	because the
0.6	1.4	0	value of R was
			kept constant.

Figure 13. Results of experiment III

7.0 CONCLUSIONS

The implementation of the system for building ***perfect hash functions*** is complete and its behavior has been studied under several different conditions. However, the system has not yet been used to generate the PHF sets for the entire set of lexical items in the CDEL.

Though the time taken to build the PHF's is polynomial, $\propto$ card(W)^4, the 4th power still makes the execution very expensive (in terms of time) for card(W) $\approx$ 250-500. For example, to build a set of M = 120 took $\approx$ 18 minutes (with LF = 1, R/M = 1). To build a set size of 240, it might take $\approx$ 18 $\times 2^4$ minutes, which is nearly prohibitive! But as found out from the experiments, fine-tuning the system could result in significant gain (often a whole order of magnitude).

Therefore, a direct application of this system to a large lexical set might not yield favorable results, but after fine-tuning the system for a particular set size, building PHF's for large collections should be possible.

Also, it is fair to point out that neither the theoretical justification of the heuristics, nor the proof of correctness of the algorithm are fully convincing. However, the algorithm as a whole seems to work reasonably well (i.e., in polynomial time) for the range of data tested.

To get a better insight into the heuristics and into the general working of the system, it would perhaps help to monitor the building of the PHF's more closely. More probes could be added, for example, to detect the *length* of the mincycles or perhaps the *number* of spanning trees in the forest. Also, it would help to report the values of the probe while the system is being built rather than report them only after PHF has been successfully built.

8.0 REFERENCES

[AMSL87] R. A. Amsler, "How Do I Turn This Book On," *Proceedings of the Third Annual Conference of the University of Waterloo Centre for the Oxford English Dictionary (Use of Large Text Databases),* University of Waterloo, November, 1987.

[CERC83] N. Cercone, M. Krause, J. Boates, "Minimal and Almost Minimal Perfect Hash Function Search with Application to Natural Language Lexicon Design," *Computers and Mathematics with Applications,* 1983, V9, N1, pp. 215-231.

[CICH80] R. Cichelli, "Minimal Perfect Hash Functions Made Simple," *Commun. ACM,* 1980, V23, pp. 17-19.

[CHAN86] C. C. Chang, "Letter Oriented Reciprocal Hashing Scheme," *Information Sciences,* 1986, V38, N3, pp. 253-255.

[FOXE87] E. Fox, "Development of the CODER System: A Testbed for Artificial Intelligence Methods in Information Retrieval," *Information Processing and Management,* 1987, V23, N2, pp. 241-366.

[JAES81] G. Jaeschke, "Reciprocal Hashing - A Method for Generating Minimal Perfect Hash Functions," *Commun. ACM,* 1981, V24, N12, pp. 829-833.

[KNUT73] D. Knuth, *The Art of Computer Programming, Vol. 3,* Addison-Wesley, Reading, MA, 1973.

[NAIS83] L. Naish, J. Thom, "The MU-Prolog Deductive Database," *Technical Report 83/10,* Department of Computer Science, University of Melbourne, 1983.

[PETE82] J. L. Peterson, "Webster's Seventh New Collegiate Dictionary: A Computer-Readable Format," *Technical Report TR-196,* Department of Computer Science, University of Texas at Austin, Austin, May 1982.

[RAMA85] K. Ramamohanarao, J. Sheppard, "A Superimposed Codeword Indexing Scheme for Very Large Prolog Databases," *Technical Report 85/17,* Department of Computer Science, University of Melbourne, January 1985.

[SAGE85] T. Sager, "A Polynomial Time Generator for Minimal Perfect Hash Functions," *Commun. ACM,* May 1985, V28, N5, pp. 523-532.

[SAGE84] T. Sager, "A New Method for Generating Minimal Perfect Hashing Functions," *TR CSc-84-15,* University of Missouri-Rolla, Rolla, MO, November, 1984.

[SPRU78] R. Sprugnoli, "Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets," *Commun. ACM,* 1978, V20, pp. 841-850.

[WOHL86] R. C. Wohlend, "Creation of a Prolog Fact Base from the Collins English Dictionary," *MS Report, VPI&SU,* Computer Science Department, Virginia Polytechnic Institute and State University, Blacksburg, VA, March 1986.