CANADIAN THESES ON MICROFICHE

I.S.B.N.

THESES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL-339 (r. 82/08)

Canada

16

0-315-14402-5

National Library of Canada | Bibliothèque nationale du Canada

CANADIAN THESES ON MICROFICHE | *THÈSES CANADIENNES SUR MICROFICHE*

62311

NAME OF AUTHOR/*NOM DE L'AUTEUR* Max Martin Krause

TITLE OF THESIS/*TITRE DE LA THÈSE* Perfect Hash Function Search with Application to Computer Lexicon Design

UNIVERSITY/*UNIVERSITÉ* SIMON FRASER UNIVERSITY

DEGREE FOR WHICH THESIS WAS PRESENTED/*GRADE POUR LEQUEL CETTE THÈSE FUT PRÉSENTÉE* MASTER OF SCIENCE

YEAR THIS DEGREE CONFERRED/*ANNÉE D'OBTENTION DE CE GRADE*

NAME OF SUPERVISOR/*NOM DU DIRECTEUR DE THÈSE* Dr. Nick Cercone

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

DATED/*DATÉ* Feb. 25, 1982 SIGNED/*SIGNÉ*

PERMANENT ADDRESS/*RÉSIDENCE FIXÉ*

[redacted]

NL-91 (3-74)

PERFECT HASH FUNCTION SEARCH WITH APPLICATION TO COMPUTER LEXICON DESIGN

by

Max Martin Krause
B.A., University of Chicago, 1977

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the Department
of
Computing Science

SIMON FRASER UNIVERSITY

February 1982

APPROVAL

Name: Max Martin Krause

Degree: Master of Science

Title of Thesis: Perfect Hash Function Search With Application to Computer Lexicon Design

Examining Committee:

Chairperson: James J. Weinkam

Nick Cercone
Senior Supervisor

Wo Shun Luk

Brian Funt

David G. Kirkpatrick
External Examiner
Assistant Professor
Department of Computer Science
University of British Columbia

Date Approved: 25 February '82

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Perfect Hash Function Search with Application to Computer Lexicon Design

Author:

(signature)

MAX MARTIN KRAUSE

(name)

Feb. 25, 1982

(date)

Abstract

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

"...pataphysics will be, above all, the science of the particular, despite the common opinion that the only science is that of the general. Pataphysics will examine the laws governing exceptions..., since the laws that are supposed to have been discovered in the traditional universe are also correlations of exceptions, albeit more frequent ones..."

- Alfred Jarry, Exploits and Opinions of Dr. Faustroll, Pataphysician

Acknowledgements

Although they are not responsible for the faults contained herein, a number of people are responsible for making this thesis possible and for that I am most appreciative. First among them is my Senior Supervisor Dr. Nick Cercone, who has been a constant source of encouragement and inspiration to me. His lectures first quickened my interest in language processing, and he suggested the topic of this thesis. The guidance I have received in its production has shown me what good research and good writing are. I am deeply grateful for these things, and much more.

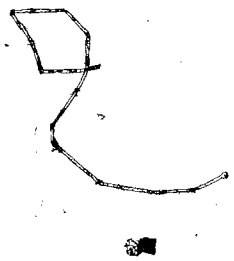
I also owe John Boates, the third collaborator in this research, a special debt of gratitude. John is the author of Algorithm 3 and has given freely of his time in providing documentation and experimental results related to his algorithm. Our many discussions of the problem have consistently provided me with intellectual stimulation and insight.

Pavol Hell, Art Liestman, and Brian Funt have given generously of their time and knowledge in discussions of various aspects of the problem. Wo Shun Luk and David Kirkpatrick have made valuable contributions to the organisation and coherence of this work.

I would also like to thank Wolfgang Richter of the Simon Fraser University Computing Centre for his patient help with text formatting problems.

Finally, but by no means last, I owe my wife Carol Sanders Krause more than I have a right to owe anyone, and certainly more than words can say.

Table of Contents



LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Perfect hash functions, a refinement of key-to-address transformation techniques, provide single probe retrieval of keys from a static table. By single probe retrieval we meant direct, random access to items in a database or table. This technique will be useful in any language processing application which has a fixed vocabulary of frequently used words.

Given a set of N keys and a hash table of size $r \geq N$, a perfect hash function maps the keys into the hash table with no collisions.

The loading factor LF of a hash table is the ratio of a number of keys to table size, N/r. A minimal perfect hash function maps N keys into N continguous locations for a loading factor of 1. We call a perfect hash function with a loading factor greater than or equal to 0.8 almost minimal.

Wiederhold [1977, p. 122] distinguishes deterministic and probabilistic direct access methods. A perfect hash function is a deterministic procedure which locates each key at a distinct table address. Unlike probabilistic key-to-address transformations, perfect hash functions do not allow collisions. This guarantees single probe retrieval and eliminates the need for collision resolution.

Three criteria of a good hash function are

1. the hash address is easily calculated;
2. the loading factor of the hash table is high (for a given set of keys); and,
3. the hash addresses of a given set of keys are distributed uniformly in the hash table.

A perfect hash function is optimal with respect to criterion 3; a minimal

perfect hash function is also optimal with respect to criterion 2[1].

Hash functions for applications which allow insertion of new keys assume uniform random occurrence of members of the key space. They are therefore designed to transform these keys into addresses which are scattered randomly but evenly in the address space. Perfect hash functions are feasible only for static sets of keys. The addition of one new key will usually require that a new function be defined for the entire set. When we are given a static set of keys, however, we can consider searching for a hash function which provides optimal distribution of the keys in the address space.

Perfect hash functions are difficult to find, even when we accept an almost minimal solution. Knuth [1973] estimates that only one in about ten million functions is a perfect hash function for mapping the 31 most frequent English words into 41 addresses. Functions which produce a minimal hash table further complicate the search.

1.1 Previous Work on Perfect Hash Functions

In this section, H is the name of a hash function, $H(k_i)$ is the hash address of the i-th key in K, the set of keys, and A,B,C,D represent constants.

The earliest published work on perfect hash functions appeared nearly twenty years ago. M. Greniewski and N. Turski [1963] used a perfect hash function of the form

$$H(k_i) = A * k_i + B$$

to map the operation codes of the KLIPA assembler into a nearly minimal

---

[1] Morris [1968] provides an excellent survey on the use of hash functions.

hash table. The authors did not provide an algorithmic method for finding such a mapping, but their idea led to more recent investigations by R. Sprugnoli and by G. Jaeschke.

Sprugnoli [1978] presents two algorithmic methods for producing perfect hash functions. The Quotient Reduction method yields a function of the form

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

The second method, called Remainder Reduction, finds a perfect hash function with the form

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

Sprugnoli's methods have limited utility; for sets of more than ten keys, the resulting hash tables are much larger than the number of keys. The author shows that this can be overcome to some extent by partitioning larger sets into segments of about ten keys and computing a perfect hash function for each segment. Applying this idea to the 31 most frequent English words, Sprugnoli produces a minimal perfect hash table in two hours of hand computation by partitioning the keys into four segments.

The most recent publication on the topic of perfect hash functions, by G. Jaeschke [1981], defines a method which he calls Reciprocal Hashing. The form of Jaeschke's hash function is

$$H_j(k_i) = \text{floor}(A/(B*k_i + C)) \bmod D.$$

The strategy used here is similar to Sprugnoli's in that number theoretic properties of the machine character code representation of the keys are used to guide the search for appropriate values of the constants.

Reciprocal hashing produces minimal hash tables, but only for sets of fewer than fifteen keys. Jaeschke, like Sprugnoli, accommodates larger sets by partitioning. He reports producing a minimal perfect hash table for 1003 identifiers partitioned into 163 groups.

All three of these 'numerical' solutions have two undesirable features:

1. the maximum size of the problem set for a single hash function is fifteen keys; and,
2. the solution to the problem is machine dependent: character code representations of the keys are transformed into hash addresses.

A different approach to the problem, based on the assignment of integers to the letters of keys, eliminates both faults.

1.2 Statement of the Problem

The problem we have set ourselves in this research is to develop faster and more general algorithms for finding perfect hash functions of the form suggested by Richard Cichelli [1980].

G. Jaeschke [1980] points out several conditions under which Cichelli's method is inapplicable to given sets of keys. These conditions are all related to the fact that Cichelli uses a fixed set of characteristics to distinguish keys. We set out to find ways of choosing a set of key characteristics so that our methods can be applied to any set of keys. Three characteristics distinguish different keys:

1. the letters which appear in a key;
2. the ordering of those letters; and,
3. the length of a key.

We wish to generalize Cichelli's method by making the choice of the set of key properties used by the hash function depend on the keys given in the problem set.

The process which dominates the cost of using Cichelli's method is the combinatorial search of a large solution space. The cost of this search dictates the upper limit on the size of key sets which can be

processed. We investigate several heuristic search methods in an effort to speed up the search, yet produce hash tables which are nearly optimal. The object of a faster search is to raise the upper bound on the size of the key sets for which perfect hash functions can be found.

Our motivation for undertaking this research is to use perfect hash functions to organize large dictionaries (50 - 70,000 items) for use in computational studies of natural language. Artificial languages for programming and conversational terminal interactions, for example, will also be provided with efficient access to their smaller lexicons using perfect hash tables.

In this thesis we report an experimental study of three algorithms we developed for computing perfect hash functions. We give an informal analysis of our approach to the problem of finding such functions, and then outline the three algorithms we have developed. This is followed by the presentation of experimental results obtained when each algorithm was applied to several sets of keys from natural and artificial languages. These experimental results illustrate the successes and limitations of each algorithm with respect to two factors:

1. the speed with which a solution is found, which determines the upper limit on the number of keys which can be processed by each algorithm;
2. the minimality of the resulting hash tables.

Following the presentation of experimental results, we present conceptual designs for the use of these algorithms for hash tables for large and small key sets in practical systems.

1.3 Cichelli's Method

The research reported in this paper is based on an algorithm recently presented by Cichelli (1980) for computing machine-independent,

minimal perfect hash functions of the form:

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

Cichelli's hash function is machine-independent because the character code used by a particular machine never enters into the hash calculation.

Cichelli's algorithm (Algorithm 0) 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 two-fold ordering process to arrange 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 perfect hash functions for sets of more than about 40 keys. Cichelli's method is also limited since two keys with the same first and last letters and the same length are not permitted.

Cichelli's two-step ordering heuristic first arranges the static set of keys in decreasing order of the sum of frequencies of occurrence of their first and last letters. Note that by sorting the keys we are implicitly sorting the letters in such a way that letters which occur most frequently are the first to be assigned integer values. In his 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. Cichelli's double ordering has the effect of forcing the maximum

number of collisions to occur near the root of the backtrack search tree, where pruning eliminates the largest subtrees and therefore gives the greatest reduction in the cost of finding a solution.

The following is an outline of Cichelli's algorithm:

ALGORITHM 0

step1: compare each key against the rest. If two keys have the same first and last letters and the same length, report conflict and stop; otherwise, continue.

step2: order the keys by non-increasing sum of frequencies of occurrence of first and last letters.

step3: 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.

step4: 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.

We now give an informal analysis of the complexity of Cichelli's algorithm.

step1: as formulated here, this is an $O(N^2)$ computation. The same check can be made by isolating and sorting the first and last letter for each key, then sorting the N keys into lexicographic order on these sets of isolated letters. We can then make one pass through the keys comparing neighboring keys for matching groups of isolated letters. The cost of this procedure would then be dominated by the cost of the lexicographic sort, which can be done in time proportional to $N \log_2 N$.

step2: this initial ordering tallies the frequency of occurrence of first and last letters, which requires one pass through the N keys. A second pass is then made to calculate the sum of frequencies for each key. Sorting the keys into descending order of this sum, the dominant cost of this step, requires time proportional to $N \log_2 N$.

step3: the second ordering is an $O(N^2)$ heuristic. As each key is added to the new ordering, the remaining keys in the old ordering are scanned to decide which, if any, of them now have their hash values determined. This may require $(N-1)*(N-2)/2$ operations.

step4: despite the tendency of the two orderings to reduce the search, for most sets of keys the backtracking phase of this algorithm is the most expensive. An average-case complexity measure is difficult to calculate; Simon and Kadane (1976) estimate an average search to include about one-half the total search space, giving a loose estimate of $O(m^s/2)$, where m is the size of the domain of values for each letter and s is the number of letters which occur in first or last position.

We have found that the time required to find a perfect hash function using this method varies greatly, depending less on the number of keys in the problem set than on the relationships among keys in terms of shared letters. A solution for one set of 61 keys was found in 135 milliseconds of CPU time (for the search), while no solution was found for another set of 64 keys after running the algorithm for over one hour of elapsed time. These trials lend credibility to Knuth's (1975) observation that

> "Sometimes a backtrack program will run to completion in less than a second, while other applications seem to go on forever...A 'slight increase' in one of the parameters of a backtrack routine might slow down the total running time

> by a factor of a thousand...These great discrepancies in execution time are characteristic of backtrack programs, yet it is usually not obvious what will happen until the algorithm has been coded and run on a machine."

Cichelli's algorithm represents a significant improvement on previous work on computing perfect hash functions. Algorithm 0 calculates machine independent perfect hash functions for sets of up to about forty keys, while the methods of Sprugnoli and Jaeschke use machine dependent character codes and can find solutions for no more than fifteen keys in a reasonable amount of computing time. Cichelli's algorithm does, however, have some serious limitations. Although Cichelli cleverly chose as identifying properties the only two letter positions to be found in every key (the first and last), making this a fixed choice severely limits the number of sets of keys which can be accommodated by this algorithm. Although he mentions the problem in his 1980 article, Cichelli makes no clear statement on how he chooses a value of m, the size of the domain of associated letter values. This is an important parameter of the problem since m is the branching factor of the backtrack search tree.

Because Cichelli's algorithm relies on a relatively uninformed exhaustive search of the solution space, the cost of finding a solution can be quite high. This in turn limits the maximum size of the problem sets to which the algortihm can be applied when we set a reasonable upper bound on the amount of computational effort to be expended in the search.

The next chapter presents an informal analysis of the nature of the problem of searching for a perfect hash function. This analysis leads to some methods for overcoming the limitations of Cichelli's strategy while retaining its important benefits.

2. AN INFORMAL ANALYSIS OF THE PROBLEM

In this chapter we present an overview of the problem of calculating perfect hash functions for sets of lexical keys. We identify four main sub-problems:

1. choosing a set of formal properties of the keys to be used in the hashing function;
2. choosing a method of searching the space of possible solutions;
3. ordering the search variables to improve the performance of the search method; and
4. finding ways of enforcing or attaining a reasonable degree of minimality of the solution.

Each of the algorithms we have developed uses a different combination of methods to solve these sub-problems.

2.1 Choosing Hash Identifiers

In order for a hash function to place each key in a different hash table location, the function must work with a uniquely identifying set of key properties. We discuss the structure of lexical keys and a method of choosing an identifying set of properties, which we call a hash identifier.

2.1.1 Combinatorial Properties of Lexical Keys

The data objects utilised in the algorithms reported in this thesis are assumed to be presented as strings of characters drawn from an alphabet A. Unless otherwise stated, we will assume that this alphabet consists of the twenty-six lower-case English letters. These strings are stored as character arrays in both Pascal and APL (the languages in which our algorithms are implemented). A key is defined to be a sequence

of length no greater than P, made up of symbols from the alphabet A. We assume also that A has an ordering defined on it, which allows us to define a lexicographic ordering on the set of keys.

A given maximum key length P and alphabet A determine a space T of possible keys. If T' = card(T) and A' = card(A), then we can express the cardinality of the set of all possible keys as the sum of the number of keys of length P plus the number of keys of length P-1 plus the number of keys of the lengths P-2...1.

$$T' = A'^{P} + A'^{P-1} + \ldots + A'$$
$$= \mathrm{SUM}(A'^{i}),\ 1 \leq i \leq P$$
$$= A' * (A'^{P}-1)/(A'-1)$$
$$= \mathrm{OMEGA}(A'^{P}) \text{ as } A' \text{ becomes large} \qquad \langle 2.1 \rangle$$

When A' becomes arbitrarily large, the limit of A'/(A'-1) approaches 1. The resultant factor, $A'^{P}-1$, reduces to A'^{P}. Thus T' grows at a rate polynomial in A' and exponential in P.

As an example, consider counting the number of lexical items which are possible when letters are drawn from an alphabet consisting of twenty-six upper case and twenty-six lower case Roman letters plus the hyphen, apostrophe, and, in a different category, the blank. Blanks are used strictly as word delimiters, never as part of a word. Upper case letters may appear anywhere in a key (e.g. a name such as LaVerne 'Boom-Boom' O'Grady); hypens may not appear in first or last position. The apostrophe, or single quote, will be removed if it occurs in both first and last positions; otherwise, it may appear anywhere. Lower case letters may appear anywhere. With these restrictions, if P=10 we can determine T' as follows:

A' = 26 + 26 + 2 = 54, P = 10

A_i' = number of letters which can appear in keys of length i

T_i' = the number of keys of length i

$A_1' = 52, \quad T_1' = 52$

$A_2' = 53, \quad T_2' = 53^2 - 1$

$A_n' = 54, \quad 2 < n \leq P, \quad T_n = T_2' * 54^{n-2}$

The size of the key space T for this example is then

$$T_{10}' = T_2' * 54^8 = (53^2 - 1) * 54^8$$
$$= 203,023,907,440,293,888$$
$$= \text{approximately } 2 * 10^{17} \text{ keys.} \qquad \langle 2.2 \rangle$$

As another example, consider an alphabet of only the 26 lower case Roman letters and a maximum key length of six, with no restrictions on where letters may appear. The size of the key space, T', for this example is

$$\text{SUM}(26^i) = 321,272,407 \text{ where } 1 \leq i \leq 6$$
$$= \text{approximately } 3.2 * 10^8. \qquad \langle 2.3 \rangle$$

These examples demonstrate, using the ordinary alphabet, that even a small maximum key length defines a very large space of possible keys.

2.1.2 Letter Order in Keys

In order to completely distinguish keys, we must treat the occurrences of the same alphabetic symbol a_i in different positions j and k as occurrences of different letters, a_{ij} and a_{ik}. The number of letters which can occur in each position remains the same, although, in a sense, we have changed the number of letters in the alphabet from A' = card (A) to A'' = P * A'. Even with a larger alphabet, the number of possible keys is no larger, since each letter a_{ij} can appear only in position j of a key. The number of keys which can be distinguished when positions of occurrence are taken into account is then exactly the number of keys, T', in the

space of keys T.

Algorithm 0 and Algorithms 1 and 2 (described later) all use hash functions which assign the same value to a letter independent of the letter's position in the key. Keys are then distinguished by the unordered set (i.e., combination) of letters which occur in chosen positions. Throughout this thesis we shall call the identifying properties of a key its hash identifier. Keys which can be distinguished when the positions of occurrence of letters are taken into account may not be distinguishable by unordered sets of letters; examples are the pairs of keys ('on', 'no') and ('loop', 'pool'). Given the same alphabet A and maximum word length P, using unordered sets of the formal properties of keys distinguishes fewer keys than does the use of ordered sets of key properties.

The number of keys which can be distinguished when the sets of properties which are used as hash identifiers are not ordered is given by the expression

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

where CH(n,m) is the familiar "choose" function, defined as

$$CH(n,m) = n!/(m!*(n-m)!).$$

If A'=26 and P=6, then the size of the key space is:

$$\begin{aligned} CH(A'+i-1,i),\ 1 \leq i \leq P &= CH(26,1) + CH(27,2) + \ldots + CH(31,6) \\ &= 906,091 \\ &= \text{approximately } 9 * 10^5 \end{aligned} \qquad \langle 2.4 \rangle$$

Compare this number with the $3.2 * 10^8$ distinguishable keys for the same values of A' and P when order of occurrence is taken into account (See ⟨2.3⟩). Without ordering, only about one in 350 keys in this example key space can be distinguished.

The question of how we will distinguish keys is most important; a perfect hash function is impossible unless every key is distinguished from all others.

Cichelli's algorithm (Algorithm 0) and Algorithm 2 use a static specification of key properties to be used in the hash function; for both algorithms, the key length and the first and last letters are employed to distinguish keys. As we have just shown, this restricts the number of keys which can be distinguished by these two algorithms to CH(A',2) for each key length. If the maximum key length is P, then at most P*CH(A',2) keys can be accommodated by either algorithm.

2.1.3 Hash Identifiers

Algorithm 1 incorporates a procedure which automatically chooses, for each subset of keys of the same length, the smallest set of letter positions which distinguishes each key, when the order of occurrence of letters within a key is disregarded. Since each letter has one associated value regardless of its position of occurrence, a key's hash address is determined by the combination of letters in chosen positions. The number of different (unordered) subsets of letters is much smaller than the key space with the same maximum length, as discussed above, so the number of subsets of the key space which can be processed using this algorithm is restricted.

In order to keep the search for (and, later, the use of) a perfect hash function as simple as possible, we select, in Algorithm 1, the smallest subset of the P letter positions in keys of length P which distinguishes each of the given keys. When the set of keys is small only one letter position may be required. The limit on the number of keys which can be discriminated using only one letter position is A'.

It may be possible to distinguish up to A'^2 keys with two chosen letter positions, but only if 'ab≠'ba'. In Algorithm 1 this distinction is not made, which reduces the number of distinct keys which can be recognised by this method to A'(A'-1)/2 when two letter positions are chosen.

In a set of keys of length P, there are 2^P different subsets of letter positions which can be used to discriminate keys[2]. We can also choose whether or not to use the key length, giving us a total of $2 * 2^P = 2^{P+1}$ possible forms of the hash function. We assume that for sets of keys of varied length, we can fill to the right with either a dummy symbol or one of the symbols from the key, say the last. Note, however, that any symbol found in a selected position must have an associated integer value in order for the hash function to be defined for all keys in the problem set.

The algorithm which makes the choice of letter positions generates trial combinations of one position, then two positions, up to all P positions. When each trial combination is generated, it is tested for its ability to discriminate members of the set of keys. If no two keys have the same letter occurrences in the p selected positions, then the algorithm returns this trial combination as the solution and terminates; otherwise, the next combination is generated. The details of this method will be discussed with Algorithm 1.

Algorithm 3 relies on human judgment to choose a set of characteristics for the hash function. Interactively, the user specifies a set of letter positions and whether or not to include the key length in

---

[2] This number includes choosing none of the positions, which can be a solution only in case we have no more than one key.

the hash function. The program then tests the user's selection for key discrimination, inviting the user to try again if any two keys cannot be distinguished. We will see later that Algorithm 3 takes into account the position of occurrence of letters and therefore has the greatest possible discriminatory power.

2.1.4 Direct Hash Functions

The simplest method of defining a perfect hash function for sets of keys such as those described above is to treat each letter as a digit in a card(S) base number system, then convert the key from a string of letters to a string of digits which can be interpreted as a hash address. This is a direct hash function. Because all keys are distinct, all addresses computed by such a hash function will also be distinct.

In general, the key space is too large and too sparsely populated with keys for any practical problem set to justify the use of a direct hash function. If the keys in a given problem set K are uniformly distributed in the key space T, then their hash addresses will be spread widely through the memory space of any computer (if, in fact, our computer has a large enough memory space to accommodate such astronomically large hash addresses at all). Since most key spaces of practical interest are very large, and in the interest of portability and parsimony we want a compact hash table, direct hash functions are generally impractical[3].

Direct hash functions are a subset of the class of perfect hash

---

[3] If we are in a position to create a set of identifiers for some purpose without demanding that these identifiers express any particular meaning except as internal identifiers for a machine, it may be practical to construct the set so that we can use a direct hash function.

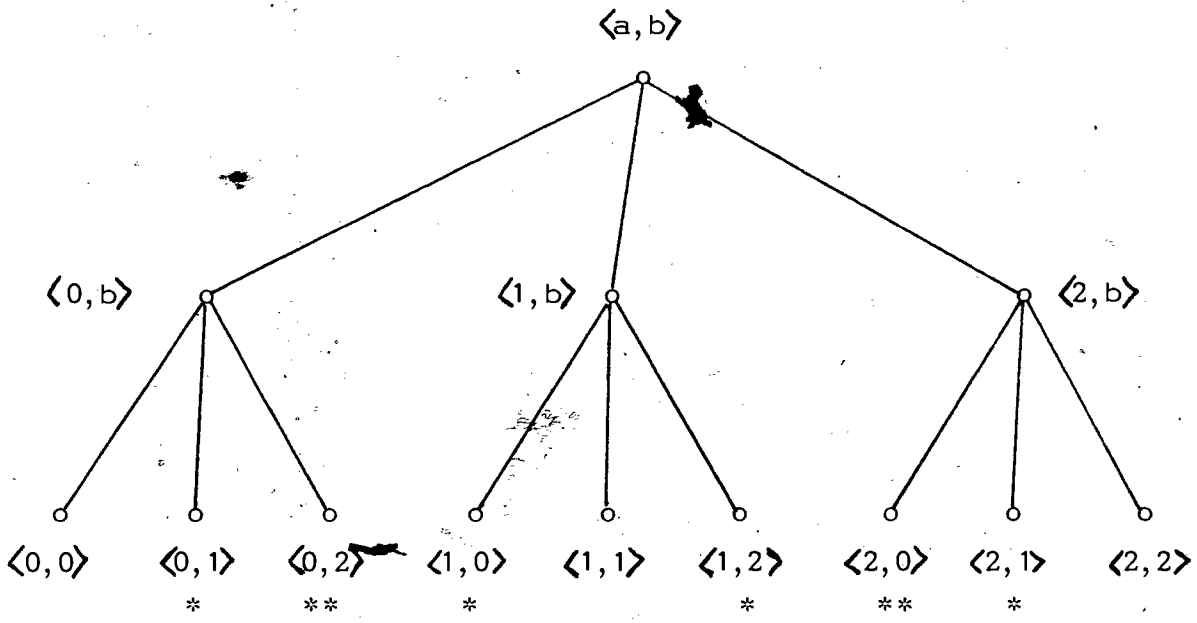
functions. Direct hash functions are advantageous when compared to other perfect hash functions only because they are easy to find. In general, finding a perfect hash function involves searching a very large space of possible functions, a process which may require a great deal of computation when we demand reasonably compact hash tables.

2.2 Assignment of Associated Letter Values

Once a set of letter positions has been selected and it has been decided whether to use the key length as part of the hash function, we consider how to organise the search for an assignment of integer values to the letters which will map the keys into the hash table with no collisions. Although most randomly chosen assignments are unacceptable, many combinations of integer assignments will give us a perfect hash function which is either minimal or almost minimal. An efficient search is necessary to find an acceptable solution in a reasonable amount of time for any but the smallest set of keys.

2.2.1 A Model of the Search Space

If we view the search space as a tree, illustrated in Figure 2.1, there is a path of polynomial cost from the root (the initial state of the search) to each of an exponential number of possible solutions.



Small example search space where the set of keys is (aa,ab,bb), L = 2, M = N = 3, and the letters are ordered ⟨a,b⟩. The leaf nodes are the $m^s = 3^2 = 9$ possible combinations of associated values. Minimal solutions are marked '*', non-minimal ones '**'.

FIGURE 2.1

In Figure 2.1 we have a set of three keys (N=3), two letters from chosen positions (s=2), and a maximum associated value of two (m=3,M=[0,1,2]). The number of different assignments of integers to letters is m^s, the number of leaf nodes in the tree. At tree depth one, the letter 'a' is assigned a value which determines the hash address of the key 'aa'. At depth two, the assignment of a value to 'b' determines the hash addresses of the keys 'bb' and 'ab'.

Our problem is to find a path which leads to an acceptable solution while generating as little of the search tree as possible.

Problems of this type are often attacked using a backtrack search. Our problem has the form of an assignment of values $\langle x_1, x_2, \ldots, x_n \rangle$ to variables $\langle a_1, a_2, \ldots, a_n \rangle$, where each variable corresponds to a letter which appears in a selected letter position. An acceptable solution must satisfy the criterion that the hash function maps each key into a different hash address.

2.2 Importance of Variable Ordering

When the keys are ordered solely according to the sum of letter frequencies, as at the termination of the first ordering in Algorithms 0 and 1, then we may find that a key's hash address is determined early in the search process, yet that key's sum of letter frequencies is small enough to allow several other keys to occur in the ordering between the point in the search where the hash value of the key is determined and the point where its address is tested for a collision with another key.

Suppose, for example, that we are given the set of keys

$$K = (aa, ab, ac, ad, ae, af, ag, bd, cd, ce, de, fg).$$

Both first and second letter positions are necessary to distinguish each key; length can be ignored since all keys are of the same length. Sorting the keys on the sum of the frequencies of their respective first and last letters produces the following ordering:

$$aa, ad, ac, ae, ab, af, ag, cd, de, bd, ce, fg\ .$$

Such an ordering of the keys induces an ordering of the letters from chosen positions. In this example, the order of first occurrence of the letters in the above vector of keys is

$$\langle a, d, c, e, b, f, g \rangle$$

This is the order in which associated letter values will be assigned if the keys are taken in the above order for placement in the hash table.

Note that although the hash address of 'cd' is determined as soon as the key 'ac' is placed in the hash table, the placement of 'cd' is not tested until four intervening keys ('ae','ab','af','ag') have been assigned hash addresses. If 'cd' is found to collide with any of the keys preceding it, then the effort expended between the placement of 'ac' and the attempted placement of 'cd' is wasted. In this situation we are forced to backtrack to place 'ac' in a new location. The remedy for this problem is to order the search variables and the keys in such a way that all possible failure conditions are tested as early as possible during the search.

This example illustrates the problem with using only the first ordering strategy: ordering the keys by sum of letter frequencies does not, in general, produce a strict ordering of the letters in chosen positions. In order to minimise the search expense, we must group the keys relative to the letters so that when each letter a_i is assigned a value, all the tests necessary to decide whether that value is acceptable with respect to the current state of the hash table will be performed before values are assigned to any further letters.

We now consider a more detailed model of our search for an assignment of integers to letters, followed by a discussion of the effect of different search variable orderings on the cost of the search.

The criterion for backtracking; call it predicate Q, can be defined in the following way:

given an assignment of values $\langle x_1, \ldots, x_n \rangle$ to the variables $\langle a_1, \ldots, a_n \rangle$, define

$Q(X_1,\ldots,x_n)$ = FALSE if there exist k_i, k_j, $i \neq j$, in K such that for both keys, all letters in chosen positions are in $\langle a_1,\ldots,a_n\rangle$ and $H(k_i) = H(K_j)$.

= TRUE otherwise.

When $Q(X_1,\ldots,x_n)$ is TRUE, then $\langle x_1,\ldots,x_n\rangle$ represents a perfect hash function for the subset of keys in K for which H now has a value (those for which all letters in selected positions have been assigned a value). The backtrack condition $Q(x_1,\ldots,x_n)$ demands that no two keys have the same hash address. In order to test the predicate efficiently we keep an array of the possible hash addresses where we record which addresses are occupied by keys whose chosen letters have been assigned values previously. When no letter values have been assigned, Q is vacuously true. Suppose that $Q(x_1,\ldots,x_{i-1})$ is satisfied; extending the solution to $Q(x_1,\ldots,x_{i-1},x_i)$ involves two steps:

step1: a value x_i is assigned to a_i;

step2: the set of "new" keys, whose hash addresses are dependent on letters which are all found in $a_1,\ldots,a_i$ must have their hash addresses calculated and compared with the present state of the hash table.

If we let d_i denote the number of keys for which a_i is the last chosen letter to be assigned a value, then each ordering of the letters will in general produce a different vector of values $\langle d_1,\ldots,d_s\rangle$. The sum of these d_i, $1 \leq i \leq s$, is N, the number of keys in the problem set. If we assign unit cost to generating the next trial value for a variable, then the cost of generating the entire tree is the number of nodes in the tree.

The number of nodes in a complete tree with depth s and branching

factor m is the sum of the number of nodes at each level in the tree, where the root is at level 0:

$$C_{max} = SUM(m^i),\ 0 \leq i \leq s$$
$$= m * (m^s-1)/(m-1)$$
$$= O(m^{s+1})$$
$$= OMEGA(m^s) \text{ as } m \rightarrow \text{infinity} \qquad \langle 2.5 \rangle$$

In our application, s is the number of letters to be assigned values and m represents the number of values in the domain of each variable, M = [0...m-1].

Since the size of the domain from which associated values are chosen determines the branching factor of the search tree, it is essential that we choose a value of m carefully. If m is set to too small a value, there may be no solution for the given problem set[4]. If we set m's value high enough, e.g. infinity, then we are assured a solution exists; this certainty incurs a much larger search space and a lower probability of finding a minimal perfect hash function before finding a non-minimal one. Although a search space of infinite size will undoubtedly contain a minimal solution, there is little reason to expect that it will be the first solution found. There must be a smallest value of m such that the search space contains a minimal solution; this value also minimises the size of the search space, so it is important to the efficiency of the backtrack search that we make a wise choice of m. We know of no analytic method of determining the optimum value of m for a given set of keys, although we have obtained impressive results by setting m to N,

---

[4] For example, if only one letter position is chosen for a set of ten keys, then it is clear that any value of m less than ten prevents us from finding a solution.

the number of keys in the problem set. This heuristic consistently leads to finding a minimal or almost-minimal solution, when a solution is found.

If, for example, s=10 and m=10, we have a search space of

$$10^0 + 10^1 + \ldots + 10^9 + 10^{10}$$

nodes, where there are 10^{10} leaf nodes representing complete assignments of values. The size of the tree is the sum of these terms: 11,111,111,111 nodes. Given the assumption of one time unit to generate a new value for a variable, this number represents the cost of generating all the possible solutions. This number remains the same regardless of which ordering is given the search variables.

In order to determine whether the current partial solution satisfies the backtrack predicate Q, one must perform d_j tests at each attempt to extend the solution to the j-th letter; d_j represents the number of keys whose hash addresses are determined by $\langle x_1, \ldots, x_j \rangle$ but not by the assignment $\langle x_1, \ldots x_{j-1} \rangle$. We can therefore assign to each node at depth j a cost of d_j+1, the cost of generating this 'next' value for a_j plus d_j times the (unit) cost of testing the hash address for a key against the present state of the hash table. If c_j is defined as d_j+1, then the cost of visiting every node in the tree is

$$\text{SUM}(c_j * m^j), \quad 0 \leq j \leq s$$

We propose to consider $\langle 2.6 \rangle$, the weighted tree cost [WTC], as a measure of the number of basic operations needed to visit the entire tree, satisfying the predicate Q at each step. Each ordering of the variables determines a (possibly different) value of WTC; we consider that ordering of the variables which give the minimum WTC as the best ordering.

2.3 Ordering Search Variables

We now consider how to choose the best ordering of the s search variables, $a_1,\ldots,a_s$. Given a permutation $B = \langle a_1,\ldots,a_s\rangle$ of the search variables, we define $D(B) = \langle d_1,\ldots,d_s\rangle$ to give the number of keys d_i whose hash addresses are newly determined when a_i is assigned a value. $C(B) = \langle c_1,\ldots,c_s\rangle$ is D(B) with one added to each d_i so that c_i is the cost of visiting any node at level i in the search tree.

We can regard C(B) as a vector of coefficients for the series of terms m^1, $0 \leq i \leq s$, which make up WTC.

$$WTC = SUM(c_i * m^i),\ 0 \leq i \leq s$$
$$= c_0 + c_1 * m + \ldots + c_s * m^s \qquad \langle 2.7 \rangle$$

The initial term, with c_0 defined to be one (unit time expense), is the cost of generating the root node of the search tree. Expanding this root to generate the next level in the tree incurs a cost of m time units, one to generate each of the m values which we assume each variable has in its domain[5]. We add to this total one unit of cost for testing each of the d_1 keys which must be examined to prove that the hash function produces no collisions. This node cost is added to the total cost m times, once for each descendent of the current node, so the cost of visiting all nodes at depth one is $c_1 * m^1$. The m factor in each of the terms in the total cost is growing exponentially with its distance from the root.

Examination of ⟨2.7⟩ should soon convince us that we want the smallest possible values assigned to the coefficients in the order $c_s, c_{s-1},\ldots,c_2,c_1$, where c_s is as small as possible and c_1 is as large as possible. We cannot have $d_s < 1$, since at least one key has the last

---

[5] We are only counting in a breadth-first manner; the backtrack search is done depth-first.

letter in the ordering as its last letter to be assigned a value. The best we can do is to find a letter a_s which has a frequency count of one so that it can be the determining value of only one key, giving c_s a value of two[6].

We can show that m^s is larger than the sum of the remaining terms in the polynomial which describes the size of the search tree:

1. $SUM(m^1) = m * (m^{s-1}-1)/(m-1)$, $1 \leq i \leq s-1$;
2. $m^s = m * m^{s-1}$;
3. $m * m^{s-1} > m * (m^{s-1}-1)/(m-1)$;
4. $m^{s-1} > (m^{s-1}-1)/(m-1)$;

Since m^s will contribute most of the cost of the tree, its coefficient (in the tree of minimum cost) must be the smallest which occurs in any of the s! possible permutations of the variables. We therefore want to find a key which has at least one unique letter occurrence since it is only such a letter which can come last in the ordering and still place a single key in the hash table. As a consequence, the optimal ordering will have all the keys which contain a unique letter occurrence at the end of the key ordering; this follows from the necessity of placing letters which have a unique occurrence at the end of the ordering of letters.

A heuristic ordering strategy for the letters based on this observation would order the letters by frequency in non-increasing order, so that a_1 would have the highest frequency of occurrence and a_s would have the lowest. We find that this arrangement tends to occur when we first order the keys by sum of letter frequencies, then from each key choose the letters which have not occurred before in decreasing order of frequency of occurrence. The second ordering has the effect of making

---

[6] Algorithm 3 uses this strategy explicitly to order the keys. Unique letter occurrences also aid in making the hash table compact.

the coefficients of the m factors of the cost equation increase for the smaller factors and decrease for the larger m factors. Another way of modelling this process is to regard moving a key forward in the ordering as shifting its 'weight' toward the root of the search tree. This tends to reduce the WTC for that tree if the order of letters induced by the new key order has larger weights near the root[7].

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

We can approach the optimal ordering by examining far fewer than s! permutations. This is accomplished by applying a refinement to the second ordering, published by Slingerland and Waugh [1980]. The two authors suggest modifying the key reordering process and, ultimately, the ordering of the s letters which occur in chosen positions, such that:

> "each sublist of words which have equal frequency counts be ordered such that the words that will have the greatest second ordering effect, that is, words that will 'expose' the most words from the rest of the list, occur first."

Slingerland and Waugh report that a program which added their refinement to Cichelli's second ordering showed a consistent improvement in the speed of computation of solutions over Cichelli's second ordering alone. This is explained by our model, since, at each stage in the reordering process,

---

[7] Remember that the sum of these coefficients along any root-to-leaf path in the search tree is the sum of the number of keys and the number of letters which occur in chosen positions, N + s; this is a constant for a given problem instance.

we select the next key whose new letter will determine the greatest number of hash addresses among those keys which have the highest current sum of frequencies. This strategy tends to increase the coefficients of small m factors and therefore decrease the coefficients of large m factors. In turn, this strategy reduces the weighted cost of the search tree, which serves as a good indicator of the relative optimality of the arrangement of search variables.

Note that the WTC indicates only the size of the tree we are searching; it is a measure of the worst case complexity of our problem when we seek only one acceptable solution. The great value of the backtracking approach is that if we test the validity of all partial solutions, when we find that a partial solution $\langle x_1, \ldots, x_i \rangle$ does not satisfy the predicate Q, then we can 'prune' the subtree which has this value of x_i as its root. We thus avoid generating, for a value rejected at level i,

$$\text{SUM}(m^j),\ 1 \leq j \leq s-1 \qquad \langle 2.8 \rangle$$

full and partial solutions which have $\langle x_1, x_2, \ldots, x_i \rangle$ as an initial segment. The cost of this rejected subtree is

$$\text{SUM}(c_{1+j} * m^j),\ 1 \leq j \leq s-i. \qquad \langle 2.9 \rangle$$

This pruning process, usually called preclusion in a backtrack search, is valuable because the subtree of eliminated cases is growing at an exponential rate. In order for preclusion to have its greatest effect, we want to discover at minimal tree depth that some value for a variable is prohibited[8].

---

[8] See page 28.

Fortunately, the frequency of occurrence of a letter a_i is an excellent heuristic value for predicting how likely it is that a_i occurs in a key which may collide with other keys. The sum of letter occurrences for one key is likewise an excellent predictor of how likely it is that that key will collide with other keys.

In general, we may conclude that any polynomial-cost analysis that can be performed dynamically in the depth-first search which allows us to exclude from consideration values in the domain of a search variable will be worth pursuing since an exponentially-growing subtree will be pruned for each potential value we eliminate.

2.4 A Different Search Method

The ideal backtrack search is one which never backtracks. In order to achieve this level of performance, the search must be organised in such a way that a choice made at any stage in the search is known

---

[8] We can show greater advantage of preclusion at depth i over preclusion at depth i+1 in the following way:
Define PR(x) as the number of nodes descendent from a node at level x in the complete search tree. Then we have, from ⟨2.8⟩ above:

$$PR(i) = SUM(m^j),\ 1 \leq j \leq r\ (r = s-1)$$
$$= m^r + m^{r-1} + \ldots + m$$
$$PR(i+1) = SUM(m^j),\ 1 \leq j \leq r-1\ (= s-(i+1)$$
$$= m^{r-1} + m^{r-2} + \ldots + m$$
$$PR(i) - PR(i+1) = m^r = m^{s-1}$$

Recall that for the above sums

$$m^r > m^{r-1} + m^{r-2} + \ldots + m.$$

[i] Minimising the depth i where a preclusion is made maximises the size of the subtree which is pruned.

to be ultimately acceptable.

Blind backtracking (systematically enumerating all possible solutions) performs a small amount of work at each node, but may visit a great number of the nodes in the search tree. The non-backtracking approach we are discussing as an alternative visits only the minimum number of nodes, equal to the number of search variables, but performs a considerable amount of work at each node to ascertain the the value being assigned next meets a set of global requirements which guarantee that an acceptable solution lies at the leaf level of the chosen path.

Backtracking can involve visiting a number of nodes exponential in s, the number of search variables. At each node a constant amount of computation, say c_1, is performed. If a number of possible values at each stage is another constant, m, then the cost of the backtrack serach can be expressed as

$$C_{back} = \text{approximately } c_1 * m^s$$

The cost of performing a non-backtracking search is the sum of the costs of s iterations of 'looking ahead' and assigning a value. Thus we consider how the hash values of the keys which contain the current letter are related to each other and to those hash values which have been assigned (the current state of the hash table). For simplicity, assume that at each stage in the search the hash values of N/s keys are determined (i.e., $d_i = N/s$ for all $1 \leq i \leq s$).

An optimal solution would simply assign to the ordered set of search variables a series of values from some integer sequence which can be calculated independently of which letters actually occur in the set of keys. A large set of integer sequences which can be assigned to produce a perfect hash function is all those of the form $F(n) = b^n$, i.e., the powers of some integer base b. One drawback is that nearly

all these sequences are made up of very large values which promotes production of sparse hash tables.

Although this method gives hash tables which are unacceptably sparse for most sets of keys, such a series of values can be assigned to letters as upper bounds on their associated values. For the i-th letter in the ordering, this upper bound is set to F(i). In the worst case, this method will utilise all F(i) possible values for each a_i, $1 \leq i \leq s$. The use of a series which guarantees distinct sums of pairs of elements does, however, guarantee that this worst case will produce a perfect hash function, although it is generally far from minimal.

2.5 Minimality of Resulting Hash Tables

A minimal perfect hash function for a set of N keys maps those keys into a range of exactly N hash addresses. The loading factor LF of a hash table is defined as the ratio of keys to hash addresses, N/r, for N keys placed in a range of r hash addresses. The loading factor for a minimal hash table is 1. Useful perfect hash functions must presume some lower bound on the loading factor. We have selected 0.8 as the smallest acceptable loading factor.

One heuristic which is applied in all our algorithms attempts to assign the smallest associated values to those letters which occur most frequently in chosen letter positions. The use of this heuristic promotes small hash addresses for many keys.

All hash addresses fall within the range [least..least (N/0.8)]; we can always map the keys into addresses [0..N/0.8].

For the methods which employ backtrack search to assign letter values, Algorithm 0 and Algorithm 1, we can ensure that we achieve a loading factor of at least L by simply limiting the size of the hash

table to r = N/L. The search procedure will then fail on any combination of letter values $(x_1, x_2, \ldots, x_i)$ such that for some k_j in K

$$H(k_j) > N/L$$

When such a failure occurs, the search procedure is forced to backtrack; all values smaller than x_i in the domain of a_i have been excluded and any larger values of x_i will surely make $H(k_j)$ greater than N/L. A variable which occurs earlier in the ordering must have its value altered to allow smaller values for at least one of the letters in k_j which precede a_i in the ordering of letters.

In those methods which do not use the backtrack search paradigm, we must rely on the beneficial effects of the ordering of the search variables and careful selection and testing of associated letter values during the course of the assignment of values to letters to achieve the highest feasible loading factor. In practice these non-backtracking algorithms produce solutions relatively quickly, but as the number of keys in the problem set increases, the loading factor tends to become smaller.

In an effort to improve this performance factor, Algorithm 3 uses backtracking whenever the hash table is becoming too sparsely populated with keys. This will be described in Chapter 3.

2.6 Approaches Employed by Each Algorithm

Each of the four algorithms we will discuss in this thesis employs a different combination of methods for the four subproblems involved in finding perfect hash functions of the form we seek.

Algorithm 0

1. Cichelli makes a fixed choice of key properties to be used by the hash function: first and last letters and key length.

2. The keys are first ordered by sum of chosen letter frequencies, then by a second ordering which moves keys forward to the position where they are first determined.
3. The letter assignment is done by a blind backtrack search.
4. A minimal solution can be found by rejecting any non-minimal solution or by limiting the size of the hash table to the number of keys in the problem set.

Algorithm 1

1. The problem set is partitioned into subsets by key length in order to reduce the size of the problem for each invocation of the backtrack search.
2. Krause algorithmically chooses a combination of letter positions which distinguishes each key in the problem set by the combination of letters from chosen positions.
3. The letters are ordered as in Algorithm 0, but ties between words with equal sums of letter frequencies are broken using a refinement due to Slingerland and Waugh.
4. A letter assignment is calculated using a backtrack search which has been streamlined for this problem.
5. Minimality is not guaranteed, but a high loading factor is obtained by making a good heuristic choice of domain size for the associated letter values and by allowing the ranges of hash addresses for the subsets to overlap to some extent.

Algortihm 2

1. Cercone's algorithm uses the same fixed choice of key properties as Algorithm 0, i.e. first and last letters and key length.
2. The keys are ordered by sum of letter frequencies.
3. A non-backtracking search algorithm is used.
4. Solutions are in general non-minimal, due to the search method employed.

Algorithm 3

1. Boates' algorithm prompts the user to make a choice of key properties, including letter positions and key length if

desired. This choice is sensitive to position of occurrence of letters.

2. Keys are ordered according to sum or product of letter frequencies, then grouped in a way which promotes efficient search.
3. The search is non-backtracking, an improved version of that invented by Cercone for Algorithm 2.
4. The minimality of the hash table is dependent on the effectiveness of the analysis of the way in which keys share letters.

In the next chapter we outline algorithms 1, 2, and 3.

3. DESCRIPTION OF THE ALGORITHMS

In this chapter we describe three algorithms for finding perfect hash functions. For each algorithm, we present a short outline of the processing steps, followed by an informal analysis of the complexity of the algorithm.

3.1 Algorithm 1

Algorithm 1 partitions the original set of keys in such a way that perfect hash functions can be calculated for each subset, then combined to form one perfect hash function for the entire set. The complexity of each subproblem is at least linearly and often exponentially smaller than that of the overall problem, while the increase in the number of problems sets is linear, resulting in a marked reduction in computation.

The keys are partitioned into subsets by their length. We calculate a separate hash function for each subset, and fit each of these hash functions into the hash table by providing an offset for each subset which keeps its hash addresses separate from those of any of the other hash functions. We can do this for the following reason: if $F_i(x)$ is a perfect hash function for keys of length i, then so is $G_i(x) = F_i(x) + c_i$, where c_i is some constant. If we make the proper choice of c_i for each subset i, then $G_i(x)$ can be made nearly minimal by allowing the range of hash addresses for neighboring subsets to overlap. The form of the function is given by the following fragment of Pascal code, where "offset" corresponds to c_i:

```
FUNCTION hashaddress (key: ARRAY OF CHAR): INTEGER;
VAR len, addr, i : INTEGER;
BEGIN
  len := length(key);
  WITH subset[len] DO
    BEGIN
      addr : = offset;
      FOR i : = 1 TO len DO
          IF chosen [i]
            THEN addr : = addr + assoc[key[i]]
      END;  (* WITH *)
   hashaddress : = addr
END;  (* hashaddress *)
```

This program assumes that, for each subset, we record which letter positions are chosen (a vector of boolean values), an integer offset, and a table which associates an integer value with each letter in the alphabet. The calculation of each hash address requires time proportional to the number of chosen letter positions. Because a table of associated letter values is stored for each subset, the increase in storage required is proportional to the number of subsets induced by the partition.

In addition to this partitioning of the original problem set, the implementation of Algorithm 1 includes several of the improvements to Cichelli's algorithm discussed in Chapter 2. These improvements include:

1. the addition of a general algorithm for the selection of a minimal set of letter positions (done for each subset, in Algorithm 1) which distinguishes each key from the rest;
2. choosing the cardinality of a subset as the upper bound on the range of associated values for the symbols which occur in chosen positions in that subset;
3. Slingerland and Waugh's improvement to Cichelli's second ordering of the keys.
4. incorporating a method of precluding the generation and testing of inadmissable combinations of associated values during the backtrack search process.

Algorithm 1 is an informal outline of this modified version of Cichelli's algorithm:[9]

ALGORITHM 1

step 1: Sort the keys into ascending order, by length, in order to partition the keys into subsets of keys which share the same length. With each of these subsets, perform the following steps.

step2: Choose the smallest set of letter positions such that no two keys of the subset have the same set of letters in the chosen positions.

step3: 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.

step4: Assign the cardinality of the current subset as the upper bound on the range of associated values for symbols which appear in chosen positions in members of this subset. The lower bound of this range is set to zero.

step5: Use a modified version of Cichelli's backtrack search procedure to assign associated integer values to symbols such that each key is assigned a different hash address in the subrange of the hash table defined by [offset...(p*m)], where offset is the integer offset for the current subset of keys, p is the number of chosen positions, and m is the upper bound of the range of associated letter values.

---

[9] The Pascal program embodying Algorithm 1 is Appendix D to this document.

step6: If any unproccessed subsets remain, adjust the offset of the next subset in the following way: 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, $r \geq n$. This allows the hash addresses for different subsets to overlap somewhat, encouraging the minimality of the final hash table.

step7: If any unprocessed subsets remain, return to step2; otherwise, all keys have been placed in the hash table and the algorithm terminates.

We now consider the complexity of this algorithm, describing some of the details of each step.

Sorting the keys into order by length is an $O(N \log_2 N)$ operation since we determine the length of each key as it is added to the list of keys initially. We employ a recursive version of Quicksort to order the keys within the array 'keys'. This partition of the keys by length could be done in O(N) time by a bucket sort since the maximum key length is known to be small (usually no more than 15 or 20 letters). This sort is performed once for the entire set of keys.

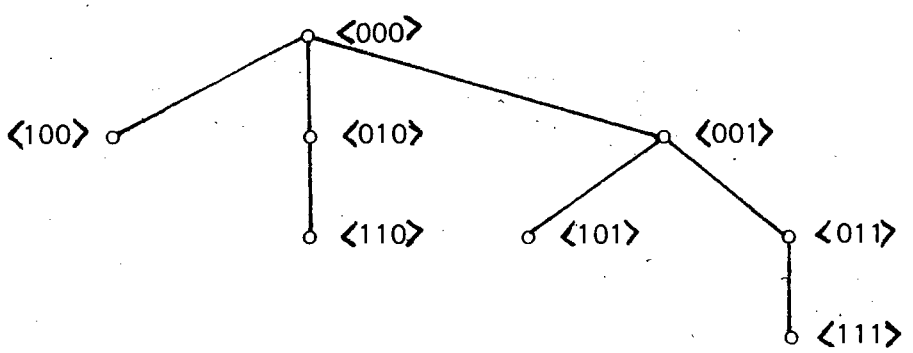
A general method for choosing a set of letter positions which serves to distinguish each key in a subset requires that we test each of the 2^P members of the power set of the set of letter positions for keys of length P. Each of these subsets represents a different combination of letter positions. The Pascal function 'findcombo' generates these subsets in ascending order of number of chosen letter positions since we wish to find that combination which has the fewest elements, yet distinguishes each key. This function returns a vector of boolean values which characterises the chosen positions. This function operates in two phases:

(1) generate the next combination of positions;

(2) test this set for unique identification of keys.

In order to generate these combinations in sequence without storing them all, we begin by creating a queue of combinations which are not candidates, but which can be expanded to produce candidates which have one more chosen position than the combination at the head of the queue. Initially, the queue contains only the null combination, i.e., ⟨000⟩ for P = 3. Unless the subset of keys has only one member, this is not a candidate. The algorithm proceeds by generating a series of new combinations by substituting a 1 for each 0 to the left of all 1's in the combination at the head of the queue. As each of these candidate combinations is generated it is tested for its distinguishing power. If it fails this test and there exists at least one 0 to the left of the first 1 in the sequence, then this combination is placed at the endof the queue, to be expanded later if no suitable candidate is found by expanding and testing combinations which precede it in the queue. As soon as a combination is found which meets our requirements, it is returned as the value of the function 'findcombo'. This method amounts to the breadth-first search of a rather peculiar sort of tree, called an S_k tree[10], a small example of which is illustrated in Figure 3.1.

[10] See Baase [1978], p.247.



Tree S_3

Figure 3.1

Note that the second phase of 'findcombo' tests only whether two keys have different sets of symbols in chosen positions, without regard to the order of occurrence of these symbols within the key. Therefore there may be no way of distinguishing two (or more) of the keys in the subset, e.g. 'on' and 'no'; findcombo may fail. When no combination of positions exists which meets our requirement, 'findcombo' will eventually encounter a situation where there are no expandable nodes in the queue. When this occurs, 'findcombo' returns a vector of boolean FALSE elements, indicating that no solution exists.

The test of phase 2 is carried out in the following manner by the Pascal function 'check':

1. for each key of length p, isolate those symbols which appear in chosen positions according to the current trial combination.
2. for each key, sort that key's selected symbols into ascending order.
3. sort the keys lexicographically using their ordered sets of selected letters as the sort key.
4. make one pass through the set of keys comparing neighboring sets of selected letters. Any sets that match will be neighbors after the sorts in steps 3 and 4, allowing us to test all possible matches in a single pass.

If a match is found in step 4, 'check' will immediately return FALSE, or failure. Assuming random uniform distribution of conflicting letter combinations, we estimate that approximately n/2 comparisons will be executed on the average unsuccessful set of a subset of n keys of length p. The cost of isolating the selected letters is simply the number of keys multiplied by the number of letters in each key. Since the number of selected letter positions cannot exceed the length of the key, sorting the selected letters is done using a simple exchange sort. This adds a term of p*n to the cost.

The lexicographic sort in Step 3 is done for n keys in time proportional to n $\log_2$ n, on the average, by a recursive quicksort routine which uses the array of selected symbols as the sort key.

Each unsuccessful test of a combination of letter positions incurs the following estimated cost:

step1: n*p

step2: n*k*(k-1)/2), $1 \leq k \leq p$

step3: n $\log_2$ n

step4: approximately n/2

Since we may have to try 2^P-1 combinations, the total cost of executing 'findcombo' can be

$$\text{SUM}(CH(P,k)*n*(k*((k-1)/2)+\log_2 n+1/2)),\ 1 \leq k \leq P$$

$$= n(2^P-1) + n*\text{SUM}(CH(P,k)*(k*(k(k-1)/2)+\log_2 n+1/2),\ 1 \leq k \leq P.$$

The cost of this process is proportional to $n*2^P$ for each subset. This search for a minimal uniquely identifying set of letter positions can be very time-consuming whenever the set of keys share many letters. For those few examples we have tried, however, this algorithm produced a solution in time proportional to the number of keys in the problem set. the largest of these sets consisted of 61 keys sharing 23 letters. The

key length P is generally small for even large sets of keys and the likelihood of finding a solution increases rapidly as the number of selected position rises.

The method employed in 'findcombo' is not guaranteed to find a distinguishing set of positions since there are cases where only the ordering of letters from selected positions can distinguish all keys (in conjunction with the occurrence of different symbols). Consider, for example, the set of keys of length two consisting of (in, it, on, no). The possible combinations of chosen positions are (<00>, <10>, <01>, <11>). The null vector cannot be a solution for a key set whose cardinality is greater than one. The keys 'in' and 'it' conflict in position one, 'on' and 'in' conflict in position two, and 'on' and 'no' conflict in the unordered combination of positions one and two. No solution exists for this set, under the assumption that we will have only one value assigned to each symbol for each subset. Greater generality of the algorithm, which would distinguish the occurrence of one symbol in different positions, entails storing an associated integer value for each position in which it is found[11]. Although our method in 'findcombo' does not guarantee a solution, it seldom fails to find one.

Once we have chosen a set of letter positions which provide unique hash identifiers for all keys in the current subset, we consider how best to organise the search of the solution space. In a manner similar to Cichelli, we have used the sum (or product, at times) of frequencies of occurrence of symbols which make up hash identifiers in the current subset. The keys are ordered so that the sum (products) of frequencies are non-increasing, corresponding to Cichelli's first

---

[11] This more general method which recognizes order of occurrence of symbols has been implemented by John Boates in Algorithm 3, to be discussed below.

ordering. The cost of performing this ordering is at most

$$(p*n)+(n \log_2 n) = n*(p+\log_2 n).$$

The second part of step 3 in Algorithm 1 corresponds to Cichelli's second ordering with Slingerland and Waugh's optimising modification added. Since the keys are ordered by decreasing sum (or product) of letter frequencies, we can think of this as inducing a partition of the keys of this length into subsets, where the members of each subset have the same sum or product of frequency counts. The action of the algorithm is as follows:

1. a. each element of the boolean array used [1..A'] is set to false, signifying that none of the A' letters in the alphabet have been added to the ordering of search variables.
   b. a queue which will contain the keys in their final order is initially set to NIL.
   c. "remkeys" is, initially, a list of all the keys sorted by descending sum of letter frequencies. This represents the set of keys which have not yet been placed in the final ordering.

2. a list called "candidates" is formed of all keys which have the highest frequency count. While this list is not empty, we perform the following steps:
   a. for each member of candidates make a list of the keys whose hash address will be determined if the unused letters in the candidate are assigned values.
   b. choose the candidate with the longest list of determined keys; add it to the final ordering, along with its list of determined keys. Remove the candidate and the members of its list from both candidates and remkeys. Mark the letters from selected positions in the newly-chosen keys are used.

Part 2 is repeated until remkeys is empty, i.e. until all the keys have been added to the final ordering. The keys are now inserted in their proper order in the range of the array keys which corresponds to this subset.

When adding the i-th key to the final ordering, we have to examine the possibility of the hash addresses of all remaining n-i keys being newly determined. The cost of part (a) is SUM(n-1), $1 \leq i < n$, or $C = n*(n-1)/2$. The cost of part (b) is simply that of removing each key from the list of candidates and remkeys and adding it to the final ordering, or 3 * N in all. The dominant term in this cost expression is $O(n^2)$. When we consider that the result of this reordering process, a nearly optimal ordering of the variables for the backtrack search, can help us to prune branches from a search tree which is growing at a rate exponential in the ordinal position of the search variables, this $O(n^2)$ cost is well worth considering.

In the Pascal program of Algorithm 1 we now invoke the procedure 'setunused' which prepares the keys for the backtrack search. 'Setunused' scans the keys in the order determined in the previous procedure. For each key, 'setunused' determines which symbols in chosen positions in this key have appeared in no key which precedes the current one. These symbols are recorded in the array 'unused' associated with each key. The number of symbols in 'unused' is then recorded for each key in the integer variable 'numun'. If a key k_i has no unused symbols, the symbols which appear in chosen positions must have occurred earlier in the ordering of keys; each k_i for which 'numun' = 0 has a determined hash address when its predecessor's address has been determined.

The complexity of 'setunused' is proportional to p*n when we keep the array 'used', indexed by symbols of the alphabet. 'Used' is updated as we move through the list of keys, scanning at most p symbols to determine whether this key contains any unused symbols. The cost of 'setunused' is then (p*n) + A', roughly. If we order the keys as we have done, but do not make this record of new symbol occurrences, we

have to isolate these new letters at each trial of each key in a backtrack search, possibly thousands of times[12].

We suggest that the proper method of approaching this entire problem is to explicitly recognize the primary importance of the ordering of the search variables, in our case those symbols which occur in chosen positions. We should program our ordering strategies to choose a heuristically good permutation of the symbols which in turn induces an ordering of the keys. This would certainly promote elegance and likely the efficiency of the program[13].

These ordering functions precede the backtrack search. In Algorithm 1, the search is initiated by invoking the procedure 'assignvalues', which sets the size of the domain from which associated letter values can be chosen before calling the procedures which actually carry out the search. In Algorithm 1, this domain size m is assigned the number n of keys in the subset for which we are currently finding a perfect hash function. We reason that the largest value of m will be required when placing the last key in a subset beyond all other keys in the hash table if all its letters but the last have been assigned the value 0. In this case, there must be an open hash table address somewhere within the range [0...n] since only n=1 keys from this subset have been placed in that range so far. This is not strictly true, since a previous subset may have overlapped the current range somewhat; in

---

[12] This applies only to hash functions which use a variable set of identifying properties. For Algorithms 0 and 2 which both have a fixed set of key properties defined for their hash functions, this selection process can be simply specified in the form of the hash address calculation, e.g. first and last letter positions.

[13] This is corroborated by a study which came to our attention just as this thesis was completed [Cook and Oldehoeft, 1982]. The authors have implemented algorithms which are "letter oriented", with good results.

practice, we have obtained very good results using the number n.

There are three search procedures, 'addword', 'vary', and 'try'; these routines perform the recursive search through the space of partial solutions.

'Addword' is the top-level procedure. It is passed the index i of the next key to be placed in the hash table. If k_i has no unused letters, then its hash address is fixed by values assigned to letters previously; therefore 'try' is called. If at least one of k_i's chosen letters has not yet been assigned a value, 'vary' is called.

'Vary' is the heart of the backtrack search because it assigns values to the search variables in a systematic way, ensuring that all partial solutions of interest are generated. Since the number of unused letters is not fixed it was found most natural to program this as a recursive procedure, allowing backtracking by exhausting all the possibilities for each invocation's search variable. When 'addword' calls 'vary', the number of unused symbols is passed. This parameter, 'u', determines which of the unused letters is assigned a value by an invocation of the procedure; when vary (j) is invoked, the j-th symbol in the array of unused letters for the current key is given a trial value. If j=0, then the current key's hash address is determined and 'try' is invoked; otherwise, we invoke vary(j-1).

The procedure 'try' attempts to place the current key in the hash table. If its hash address is open, then 'try' marks the address taken and calls addword on the index of the nest key in the ordering. If the hash address was taken by a previous key, then 'try' fails and control returns to the calling procedure. When this previous procedure was 'vary', the next larger integer is assigned to the symbol being considered by the latest invocation of 'vary'. If the calling procedure

was 'addword', then the current key's hash address was determined with no unused letters, so control reverts to the previous invocation of 'try' which must have called 'addword'.

The worst-case cost of this approach is the cost of searching the entire tree of partial and complete solutions. We direct our preprocessing efforts to finding an ordering of the search variables which minimises the size of the search tree (the total weighted cost over all vertices). Once the search has begun, we can perform an additional improvement; we maintain a global index value 'firstopen' which is initially the offset for this subset of keys. Whenever a hash address j is newly claimed by some key, firstopen is updated thus:

```
if j = firstopen
    then firstopen := nextopen (j)
```

When the last unused letter for a key is assigned zero as its initial associated value many 'doomed' values may be generated and rejected before an acceptable value is discovered for this symbol (one which places the key in the first open hash address beyond the sum of all the associated values for the other symbols in this key, plus the offset). We can look at this situation on the number line for non-negative integers:

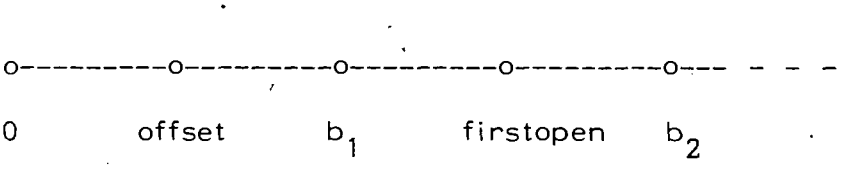


Figure 3.2

Suppose we have a set of symbols (a_1, a_2, a_3) which make up the hash identifier for some key k_i and a_i is the only symbol in the set without an associated integer value:

assocval[a_1] = undefined x_1

assocval[a_2] = x_2

assocval[a_3] = x_3

The problem is to make the optimal choice for the associated value x_1. If the partial sum of associated values $b_1 = x_2 + x_3$, where offset $\leq b_1 <$ firstopen, then we want to set x_1 so that

$$x_1 + x_2 + x_3 + \text{offset} = \text{firstopen}.$$

Then we have:

$$x_1 := \text{firstopen} - (x_2 + x_3 + \text{offset}).$$

This value of x_1 is our best choice since it immediately places the key in the open hash address most likely to lead to a minimal range of hash addresses, given the values assigned to the other letters.

Alternatively, suppose the partial sum of associated values $x_2 + x_3$ is equal to b_2, where $b_2 \geq$ firstopen. Then we set x_1 to the following value in order to place it in the first available hash address:

$$x_1 := \text{nextopen}(x_2 + x_3 + \text{offset}) - (x_2 + x_3 + \text{offset}).$$

This is just a streamlined way of searching for the smallest value of x_1 which places the key in an open hash address, but it does eliminate many procedure calls. We still generate the same number of trial values for x_1, but they are tested in a tight loop in the function 'nextopen', which can be defined recursively as:

nextopen(j) := j, if not taken [j],

:= nextopen(j+1) otherwise, j < table size.

The Pascal version of 'nextopen' is a 3-statement WHILE loop which replaces at least one invocation of 'try'. This eliminates the overhead associated with invoking 'try' for each rejected value. Note that this improvement is possible only in the case where the last letter which is unused for some key is being assigned a new value; where the associated

value of an unused letter does not determine the hash address of a key, we must simply assign an initial value of 0 to the letter. This modification to the search reduced its time of execution by about twenty percent.

When a perfect hash function has been calculated for S_i, the keys of length i, we seek the next subset which hasn't been processed. Suppose that S_j, $j > 1$, is the next subset. When the keys were first partitioned into subsets by length, the smallest array index containing a key of length k, $0 < k \leq P$, was recorded for each subset S_k. The original value of offset for each subset is the number of keys which will have been placed in the hash table prior to placing that subset. Since we place the keys in order of length, this also is a count of the number of keys which are shorter than those in the current subset, S_j.

When we adjust the offset of S_j, offset has the value which anticipates that all previous hash functions have been not only perfect but also minimal. While Algorithm 1 often finds a minimal perfect function for reasonably small sets, our search algorithm does not guarantee minimal solutions. We may find that this original value of offset denotes a hash address which has been taken by a key from the preceding subset, S_i, preventing us from using one of the possible combinations of values which could otherwise place a key from S_j in the hash table. In order to avoid unnecessarily restricting the space of possible solutions, we set the new offset to the smallest index i such that

$$\text{offset} \leq i \leq r \text{ AND not taken}[i],$$

where 'taken' is an array which records for each hash address in the range R[1...r] whether some key has occupied that location. This is done by calculating nextopen(offset). It is quite possible that some

hash addresses with indices greater than the new offset are occupied, but we employ our backtrack search to fit the new subset S_j around any stragglers from S_i.

This heuristic (setting the offset to the number of previously-placed keys) promotes the minimality of the hash table by allowing the ranges of the hash functions for neighboring sets to overlap although they must naturally remain disjoint. Algorithm 1 tends to produce hash tables with very high loading factors. The smallest loading factor so far produced by this algorithm was 0.97 for the two hundred most frequent English words (200 keys placed in a range of 207 addresses).

The cost of performing 'adjustnextoffset' is the sum of the costs of finding the next subset and then finding the next open hash address. The cost of this operation is negligible, less than N for the entire original set of keys.

3.2 Algorithm 2

Algorithm 2 performs an enumerative search of a limited solution space (as do Algorithms 0 and 1). For the backtracking search which was used in Algorithm 1 we assigned a maximum associated value, m, so that for every symbol a_i member of A, the associated value of a_i is greater than or equal to zero and less than or equal to m. This upper bound limits the number of possibilities tried for each search variable. Algorithm 2 chooses an upper bound for each search variable from a series which has no pairwise sums among its elements. This property of the upper bounds for the associated values ensures that a solution will be found, but makes no guarantee that the loading factor of the resulting hash table will be acceptable (though it usually is for small sets of keys).

The essential problem in all the methods of computing perfect hash functions described in this document is the choice of a method for assigning integers to symbols of the alphabet. In order to calculate solutions for large sets of identifiers, the assignment must be done efficiently and must maintain the "non equal" relation between hash addresses of pairs of keys.

$$neq[i,j] = 1 \text{ if } H(k_i) \neq H(k_j)$$
$$= 0 \text{ otherwise.}$$

We consider the possibility of an easily-calculated series of integer values which guarantees distinct sums. If such a seris of integers could be assigned to the search variables in order, then no unsuccessful candidate values would have to be tried and rejected. The difficulty with this approach is to maintain an acceptable loading factor in the hash table. In order to achieve a compact hash address space we seek a series which grows slowly and produced distinct addresses. Consider the assignment of the integers 0,1,2,...,25 to the alphabet 'a,b,c,..,z', respectively. The hash function applied to the word 'program', for example, returns:

hashval('program') = length('program') + assocval('p') + assocval('m')

= 7 + 15 + 12

= 34

It is clear that if the word 'program' appears in a set of fewer than twenty-seven keys and this hash function is used, then the hash table cannot be minimal or even almost-minimal. In addition, many combinations of associated values and word length return a hash address of 34:

an 8-letter word whose first and last letters are 'n'; or, a

4-letter word whose first letter is 'r' and whose last letter is

'n', for instance.

We conclude that a randomly-chosen simple assignment of a series of distinct values to letters will not provide a good solution to our problem.

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

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

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 frequency of occurrence in the list of keys, we obtain a series of distinct values, as desired. This assignment method ensures that all pairwise sums of associated values are distinct as well. The difficulty with assigning associated values in this way is that the magnitude of the series elements rises very rapidly, even if the multiplicative factor m is small; with m=1, the tenth most frequent letter will have an associated value of 1731, which obviously defeats our purpose of producing almost-minimal sized hash tables.

We consider two other series which produce distinct pariwise sums. The first of these is the "powers of 2". Given an integer b greater than one, the series F_b of integers produced by raising this base b to the powers i, i = 0,1,2,..., has all distinct elements $b^0, b^1, b^2, \ldots$ We can show that different elements in the series produce distinct sums by simply noting that b forms the basis of a number system; any two numbers in the base b can be equal only if they are the sum of the

same powers of b when their coefficients are 1. When a key has the same first and last letter, then the coefficient of that letter is 2; but we have already determined that no other key has the same first and last letter and the same length. These are the only non-zero coefficients possible when two letter positions are chosen as is the case in Algorithm 2.

Even for the smallest base, b = 2, this sequence grows rapidly: the tenth letter value will be $2^{10} = 1024$. This technique might be useful for very large sets of keys where the number of letter combinations which actually occur is approaching the theoretical limit, i.e. the key space is densely populated. For small sets of keys, however, assigning values from such a sequence is not practical because of the exponential growth of succeeding elements, which leads to very poor loading factors.

In considering additive integer sequences we looked at using a suitably modified version of the Fibonacci sequence, thus:

$$F_f(1) = 0$$

$$F_f(2) = 1$$

$$F_f(3) = 2$$

$$\vdots$$

$$F_f(n) = F_f(n-1) + F_f(n-2) + 1, \quad n > 3.$$

This sequence is 0, 1, 2, 4, 7, 12, 20, 33, 54, 88,.... The growth rate of this sequence for $n \geq 3$ is, approximately:

$$F_f(n+1) = 1.62 * F_f(n).$$

This low growth rate means that the associated letter values stay considerably smaller than they do for the powers of 2: $F_f(10) = 88$ while $F_2(10) = 1024$. Nevertheless, we cannot make use of this series to simply assign the values of letters; with two chosen letter positions (first and

last for Algorithm 2) and ten letters which occur in a chosen position there can be no more than ten choose two [CH(10,2)], or forty-five, different keys. The maximum hash address which would occur if we assigned associated values directly from the series $F_f(n)$ would be

$$F_f(10) + F_f(9) = 88 + 54 = 142$$

The loading factor in this case will clearly be far too low to qualify the hash function as almost minimal. If we employ the key length as part of the hash function, then the loading factor could be somewhat better, but not enough to make this a viable method for handling most sets of keys where the number of letters which occur in chosen positions is usually nearer twenty than ten.

It would be highly gratifying to find a series of integer values which fits our criteria and produced compact hash tables by just assigning the elements of the series to the letters in order. Unfortunately, this series elude us, and may not exist. Instead, we use the elements of one of the above series to provide an upper bound on the associated values of the letters which occur in chosen positions in the set of keys. This allows us to test all lower associated values, hoping to find an acceptable one, knowing that the upper bound is a solution (although not necessarily a best solution overall). We can therefore avoid all backtracking, leading a faster search algorithm.

Algorithm 2, which uses the same set of key characteristics as Cichelli's algorithm (first and last letters and key length) can be described informally as follows:

ALGORITHM 2

step1: count the frequency of occurrence of each letter which appears in either first or last position in the set of

keys. Order the letters by decreasing frequency of occurrence.

step2: to each a_i in the ordered set of letters assign the upper bound F(i), where F(n) is one of the above series. This assigns the smallest limiting values to the most frequently occurring letters, promoting the minimality of the resulting hash table.

step3: for each key k_i, $0<i\leq N$, calculate tval[i], the sum of the temporary values of first and last letters plus the length of the key. Sort the keys on this sum, producing a list of keys ordered by the sum of the frequencies of their first and last letters; keys with the same combination of letters in these positions will be orderd by increasing length. If any two keys have all characteristics in common, Algorithm 2 cannot be applied to the current set of keys.

step4: for each key k_i, $0<i\leq N$, do the following:

(a) if both the first and last letters in k_i have been assigned values, continue with the next key: k_i has been placed in the hash table previously;

(b) if neither letter has been assigned a value, set the most frequent of them to zero;

(c) if only one letter a_j has no assigned value, vary its associated value from zero to the upper bound until all the keys whose hash addresses are determined by this letter have been placed in open hash table locations. Each time the associated value is incremented, the function 'check' is called which first changes all hash values which are affected by the current letter, then makes an $0(N^2)$ pass through the set of keys to determine whether any pair of keys have the same hash address.

step5: mark the current letter 'tried' and continue at step 4

with the next key in the ordering. If there is none, we have a solution and the algorithm terminates.

Algorithm 2 is not a backtracking algorithm in the classical sense, but an intelligently-controlled enumerative one. This algorithm is designed to have a speedy search which, hopefully, gives almost-minimal perfect hash tables. The search relies entirely on the good effects of the ordering of search variables to achieve a compact solution; unlike backtracking search, this method never "undoes" partial results. Once a key is placed in the hash table, its address never changes.

The cost C of Algorithm 2 applied to a set of N keys of maximum length P which contain s letters which occur in first or last position is calculated by summing the costs of steps 1 through 3 and s instances of steps 4 and 5. The number of keys dealt with in each instance of steps 4 and 5 varies, but the total must be N.

Counting the letter frequencies entails accessing all 2*N characters which appear in chosen letter positions in the set of keys, plus 2*N additions. Ordering the letters by frequency incurs a cost of $0(N \log_2 N)$ using a recursive Heapsort. The total cost of step1 is $0(N * ((\log_2 N) + 4))$.

Calculating the temporary values from the series F(n) is simple no matter which series is chosen. This incurs a cost of 0(s).

Calculation of sort values for N keys costs two additions per keys and is therefore 0(N). The sort is an $0(N^2)$ exchange sort. This could be improved by first sorting the keys using any $0(N \log_2 N)$ sort, then comparing neighbors for equal sort values which would indicate possible irresolvable conflict. Two keys may have the same sort value without having the same combination of letters, but only those which have the same sort value can have matching letters and lengths. We can test for

conflict by comparing only those keys which have the same sort value (and therefore must lie in sequential array locations when the sort is done). As given, the cost of step3 is $0((2*N)+N^2)$.

For a letter a_i which is the last to determine the hash address of a key (or keys), this step may try from one to F(i) associated values, where i is the ordinal position of this letter in the ordering of s search variables. Each of these letters may determine the placement of from zero to N-s+1 keys; let us call d_i the number of keys placed by letter a_i. We know that no more and no fewer than N keys must be placed, so $N = SUM(d_i)$, $0<i\leq s$. The present implementation of this algorithm makes $N*(N-1)/2$ comparisons of hash values to verify that no two keys are mapped into the same hash address each time a new value is tried for a letter. The maximum step4 cost is $s * N * (N-1)/2 * SUM(F(i))$, $0<i\leq s$. The running time of the algorithm 1, once the order of the letters is established we also know in which order the keys will be placed in the hash table. For each letter we can then maintain a list of those keys whose hash addresses are determined when that letter is assigned a value. If we also maintain an array of taken hash addresses, then we need only compare the hash values of the keys in the current letter's list against the corresponding array elements in 'taken' to determine whether a conflict exists. Since we are not interested in solutions with a loading factor of less than 0.8 the size of the array 'taken' need be no greater than 1.25*N; the lists associated with letters must contain a total of N keys.

The cost of marking s letters as 'tried' is s. The total cost of performing Algorithm 2 is

step1: $O(N * (\log_2 N + 4)$

step2: $O(s)$

step3: $O(2 * N) + (N^2))$

step4: $O(s * N * (N-1)/2 * SUM(F(i))), 0<i\leq s$

3.3 Algorithm 3

Algortihm 3 was programmed in the APL language on the MTS/APL system by John Boates. This algorithm finds perfect hash functions rapidly for large sets of keys.

Algorithm 3 incorporates a development and refinement of the non-backtracking enumerative search procedure used in Algorithm 2. No upper bound is placed on the size of associated letter values. Algorithm 3 tends to produce sparse hash tables for the same reasons as for Algorithm 2[14].

Algorithm 3 embodies three improvements over previous attempts:

1. Although the letter positions to be used in the hashing function are not chosen algorithmically (the user is prompted to name a set of positions), this program does distinguish letters by position of occurrence in the assignment of associated values. There is no set of distinct lexical keys which cannot be distinguished in this way.
2. The process of ordering the search variables has been refined and considerably adapted to this problem.
3. The search process is managed in a way which eliminates many doomed choices of associated values by analysing relationships among keys in terms of shared letters.

The following is an informal outline of Algorithm 3:

---

[14] John has now programmed an improved version of this approach which performs a limited amount of backtracking when a solution has a low loading factor. This promises to retain much of the speed of the current version while reducing the size of the hash tables.

Algorithm 3

step1: the user is prompted to supply two specifications: (a) the set of letter positions to be used in the hashing, and (b) whether or not the key length is to be part of the hash function.

step2: If any two words cannot be distinguished with the hash function as specified by the user, then report conflicting keys and return to step1; if the hash function is acceptable, then continue with step3.

step3: count the number of occurrences of each letter in each position, then subtract one from each sum. For each word, assign a value which is the product of the occurrence counts for the selected letters in this key. Those keys whose assigned values are zero must have at least one unique occurrence of a letter in some chosen position. Place these keys at the head of a list of keys with unique occurrences. Repeat step3 for the non-zero keys until no more keys with unique letter occurrences are found. Keys selected in this process will follow all keys with no unique letter occurrences in the final ordering.

step4: order the remaining keys, those with no unique letter occurrences, by decreasing product of their letter frequency counts.

step5: form a group by first choosing the key nearest the head of the list which has the fewest letters with no assigned value ("new" letters); next, find all the keys whose hash addresses will be determined when the chosen key's new letters are assigned integer values. Repeat step5 until all keys have been chosen.

step6: order the keys within each group so that for any two keys k_i and k_j, if we calculate the set differences between the letters from chosen positions in each key ($D_{ij} := L(k_i) - L(k_j)$, $D_ji := L(k_j) - L(k_i)$, where $L(k_m)$

is the set of letters in chosen positions for k_m), then if k_i precedes k_j in the ordering, all letters in D_{ij} will be assigned values before the last letter in D_{ji} is assigned a value.

step7: for each key, determine which of its chosen letters will be the last to be assigned a value in the search. This letter's value can be manipulated to place the key in an open hash address. In cases where length is the only difference between neighboring keys, the distance back to a key which differs in letters is noted.

step8: taking these noted letters in order, we determine for the next letter which of its possible values are precluded by conflict with the hash addresses assigned to previous keys.

step9: assign letter values. If a single key is being placed, its determining letter value is just the one which places it in the lowest possible open hash address. If the key is part of a group whose hash addresses are determined by assigning the current key's hash address, then we must choose the smallest possible value that maps all the keys into open hash addresses.

step10: if no letters (and therefore no keys) remain, then the algorithm terminates. Otherwise, continue with step 8.

There is a description of the original version of this algorithm which permitted no backtracking. A second version of this algorithm has been written which allows backtracking whenever we discover, at step 10, that the hash table has become too sparse. Step 10 is replaced by the following two steps:

step10: the loading factor LF of the partial solution generated to this point is calculated. If the present value of LF

is acceptable, continue with step 8. If LF is too small and the number of allowable backtracks (set by the user in step 1 in response to a prompt is not exceeded, then proceed to step 11).

step11: for the latest group added to the table, determine which keys have the highest and lowest hash addresses; call them k_{max} and k_{min}. Choose a letter from k_{min}, say a_i, which does not affect the hash address of k_{max} and increment the associated values of all letters which were assigned after a_i. Remove from the table all keys which were placed after the value of a_i was assigned, and adjust the sum of assigned letter values for each affected key. Place these keys at the head of the list of keys which have not yet been assigned a hash address. Adjust the order of letters which determine the hash addresses of groups of keys and return to step 8.

Since this algorithm assigns different values to the same letter in different positions, it has the effect of multiplying the size of the alphabet, A, by the number of selected letter positions. We can therefore regard an 'e' in position one as a different letter from an 'e' in position three, for example. As suggested in Chapter 2, this distinction allows us many more possible combinations of integer values which can serve as partitions of the integers which represent hash addresses.

We will also find more keys which have at least one unique occurrence of a letter in a selected position. These unique letter occurrences allow us to place the keys in which they appear anywhere in the hash table beginning at the sum of the values associated with their other selected letters. For this reason, these keys are the last to be placed in the hash table; they can be used to fill hash addresses which were left open during the placement of the preceding keys. The result is a more nearly minimal hash table. We give an informal analysis

of the complexity of each step of Algorithm 3.

Initially, the program interactively prompts the user to name a set of letter positions which will be used in the hash function calculation. The user is also asked to decide whether the key length should be used in the hash function. Step 2 of the algorithm tests this combination of key characteristics to determine whether all keys in the problem set can be distinguished. Potentially, step 1 is repeated until a 'suitable' combination of characteristics is specified by the user. Each iteration of step 1 requires I/O operations, two prompts (letter positions and key length) and two reads of the user's input. We ignore this expense as part of the overall cost.

In order to test the distinguishing power of the hash function specified by the user in step 1, the keys are sorted lexicographically on the chosen letter positions. One pass is made through the sorted array to compare neighbors for matching length (if length is part of the hash function) and matching sets of letters in chosen positions. If such a match is found, the present form of the hash function is rejected and step 1 is executed again. If we assume that only half the keys will be examined on the average in an unsuccessful test, the cost of testing each unsuccessful choice of key properties which define a set of hash identifiers will be:

$$C' = (p*N)+(N \log_2 N)+(N/2)$$
$$= N * (p + \log_2 N + 1/2)$$

The cost of the final iteration of this test, the first successful one, will be:

$$C'' = N * (p + \log_2 N + 1)$$

because all keys will be examined, not half of them.

The minimum cost of this step, which occurs when the user's first choice of identifying properties is successful, is C''. Theoretically, the maximum cost is infinite since the user may repeatedly specify unacceptable hash functions. The maximum number of different hash functions for a set of keys of maximum length P is 2^{P+1}, taking into account whether or not the key length is made part of the hash function. If each of these possible choices is made once and only the final choice is found to be acceptable (the worst likely case), then the cost C of step 2 would be

$$C_{max} = ((2^{P+1}-1)*C')+C''$$

$$C_{min} = C''$$

The user normally finds a suitable set of key properties within a few iterations of steps 1 and 2.

In step 3 all keys which have a unique occurrence of a letter in one of the chosen positions are extracted from the set of keys. In the j-th iteration of this step, counting the occurrences of each letter entails performing $p * n_j$ additions, where p is the number of chosen positions and n_j is defined to be the number of keys which remain after the previous j-1 steps. If we define u_i as the number of keys extracted at the i-th iteration, then

$$n_j = N - SUM(u_i),\ 0 < i < j$$

Decrementing the count for each letter involves performing A_j' subtraction operations, where

$$A_j' = A' - SUM(u_i),\ 0 < i < j$$

For each key we now calculate a temporary value t at a cost of $p * n_j$ multiplications. Keys with unique letter occurrences will have t=0. These keys are each added to a new list, call it U, comprising all those which have at least one unique occurrence of a letter in a chosen position.

If U' = card(U), then the cost of this scan of the keys is $n_j + (2 * u_j)$ in the j-th step. The second term represents the cost of deleting keys from the list K and adding them to U. The total cost of k iterations of step 3 is then

$$C_{total} = SUM(C_j),\ 1 \leq j \leq k$$

where

$$C_j = (p * n_j) + A_j' + (p * n_j) + n_j + (2 * u_j)$$
$$= (n_j * (2 * p)+1) + A_j' + 2 * u_j.$$

Completion of step 3 leaves two disjoint sets of keys, U and W. The set U contains those keys with unique letter occurrences, and W contains those which have no such unique letters.

In step 4, the W' members of W are sorted into non-increasing order of their values of t (the product of letter frequencies), so that the keys with the highest-frequency letters will occur early in the final ordering. The cost of step 4 is $O(W' \log_2 W')$. This step corresponds to Cichelli's first ordering.

Steps 5, 6, and 7 taken together perform Cichelli's second ordering for W. These three steps move keys from W to a new ordering Y which, with U appended, will form the final ordering of the keys.

In step 5, we choose the next key in the ordering, mark its new letters "used", and finally scan the remaining keys for those whose hash addresses will be newly determined by the choice of values for the chosen key's new letters. These keys are deleted from the old ordering and placed in the new ordering Y as a group. This step is repeated until W is empty.

The next key chosen will be that with the smallest number n of "new" letters which occurs earliest in the old order. This can be done in no more than p*W' operations by beginning at the head of the old

order and, for each key in turn, calculating the number n of new letters in the key. If n=1, then choose this key[15]. Otherwise, if the value of n for the current key is less than the minimum value found in the list so far, set the new minimum value to n and note the location of the key which has this value. If we exhaust the list W and found no key with n=1, we select the key whose value of n was the minimum as noted in scanning the list. In either case, at least one key must be examined[16]. No more than W' keys will be examined.

The selected key will now have each of its n new letters marked 'used', at a maximum cost of p assignments. This key is removed from the list W leaving W'-1 keys. The remaining keys in W are examined to find any whose hash addresses are determined. This requires time to test p*(W'-1) letters to see whether they are now used. Those keys in which all letters are used are associated with the key chosen in the first part of this iteration of step 5 to form a group G_i if this is the i-th iteration; anywhere from zero to W'-i keys may be selected in this process. Each of the keys selected for this group must be deleted from W and added to G_i.

The number of iterations of this step (i.e., the number of groups formed) varies, depending on how letters are shared among the keys. Let us say that k iterations are required to group all W' keys from the original ordering W into G groups, G_i, $1 \leq i \leq k$[17]. In the worst case,

---

15 At this point every key must have n>0, otherwise its hash address is determined, so the key should have been made part of some group at an earlier stage. The minimum value of n is therefore one, allowing us to immediately select the first key which has n=1.

16 We can simply choose the key at the head of the list since all keys have p new letters before the first key is chosen.

17 Note that k keys chosen because they add the fewest new letters must contain all the letters in A; therefore $N \geq k \geq (A'/p)$.

k=W'; only one key is selected at each iteration, implying that we search the remaining W'-i keys each time without removing any of them from the list because of having all their letters used. This leaves the maximum number, W'-i, to be searched at the next iteration.

Finding the key with the minimum number of new letters has been facilitated by placing keys containing frequently occurring letters near the beginning of the list. In the worst case, the process would consistently find the next key after scanning W, at a cost of SUM(i), $1 \leq i \leq W'$. This cost is then

$$C_a = \text{SUM}(i),\ 1 \leq i \leq k$$
$$= k*(k+1)/2$$
$$= O(k^2).$$

It seems quite likely that the key at the beginning of the list will often be chosen immediately because it has only one new letter, due to the beneficial effect of the first ordering of W. If that fails, it will often happen that we find a key somewhere before the end of the list which has a single new letter, allowing us to stop the scan at that point. We conclude that selecting all the initial keys for the k groups requires time C_a, which is proportional to at least W' and at most W'_2.

The second part of this step has a time cost C_b proportional to W'^2 in the worst case since we must scan all W'-i keys after the i-th selection step, $1 \leq i \leq k$. When none of the k iterations removes a key from W, then k=W' and we have the cost

$$C_b = \text{SUM}(i),\ 1 \leq i \leq k$$
$$= k*(k+1)/2$$
$$= O(k^2).$$

If this scanning operation finds j keys with determined hash addresses, then the size of W will be reduced to W'-j for the next

iteration, and there will also be j fewer iterations required. We can say that the cost of part b is proportional to at least W' and at most W'_2. The cases where C_b approaches W'^2 appear to be those where the number of keys is about equal to the number of letters which occur in chosen positions, i.e. the keys in the problem set share few letters. These are also small sets of keys where few positions need be used to distinguish all keys. When the set of keys is large, we tend to find that keys share more of the available letters, which tends to reduce the time required for step 5. The computing time for step 5 is then $0(W'^2)$.

In step 6 we perform G' iterations of the basic process, one for each group in G, the set of groups. There are k = 0(s) groups with an average of W'/k keys in each group. Within each group, we compare each key with all others for a cost of $(W'/s)^2/2$. The basic operation in each comparison is to find, for two keys k_i and k_j, the set differences D_{ij} and D_{ji} between the respective sets of letters from chosen positions. If these sets have cardinality p, then the cost of each set difference calculation is $0(p^2)$. For each pair of keys, this difference must be calculated twice. The total cost of this step is roughly

$$0(s * 1/2 * (W'/k)^2 * p^2)$$
$$= 0(1/2 * p^2 * W'^2/k)$$
$$= 0(p * W)^2/(2 * k)).$$

Since $W' \leq N$ and $k \leq s$, we say that for step 6

$$C_{total} = 0((p * N)^2/s).$$

The task of step 7 is to find, for each key, in what order the letters which are previously used have their associated values assigned. If the key k_i contains one new letter a_j, then that letter will necessarily be the last of the letters in k_i to be assigned an integer value. When the key contains more than one new letter, we must choose

an order in which they will be assigned values. This step provides us with a strict ordering of the search variables, the letters. With each letter a_i a certain number of keys are associated; for these keys, a_i is the last variable to receive a value when a perfect hash function is being calculated.

In order to perform this step we require an array of boolean values, 'used', indexed by the symbols of the alphabet A, which allows us to make one pass through the ordered set of keys. We first record for each key which of its letters are new. We then order the new letters and update the array 'used'. For p chosen positions and N keys, the cost of this part of step 7 is $0(p*N)$.

Another function performed by step 7 is to note for each key how far back we will have to return along the path we have taken toward a solution if the present key is found in a situation of inevitable conflict with some other key. This is done by choosing the nearest previous key which differs in a letter or in length. The cost of this part of step 7 is negligible.

In step 8 we analyse the way in which keys from the current group share letters and the constraints this puts on the choice of associated values. Suppose we have a function $L(k_x)$ defined as in step 6 of the outline of the algorithm:

$$L(k_x) :=: L_x :=: (a_i : a_i \text{ appears in a chosen position in } k_x).$$

Suppose that set difference of two sets of letters is defined as

$$D(L(k_x),L(k_y)) :=: D_{xy} :=: L(k_x) - L(k_y)$$

Assume two keys k_i and k_j are in the same group and a_k is, for both keys, the last letter to be assigned an integer value. Before attempting to choose a value for a_k, we must determine whether the current partial assignment of values produced equal hash addresses of the two keys.

For two keys which share a_k, $H(k_i) \neq H(k_j)$ if and only if $H(L_i - a_k) \neq H(L_j - a_k)$. If $H(L_i - a_k) = H(L_j - a_k)$ then a hash address collision is inevitable because a_k must contribute the same integer value to the hash function calculation for both keys. When this situation occurs, the computation backs up to the nearest previous different key chosen in step 7 in order to change an associated value which can avoid this collision.

Because this type of analysis is not done for the set of keys as a whole, a certain amount of the computation may have to be undone in the above manner. A more complete analysis of these interrelations among keys could avoid all backing up at the cost of a considerable amount of preprocessing time[18]. This can happen when the sum of values associated with members of D_{ij} is equal to the sum for members of D_{ji}, for two keys k_i and k_j, $i \neq j$, where

$$L_i = L(k_i) = D_{ij} \cup (L_i \cap L_j)$$

$$\text{and } L_j = L(k_j) = D_{ji} \cup (L_i \cap L_j).$$

When this situation occurs, the only way to create a difference between the two hash addresses is to alter the associated value for one of the letters in

$$SD_{ij} = D_{ij} \cup D_{ji} = L_i + L_j$$

where '+' represents the symmetric difference of two sets.

In order to leave as much of the partial solution intact as possible,

---

[18] The problem we are likely to encounter when doing a more complete analysis of these relationships is that the number of facts deductible from the constraints grows very large for large problem sets. A. Mackworth [1977] gives a method for maintaining consistency in networks of relations which addresses problems like the one we face here. It is intended as a method of optimizing combinatorial search procedures by doing polynomial-cost preprocessing to eliminate inconsistent values from the domains of the search variables. A recent article by E. Freuder [1982] gives a sufficient condition for backtrack-free search.

we choose to alter the associated value of that letter a_k in SD_{ij} which was most recently assigned a value. When we alter this value, we must regroup all keys which have been placed in the hash table since a_k was assigned a value; these groups are placed at the head of the queue of groups which are waiting to be mapped into hash addresses. A new value is assigned to the letter a_k and the search continues.

The procedure described here is a specialised sort of backtracking. If we were to perform a slightly more sophisticated analysis of the relationships among keys, it would be possible to avoid even this limited degree of backtracking. Some expense would be incurred for both added time and space, but it is possible to predict where such conflicts might occur and to avoid them by simply considering which letters are held in common by every pair of keys in the problem set.

It is essential that Algorithm 3 test for the condition

$$H(D_{ij}) = H(D_{ji})$$

before tyring to assign values to the letters in the intersection $L_i \cap L_j$ of the sets of letters from chosen positions in the two keys. If such a test is not made, this algorithm can enter an infinite loop since there is no upper bound on the values which can be associated with letters. Since no value assigned to the members of $L_i \cap L_j$ can create a difference between the hash addresses of k_i and k_j, this algorithm would simply try larger and larger values for these letters until the computer runs out of larger integers.

As a small example, suppose we have a group of keys whose letters from chosen positions are the following: ⟨rig, fig, fog⟩ where 'g' is the last unused key for this group and the letters are assigned associated values in the following relative order: ⟨r, f, i, o, g⟩.

This agrees with the ordering of the combinations of letters above. Some reflection should convince us that before we come to consider a value for 'g' we must see that none of the partial sums of associated values for the subsets of letters <ri, fi, fo> create collisions. If any pair of these subsets have equal sums of associated values, then a hash address conflict is inevitable, if the three keys have the same length, no matter what value we assign to the letter 'g'. We must therefore look ahead and be certain that when the letter 'f' is being assigned a value that 'r' and 'f' do not have the same associated value ('rig' and 'fig' would otherwise be indistinguishable). When an assignment is made to 'o' we must make it different from the value of 'i' to avoid a conflict between 'fig' and 'fog'. For each letter, information about relationships between letter values must be preserved in order to preclude choices of associated values which lead to hash value collisions.

This conflict condition must be tested. The question is whether it is done before associated values are assigned, or dynamically during the search process. In either case, only keys which are in the same group are tested in this way by Algorithm 3. If we assume that the N keys are spread evenly over k groups $G_1, \ldots, G_k$, where n_i = card (G_i), then the estimated cost is

$$C = \text{SUM}(n_i*(n_i-1)/2),\ 1 \leq i \leq k$$

$$= 1/(2k) * \text{SUM}(n_i*(n_i-1)),\ 1 \leq i \leq k$$

$$= \text{approximately } N^2/2k.$$

The total cost for step 7 is $0((p*N)^2)$.

The common characteristic of keys within a group is that the same letter, say a_i, is the last of the letters in chosen positions from these keys to be assigned a value. Step 8 records information which ensures

that the contributions of the other terms of the hash function are different for each key within a group.

In step 9 we actually choose the least value for letter a_i which will map all d_i members of G_i into open hash table locations. This is essentially a kind of pattern-matching step. In step 8 we recorded the relative values of the partial sums for the keys in each group. The aim is to find the first arrangement of open hash addresses in the table which fits the relative hash values of the keys in the current group.

Consider the set of hash identifiers

(abc, adc, abe)

which have the common letter 'a'. Suppose that the current state of the solution is:

```
assocval('b') = 4        assocval('dc') = 7
assocval('c') = 5        assocval('bc') = 9
assocval('d') = 2        assocval('be') = 10
assocval('e') = 6
```

HASH TABLE

...5	6	7	8	9	10	11	12	13	14	15...
....0	0	1	0	0	0	1	0	0	0	0...

If assocval('a') = 0, then H('adc') = 7; but that hash address is taken by some previously-placed key. If assocval('a') = 1, then H('abe') = 11, which is likewise taken. The value 2 is also excluded for 'a', so 3 is the least value which can be assigned to 'a' which places all three keys in open hash table locations.

When considering a value for 'a' in this example, if we are going to place 'abc' at H('abc') = x, then we know from the above relations

between the partial sums of hash values for the keys in 'a''s group that we must also find open addresses at x+2 and x+3 to place all three keys. If "X" represents a "don't care" condition, then our problem is to match the pattern "0X00" against the array of Boolean values representing the 'taken' property for each hash table location.

4. EXPERIMENTAL RESULTS

In this chapter we report the performance of each algorithm discussed in Chapter 3 when applied to several problem sets which are of interest to us. We compare the relative utility of our programs when applied to these selected problem sets.

4.1. Measures of Performance

Performance comparisons of the different algorithms demand that we define useful measures of cost and benefit.

We consider as measures of the benefit of an algorithm the maximum number of keys that can be processed in a reasonable amount of time and the loading factor of the hash table. The larger these numbers are, the better the algorithm.

Three measures of the cost of an algorithm are:

1. execution time of a program embodying the algorithm;
2. the number of basic operations for a given problem set;
3. storage costs, the amount of memory required.

The cost of using an algorithm is most easily obtained by simply timing the execution of the algorithm's implementation when applied to various problem instances. This measure compares the relative utility of our four algorithms.

To make an analytical comparison of the relative performance of our various methods, we compare estimates of how many times certain basic operations are performed. These estimates, which were given in Chapter Three, are only order-of-magnitude measures which indicate how much time the algorithm may have to use to compute an acceptable solution for

problem instance of a given size.[19]

An important element in the cost of using any algorithm is the amount of memory necessary for its execution. In order to make an estimate of the real cost of using any hash function of the type discussed in this thesis, the size of the table of associated letter values must be one term in the cost calculation.

The loading factor measures only the memory used in storing the dictionary itself. A realistic measure of the amount of memory needed for implementing this storage scheme must include the cost of storing the associated value tables since they are essential to calculate perfect hash functions. The Effective Loading Factor [ELF] of a perfect hash function is calculated thus:

$$ELF = N/(r+t),$$

where r is the range of calculated hash addresses and t is the number of associated letter values stored.

It is possible to estimate from a particular solution and knowledge of the ordering of the search variables how many basic operations were performed to find that solution. This information can be used to determine "after the fact" a minimum and maximum amount of work that was done to reach that solution.

We propose a further relative measure of the benefit of an algorithm which we call the utility U of the solution found by a program. When applied to a problem set of size N, we define the utility of the

---

[19] Methods do exist for estimating the computing time of backtracking algorithms, such as the Monte Carlo method [Knuth, 1975]; these are generally defined in terms of estimating the actual size of a search space (as opposed to the theoretical size of the search space). What we wish to determine is not how many solutions exist but the amount of time (or the number of basic operations) necessary to find the first acceptable solution.

solution as

$$U = N * LF / T$$

where LF is the loading factor and T is the search time in milliseconds. A relatively large value of U represents a high degree of utility for an algorithm applied to a key set. We recognize that this is an arbitrary measure, since U represents no concrete cost or benefit; it does, however, reward compact solutions to large problem sets and penalizes the use of a large amount of execution time. The measure U calculated for each of the problem sets and each algorithm is given in Table 4.2 (below).

4.2 Computing Environment

Three of the programs reported here, Algorithms 0, 1, and 2, are written as Pascal programs [Pascal/UBC]; the fourth, Algorithm 3, is written in APL. All of these programs have been run on an IBM 4341 computer under the Michigan Terminal System [MTS] time-sharing operating environment. MTS provides a system subroutine which allows one to measure the amount of cpu time, in milliseconds, which has been used between calls to the subroutine. Only cpu time is included in this total; time-sharing costs such as time to swap programs in and out of memory are excluded. We believe this measure gives a good indication of the amount of time actually used to execute our algorithms.

Algorithm 0 and Algorithm 2 were originally written in UCSD Pascal. In order to compare their performance with that of the APL program for Algorithm 3, algorithms 0 and 2 were translated to Pascal/UBC (the Pascal compiler available on our system). Algorithm 3 was written in MTS/APL and Algorithm 1 in Pascal/UBC.

4.3 Example Problem Sets

In Chapter 2 we showed that the number of sets of keys to which the algorithms discussed can be applied is huge, even for relatively small alphabets and key lengths. We have made no attempt to find solutions for a large sample of the members of this problem domain. We have instead chosen a few examples because they are applications where a perfect hash function is useful.

Several of the examples were first chosen by Cichelli; we have used them as well in order to compare our results with his (although his programs ran on a different machine, a PDP 11/45).

Cichelli reported the results of applying his algorithm (Algorithm 0) to five sets of keys:

1. the 36 Pascal Reserved Words (including 'otherwise');
2. 39 Pascal Predefined Identifiers (excluding 'odd');
3. the 31 most frequently occurring English words;
4. 12 three-letter month abbreviations;
5. the 34 ASCII control codes

One of the obvious applications for a perfect hash function is a table of the keywords in a compiler. Because Pascal was designed to have an efficient compiler, the number of key words in the language is small: there are 35 reserved words (e.g., 'for', 'begin', 'integer') and 40 predefined identifiers (e.g., 'maxint', 'ord', 'new'). For his first example, Cichelli augments the 35 reserved words with 'otherwise', which he hoped would be added to the standard language as a name for the universal default condition for a 'case' statement. We retain 'otherwise' in our examples, giving us a total of 36 reserved words.

The 40 predefined identifiers found in Pascal include two which conflict for both Algorithm 0 and Algorithm 2: 'odd' and 'ord'

have the same length and the same pair of first and last letters. All algorithms were given the set of 39 keys which remain when 'odd' is deleted from the original set. Algorithms 1 and 3 have both found solutions for the set of 40 predefined identifiers and for the set of all 76 keywords of Pascal (the union of the sets of reserved words and predefined identifiers).

In the case of the 12 three-letter month abbreviations, Cichelli found that he could not use first and last letter positions to distinguish the keys 'jan' and 'jun'; instead he used letter positions two and three. Algorithm 0 and Algorithm 2 both follow this course. Algorithm 1 discovers this combination when it selects a minimal hash identifier, while the user of Algorithm 3 should soon find it as well (if he/she had not already selected all three positions, which is a solution as well).

The ASCII control codes represent another interesting possible application of perfect hash functions. Four of the codes contain non-alphabetic characters which are replaced for our algorithms by letters which do not occur in first or last position in any of the original codes. The codes DC1, DC2, DC3, and DC4 become DCJ, DCP, DCW, and DCZ, respectively. In principle, it would be easy enough to avoid these substitutions by including the decimal digits in the alphabet A; this has not been done yet, however.

Many of our examples are taken from lists of the most frequently occurring words in written English text [Dewey, 1923; Carroll, 1971]. Knuth and Cichelli use the first 31 words from these lists, as do we. Our interest in applying perfect hash functions to the design of natural language lexicons for computer text processing and language understanding programs has led us to draw most of our large problem sets from the words which occur with the highest frequencies in the

English language. Thus we have used the 64, 100, 200, and 500 most frequently occurring English words as data when trying to expand the usefulness of our algorithms.

We have used Algorithms 1 and 3 to find perfect hash functions for the key words of three programming languages other than Pascal: these are Lisp, Basic, and Algol W.

4.4 Programming Results

We present a discussion of our experiences using the three algorithms developed in this research. Table 4.1 shows the search times and loading factors obtained using each algorithm on a variety of key sets. We set the upper limit of one hour of cpu time on the amount of computing considered reasonable for finding a perfect hash function.

4.4.1 Results Using Algorithm 1

According to our informal analysis of the complexity of Algorithm 1 in Chapter 3, there are three sections of the algorithm which may require a great deal of computing time. These are

1. choosing a set of letter positions which produces unique hash identifiers;
2. ordering the keys to produce a beneficial ordering of the letters which appear in chosen positions; and
3. the backtrack search process which assigns associated values to letters.

In those cases undertaken so far, the process of choosing a set of letter positions which gives unique hash identifiers has required execution time linear with respect to the size N of the problem set, although the cost in the worst case is exponential in the key length.

Key Set	Algorithm 0	Algorithm 1	Algorithm 2	Algorithm 3
31 Most Frequent English Words	T = 290 LF = 0.97	T = 23 LF = 1.0	T = 2466 LF = 0.94	T = 1763 LF = 1.0
33 Basic Keywords	N/A	T = 16 LF = 1.0	N/A	T = 0.669 LF = 1.0
34 ASCII Codes	T = 1833 LF = 1.0	T = 41 LF = 1.0	T = 6916 LF = 1.0	T = 1993 LF = 1.0
36 Pascal Reserved Words	T = 579 LF = 1.0	T = 29 LF = 1.0	T = 5712 LF = 0.88	T = 2609 LF = 1.0
40 Pascal Predefined IDs	T = 360641 LF = 1.0	T = 30 LF = 1.0	T = 6242 LF = 0.89	T = 3060 LF = 1.0
42 Algol W Reserved Words	N/A	T = 18 LF = 1.0	T = 6046 LF = 0.91	T = 616 LF = 1.0
61 Lisp Identifiers	no results	T = 30 LF = 0.98	no results	no results
64 Most Freq. English Words	T 1 hr	T = 383 LF = 1.0	T = 26619 LF = 0.69	T = 2933 LF = 1.0
76 Pascal Keys Res. plus IDs	no results	T = 68 LF = 1.0	no results	T = 3414 LF = 0.98
100 Most Freq. English Words	no results	T = 10062 LF = 1.0	T = 125973 LF = 0.70	T = 5190 LF = 0.96
200 Most Freq. English Words	no results	T = 62035 LF = 0.97	T = 1505328 LF = 0.42	T = 8986 LF = 0.70
500 Most Freq. English Words	no results	no results	no results	T = 33505 LF = 0.61

Table 4.1

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

The time used for this operation has been, at worst, approximately 3 * N milliseconds. Since Algorithm 3 partitions the keys into subsets by length, the largest set of keys it has had to find a uniquely identifying set of positions for has been the 61 keys of length 3 in the set of two hundred

most frequently occurring English words (MFEW). This was accomplished in 113 milliseconds.

Our analytical estimate for the worst-case time requirement of the process of reordering the keys to reflect the order of search variables was $O(N^2)$. In practice, for the relatively small sets we have encountered using Algorithm 1, the time used in reordering has been linear in the number of keys in the problem set. For subsets of one key, where only the overhead of the process is measured, the time required is 5 milliseconds. As the size of the sets grows, the time increases no faster than 3 * N and seldom more than 5 + (2 * N). The largest cost, 103 milliseconds, occurred for the largest set of keys, where N = 61.

A counterintuitive result was that for most sets of less than 61 keys the backtrack search was performed in time proportional to N. The subset which consumed the largest amount of search time (30,735 milliseconds) for this algorithm was 42 keys of length 3 in the 200 MFEW. The next subset processed was the set of 61 keys of length 4, which required only 42 milliseconds. The contrast in search times for these two sets illustrates the high degree to which the complexity of these problems is dependent on the way in which letters are shared among the keys.

Although Algorithm 1 does not guarantee minimal hash tables, it almost always produces minimal results for the set of keys as a whole. Allowing the ranges of the subsets to overlap means that several almost minimal hash tables can be combined to give a table for the entire problem set which is minimal. In all cases where Algorithm 1 found a solution, the loading factor was well within our criteria for an almost minimal state.

Algorithm 1 appears to be the best for small sets of identifiers; that is, for sets of one hundred or fewer keys. It still performs well for larger sets when it finds a solution, but it often is unable to find a solution for larger sets within a reasonable amount of time. The performance of Algorithm 1 begins to deteriorate when the size of the largest subset in the partition of the key set contains more than about 40 elements. It is at this point that the pattern of sharing letters among the keys of the subsets begins to affect the number of nodes in the backtrack search tree which must be examined.

4.4.2 Results Using Algorithm 2

Algorithm 2 performs well for sets of up to one hundred keys. When the problem sets become larger, the amount of search time required increases quickly. The major objections to using this algorithm are:

1. the loading factors of the solutions produced degenerate quickly for sets of more than about sixty keys, and
2. like Cichelli's algorithm, the mechanism used for distinguishing keys is not adequate for a great number of potential data sets.

The largest set of keys for which we have results using Algorithm 2 is the 200 MFEW. This required over 25 minutes of cpu time and the resulting hash table had a loading factor of only 0.42. This loading factor is far lower than we are willing to accept.

For the smaller problem sets we have tried with this algorithm, the solutions were produced in a reasonable amount of time (generally less than one minute of cpu time).

4.4.3 Results Using Algorithm 3

As described earlier, Algorithm 3 is a refinement of the method

used in Algorithm 2. The improvements included in Algorithm 3 have led to reduced execution time for a given problem set of size N; this in turn allows us to apply Algorithm 3 to problems which are too large for any of the other algorithms discussed in this document. Algorithm 3 is by far the fastest of our algorithms for sets of one hundred or more keys. Like Algorithm 2, however, this program produces hash tables for large sets of keys which are relatively sparse. The upper limit on the number of keys for which the algorithm produces solutions with acceptable loading factors has been raised to about two hundred. For the 200 MFEW, Algorithm 3 finds a solution with a loading factor of 0.71 in just over 5 seconds of cpu time. For smaller sets, the loading factors are nearly optimal, although Algorithm 1 performs comparably in less time for these small problem sets.

Algorithm 3 shows the greatest promise for further development, since it has the most general method of distinguishing the keys and the slowest rate of increase in the time required to find a solution as the size of the problem set increases. Further work is necessary to enhance this algorithm.

4.5 Summary of Results

Table 4.2 illustrates the relative measures of the utility of the four algorithms discussed in this work. Algorithm 1 produces spectacular values of U for problem sets of up to 100 keys. When the sizes of the subsets produced by partitioning reach the neighborhood of fifty keys, however, these values decline. The utility of Algorithm 3 remains nearly constant for all problem sizes.

There can be little doubt that Algorithm 3 promises to be the most

N	ALG 0	ALG 1	ALG 2	ALG 3
12	1333.33	2000.00	65.93	50.42
31	103.69	3100.00	16.71	70.62
33	N/A	2062.51	N/A	49.33
34	0.037	1789.47	7.00	59.96
36	61.22	2571.43	7.63	60.50
39	1.14	N/A	8.95	N/A
40	N/A	2857.14	N/A	56.02
42	N/A	1888.89	6.34	68.18
44	N/A	1466.67	N/A	53.04
61	N/A	1992.67	N/A	N/A
64	N/A	344.09	2.33	59.73
76	N/A	2533.33	N/A	62.97
100	N/A	20.26	0.78	66.10
200	N/A	6.26	0.00785	42.25
500	N/A	N/A	N/A	20.82

Table 4.2

Table of relative utility for four algorithms. Utility is defined as N*LF/T, where N is the number of keys, LF is the loading factor, and T is the time in milliseconds.

useful of the four algorithms implemented in this research. The one difficulty with this algorithm has been small loading factors for large sets of keys. We believe it is possible to improve the loading factors by slightly modifying the search to enforce a certain degree of minimality; this can be effected by limiting the size of the hash table, expanding it slightly as each group of words is added to the solution. Limited backtracking can also have the same results. Preliminary results indicate that a moderate increase in computing costs will produce significant improvement in the loading factors of the solutions found by Algorithm 3.

One of the most important advantages of Algorithm 3 is the guarantee that any set of keys for which the program has an alphabet can be distinguished. This program can therefore find a perfect hash

function for any set of keys, given enough computing time. The cost of this generality is the storage required for up to P * A' associated letter values, instead of just A' values when letter ordering is not taken into account. The benefit is doing away with the necessity for editing lists of keys to remove conflicting words.

5. APPLICATIONS OF PERFECT HASH FUNCTIONS; LEXICON DESIGN

Perfect hash functions will be useful in applications which have a fixed vocabulary and are frequently used. Lexicons for artificial and natural languages meet both criteria.

5.1 The Small Lexicon

For the purposes of this discussion, we differentiate candidates for the application of perfect hash functions according to the size of the sets of keys involved.

We consider a small lexicon as one for which we can calculate a single perfect hash function in a reasonable amount of time. Anticipating the improvement of the loading factors achieved by Algorithm 3, we can say that any set of one thousand or fewer keys qualifies as small.

Although small lexicons can be dealt with by many other methods which require less initial organisational effort than perfect hash functions, these other methods will require considerably more time for retrieving information over the life of the lexicon. The one objection to using the method of Algorithm 3, in particular, is that for very small dictionaries the amount of storage required to maintain the table of associated letter values in main memory may be of the same order of magnitude as the size of the dictionary itself. The potential user of this method will have to decide whether the increased storage costs outweigh the benefit of reduced search time for his or her application, or whether a reduced loading factor would be acceptable.

5.1.1 Examples of Small Lexicons

The earliest example of the use of a perfect hash function which we are aware of was reported by Greniewski and Turski [Greniewski, 1963]. Their method of computing a perfect hash function is fundamentally different from ours, but the application achieves single probe retrieval. The above authors hand calculate a perfect hash function for a set of identifiers for the assembler language of a programming system called KLIPA, with apparently good results.

Cichelli reports [Cichelli, 1980a] using a perfect hash function for the keywords of Pascal in a compiler for that language, which resulted in a ten percent reduction in compilation time, on the average.

It seems clear that the compiler (or interpreter, for languages like LISP and BASIC) for any language would be an ideal application of perfect hashing functions. The number of keywords found in computer languages is usually small (even PL/I has only about three hundred) and usage is very high.

Another application area which meets the requirements for the use of perfect hash functions is command languages. Any system which conducts highly structured conversations with a human user, such as the terminal handling part of an operating system, would benefit from the quick retrieval of information related to the fixed set of keywords which have some meaning to the system.

5.2 The Large Lexicon

If we cannot compute a perfect hash function for a set of keys in a reasonable amount of time, then we will consider that set large.

5.2.1 Hierarchical Organization of the Lexicon

Our plan for dealing with large sets of keys is to partition them into n ≤ ceiling(N/1000) subsets, each of a size manageable with enhanced versions of our present algorithms (Algorithm 3, in particular). We propose to use an ordinary division type hash function to partition the N keys (nearly) evenly into n subsets, then compute a perfect hash function for each subset.

Given a particular set of N keys, the prospective user would need to devise a division type hash function which distributes the keys evenly into the n buckets. In order to maintain the machine independence of our method, we could require that this initial hashing use the sum of the ordinal alphabetic positions of the letters of the key, where 'a'=1, 'b'=2,..., 'z'=26 (with appropriate values for other symbols which may be included in the alphabet, such as hyphen and quote). We assume that finding an initial hash function of this type will be relatively easy, although it will have to be done anew for each problem set to ensure that the distribution is even for that set.

In some cases, a large lexicon may be kept entirely in memory. Each access would require performing an initial hashing to choose one of the n perfect hash functions, then performing an access of primary memory using the value returned by the second hash function as a memory address.

When the lexicon is large enough to demand the use of a secondary storage medium (magnetic disk, for example), then the same basic organisation can be maintained but the second hash function would have to calculate a disk address rather than a primary memory address. The number of disk accesses is always one. The high cost of secondary memory access dictates that we keep the number of such operations to

a minimum.

5.2.2 Natural Language Lexicons

The need for dealing with very large static sets of lexical items arises in many applications which deal with natural languages.

Text processing applications, such as spelling correction, require fast access to large numbers of words. A perfect hash function provides guaranteed single probe retrieval.

Computational Linguistics will someday require large English language dictionaries for natural language understanding programs. A computer model of language understanding will require access to a large data base of syntactic and semantic information which is accessed primarily through words.

Most retrieval techniques assume that all keys are equally likely to be requested.. Research in lexicography has established that in English (and, presumably, any natural language) a small number of words occur very frequently, while most words occur very seldom.

To illustrate this, we refer to the computer assisted study of a corpus of five million words of English written text[20] conducted by Carroll, Davies, and Richman [Carroll, 1971]. Over 85,000 different words occur in the corpus, but 25% of all occurrences are accounted for by the fourteen most frequently occurring words.

In a similar study by G. Dewey [Dewey, 1923], 50% of the occurrences in the corpus are instances of the first 64 words; only 732 words account for 75% of the total of 78,633 occurrences.

We can make use of this information in the design of a lexicon

---

[20] The samples are drawn from materials used in U.S. schools in grades three through twelve.

for a large natural language vocabulary which supports very efficient retrieval. The first step is to calculate a perfect hash function for the thousand most frequently occurring words. The table of associated values and a file containing the records associated with these words can be kept in main memory. We can then use the scheme described earlier to partition the remaining words into buckets containing fewer than one thousand keys each. We then compute a perfect hash function for each bucket and keep the tables of associated values in main memory.

If a word k_i is in the thousand most frequently occurring, as over 75% of word occurrences in typical English text (and speech [French, 1930]) are, we will simply calculate the first perfect hash function H_1 and retrieve the related information from the file in primary memory. If the word at $H_1(k_i)$ is not k_i then we use a second (non-perfect) hash function to select one of the n buckets, thus: $H_b(k_i) = x$, where x denotes one of the buckets. This selects a table of associated values, resident in main memory, for perfect hash function H_x. The hash function for this bucket is used to calculate a disk address $y = H_x(k_i)$, allowing us to retrieve the information at that disk location with one disk access. If the word retrieved from the disk is not k_i, then k_i is not in the dictionary.

We believe that this method, when implemented, will achieve results that are as good as possible. We might consider the addition of a Bloom filter type of spelling checker [Nix, 1981] to be applied before making the secondary memory access; this would save the considerable time required to discover that a key is not in the dictionary at all. A difficulty with doing this may be the amount of memory required for such statistical recognizers, given the amount used for the associated integer values for our n perfect hash functions and a file of one thousand records.

6. CONCLUSIONS

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

6.1 Improvements on Cichelli's Algorithm

One of the major stumbling blocks to the application of Cichelli's methods to practical problems is the fixed choice of properties used in the hash function.

6.1.1 Hash Identifier Choice

When using Cichelli's form of the hash function, it is essential that we find for each key a unique set of properties which contribute values to the hash address calculation. We have implemented an algorithmic method for choosing a set of properties which is adequate for distinguishing the members of any set of lexical items.

Many sets of keys contain groups of words which cannot be distinguished without associating different values with letters depending on their position of occurrence. We have implemented this approach in Algorithm 3, giving it the maximum possible power for distinguishing keys.

6.1.2 Partitioning the Problem Set

Algorithm 1 extends the number of words which can be processed during Cichelli's enumerative search method by partitioning the keys into subsets by length. For a given set of keys, this modification reduces

the search time by a linear factor. Improvements to the search method itself also contribute to the efficiency of the algorithm. Partitioning the problem set is helpful only when the largest subsets obtained consist of no more than about sixty words, however, because of the exponential complexity of the search method applied to those subsets.

6.1.3 Improved Search Methods

The non-backtracking enumerative search method introduced in Algorithm 2 and further developed in Algorithm 3 has allowed us to find perfect hash functions for some very large sets of keys (up to 500). The key to making this method practical is performing an adequate analysis of the relations among keys in terms of shared letters. We find that polynomial cost analysis of the way in which keys share letters allows us to reduce the complexity of the search process from exponential to polynomial (with potentially large coefficients). Unfortunately, the improvement in the speed of search is accompanied by a degradation of the loading factor of the resulting hash table for sets of more than about one hundred keys.

6.2 Limitations of the Method

We find that there is a trade-off in this problem between the time complexity of the search algorithm and the minimality of the resulting hash tables. We are continuing research into methods of balancing these costs and benefits. Introducing the option of a limited amount of backtracking for Algorithm 3 is a promising direction for further work.

Our extensions to Cichelli's algorithm seem to involve trading generality for increased memory requirements. In order to distinguish all keys, for example, we must store associated values for each letter corresponding to each selected position in which that letter occurs. This is a serious limitation, since for small and medium sized dictionaries the number of integer values to be stored in associated value tables may well be as high as the number of keys in the dictionary. This emphasizes the importance of considering the effective loading factor of the hash tables produced using our methods, where the memory cost includes the amount of storage for these tables.

6.3 Practical Applications

We distinguish two types of applications for perfect hash functions, artificial and natural languages. The size of the vocabulary for an aritificial language, such as a programming language, is typically on the order of one hundred words. Perfect hash functions seem well suited for use in such systems since the high initial cost will be amortized quickly by frequent use.

Natural languages have lexicons of thousands of words. We envision that the hierarchical organisation of perfect hash tables discussed in Chapter 5 will provide efficient access to such large dictionaries.

6.4 Directions for Further Work

Algorithm 3 appears at this point to have the greatest potential for improvement. Two parts of this algorithm need further work. The analysis of relations between keys and letters done during the search procedure to preclude choices of associated letter values which lead to

conflict could be made more complete; this would eliminate all backing up in the course of finding a solution.

A second possible improvement to this algorithm, mentioned earlier, would be to perform a limited degree of backtracking when the loading factor of the solution falls below the acceptable limit. We believe it is possible to improve the loading factors by slightly modifying the search to enforce a given degree of minimality; this can be effected by limiting the size of the hash table, expanding it slightly as each group of words is added to the solution. Limited backtracking can also have the same results. Preliminary results indicate that a moderate increase in computing costs will produce significant improvement in the loading factors of the solutions found by Algorithm 3. An initial attempt has been made to implement this approach, with encouraging improvement in the results. Further analysis of the problem may reveal a better way of performing this limited backtracking.

Three mathematical problems are closely related to the search for perfect hash functions of the form we have used:

(1) Harmonious Labeling of Graphs

(2) Graceful Labeling of Graphs

(3) Additive Bases

A brief discussion of how these are related to this work is found in Appendix B.

REFERENCES

Baase, Sarah (1978). Computer Algorithms, Addison-Wesley Publishing Company, Reading, 1978.

Bloom, B. (1970). "Space/time Tradeoff in Hash Coding With Allowable Errors", Communications of the ACM 13, 7 (July), 1970, pp. 422-36.

Carroll, John B., Peter Davies, Barry Richman (1971). The American Heritage Word Frequency Book, American Heritage Publishing Company, Inc., New York, 1971.

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

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

Cichelli, R. (1980). "Minimal Perfect Hash Functions Made Simple", Communications of the ACM 23, 1 (Jan.), pp. 17-19.

Cichelli, R. (1980a). Author's Response to Technical Correspondence, Communications of the ACM 23, p. 729.

Cichelli, R. (1981). "On Peterson's spelling error corrector", Communications of the ACM 24, 5 (May), 1981, p. 322.

Cook, Curtis R. and R.R. Oldehoeft (1982). "More on Minimal Perfect Hash Tables", submitted for publication (presented at the 13th Southeastern Conference on Combinatorics, Graph Theory, and Computing, 1982).

Dewey, G. (1923). Relativ Frequency of English Speech Sounds. Harvard University Press, Cambridge, 1923.

French, N.R., C.W. Carter, Jr., and W. Koenig, Jr. (1930). "Words and sounds of telephone conversations", Bell System Technical Journal 9, 1930.

Freuder, Eugene C. (1982). "A Sufficient Condition for Backtrack-Free Search", Journal of the ACM 29, 1 (Jan.), pp. 24-32.

Graham, R.L., and N.J.A. Sloane. "On Additive Bases and Harmonious Graphs", SIAM Journal on Algebra and Discrete Methods 1,4 (Dec.), 1980, pp. 382-404.

Greniewski, Marek, and Wladyslaw Turski (1963). "The External Language KLIPA For the URAL-2 Digital Computer", Communications of the ACM 6, 6 (June), 1963, pp. 321-24.

Jaeschke, G. (1981). "Reciprocal Hashing: A Method for Generating Minimal Perfect Hashing Functions", Communications of the ACM 24, 12 (Dec.), 1981, pp. 829-33.

Jaeschke, G. and G. Osterburg (1980). "On Cichelli's minimal perfect Hash functions method", technical correspondence, Communications of the ACM 23, 12 (Dec.), 1980, pp. 728-29.

Knuth, D. (1973). The Art of Computer Programming, V.3: Sorting and Searching, Addison-Wesley Publishing Company, Reading, 1973.

Knuth, D. (1975). "Estimating the efficiency of backtrack programs", Mathematics of Computation 29, no. 129 (Jan.), 1975, pp. 121-136.

Mackworth, Alan K. (1977). "Consistency in Networks of Relations", Artificial Intelligence 8 (1977), pp. 99-118.

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

Nix, Robert (1981). "Experience With a Space Efficient Way to Store a Dictionary", Communications of the ACM 24, 5 (May), 1981, pp. 297-98.

Simon, Herbert A., and Joseph A. Kadane (1976). "Problems of computational complexity in artificial intelligence", in J.F. Traub (ed.), Algorithms and Complexity, Academic Press, New York, 1976, pp. 281-299.

Slingerland, Jack E., and Martin D. Waugh (1981). "On Cichelli's algorithm for finding minimal perfect hash functions", technical correspondence, Communications of the ACM 24, 5 (May), 1981, p. 322.

Sloane, N.J.A. (1973). A Handbook of Integer Sequences, Academic Press, New York, 1973.

Sprugnoli, Renzo (1978). "Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets", Communications of the ACM 20, 11 (Nov.), 1977, pp. 841-50.

Widerhold, Gio (1977). Database Design, McGraw-Hill Book Company, New York, 1977.

Appendix A: Notation

Here we give a list of notational conventions used in this thesis. The symbol ':=:' denotes 'defined to be'.

$(e_1,\ldots,e_n)$	:=:	an unordered (but indexed) set of n elements.
$e_1,\ldots,e_n$	:=:	an ordered n-tuple of elements.
floor (x)	:=:	the largest integer less than or equal to x.
ceiling (x)	:=:	the smallest integer greater than or equal to x.
x mod y	:=:	the remainder of integer division of x by y.
O(f(n))	:=:	the asymptotic upper bound on f(n).
OMEGA(f(n))	:=:	asymptotic lower bound on f(n).
$\emptyset$	:=:	the null set.
$A = (a_1,\ldots,a_{A'})$	:=:	the alphabet, where A' is the cardinality of A.
B	:=:	a permutation of the search variables.
C	:=:	cost.
D	:=:	vector of integers which represent the number of keys whose hash addresses are newly-determined when a corresponding letter is assigned an associated value.
D_{ij}	:=:	set difference of letters in chosen positions from keys k_i and k_j, in that order.
F	:=:	names a sequence or a function.
H	:=:	a hash function, H:K→R.
$K = (k_1,\ldots,k_N)$	:=:	the set of keys, whose cardinality is N.
L_i	:=:	the set of letters from chosen positions in key k_i.
M	:=:	domain of associated integer values, 0...m.
N	:=:	cardinality of the set of keys.
P	:=:	the maximum length of a key.
p	:=:	the number of chosen letter positions.
Q	:=:	predicate denoting that no two keys have the same hash address.
R	:=:	the range of hash addresses in the table, 0... maxhashaddr.
T	:=:	the key space, whose cardinality is T'.

Appendix B: Related Mathematical Problems

Graham and Sloane [1980] present some results on three problems which are closely related to finding perfect hash functions:

(1) additive bases;

(2) harmonious graphs; and

(3) graceful graphs.

In order to relate these problems to perfect hash functions, we must first consider the graph representation of our problem.

Since a hash function of the type we are considering involves combinations of letters from chosen positions, our first concern is how to represent the coincidence of letters when p letter positions are used to form a hash identifier. Each vertex in such a graph represents a letter which occurs in a chosen position of some key in the problem set. When p=1, the graph of the problem is just a set of N isolated vertices (for a set of N keys), because each key must have a different letter in the chosen position. A minimal perfect hash function in this case is an assignment of the integers 0,...,N-1 to the vertices in any order.

When p=2 we can represent a key as an undirected edge whose endpoints are the vertices representing the letters in the chosen positions of that key[21]. Each edge is labeled with the sum of the values associated with its endpoints. A perfect hash function corresponds to a labeling of the s vertices with integers which induces a labeling of the N edges with distinct integers.

A harmonious labeling of a graph assigns distinct integers to

[21] When p > 2, a hypergraph whose edges are sets of p vertices is required to represent our problem set. We assume without proof that the results for graphs with p=2 can be generalized to the more complex case of hypergraphs.

the vertices in such a way that the induced edge labeling covers the N integers [0...N-1]. This corresponds to a minimal perfect hash function with the added constraint (unnecessary for our problem) that all associated letter values be distinct. We also allow the possibility that the range of edge labels include more than N values (an almost-minimal perfect hash function).

A graceful labeling associates distinct integers with each vertex of a graph in such a way that the edge labels induced by taking the difference of the endpoint labels produces distinct labels for each edge. Moreover, for N edges, these edge labels must cover the integers [0...N-1]. Graceful labelings again correspond to minimal perfect hash functions.

The paper by Graham and Sloane enumerates some classes of graphs which are or are not graceful or harmonious. The authors theorize that almost all graphs are not harmonious (p.400) and not graceful (p.401). Since our problem relaxes some of the restrictions imposed on harmonious and graceful labelings (vertex labels need not be distinct and the range of edge labels need only be nearly minimal), we conjecture that almost all graphs representing key sets can be given labelings which satisfy our conditions.

We note with interest that Graham and Sloane believe harmonious labeling to be considerably more complicated than graceful labelings. If this is true of the simpler versions of these problems which model perfect hash functions, then graceful labelings may prove to give easier solutions to our problem. This requires further study.

The third problem discussed in Graham and Sloane is additive bases. For our problem, we want a set of integers which covers the s vertices in the graph of the problem set which gives distinct pairwise

sums. This is one type of additive base considered in the article. In the discussion of Algorithm 2 in Chapter 3, we mention seeking a sequence of integers which meets this criterion. In fact, we discovered one (F_c) which matches one of the sets of integers given by Graham and Sloane up to the first six elements (p.384, Table III, $V_a(k)$, k=2 through k=6).

This series, a modified version of the Fibonacci numbers, grows slowly and guarantees pair-wise distinct sums of elements.

The integer sequence F_c is formally defined as:

$$F_c(1) = 0$$

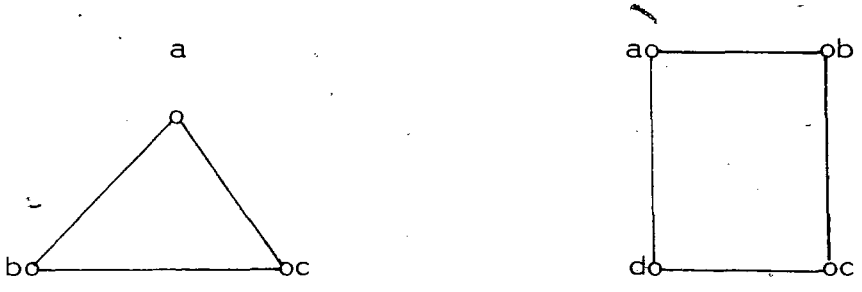
$$F_c(2) = 1$$

$$F_c(n) = F_c(n-1) + F_c(n-2) + 1, \quad n \geq 2.$$

This sequence is number 397 in Sloane [1973]. The first few members of this sequence are

0, 1, 2, 4, 7, 12, 20, 33, 54, 88, ...

The following are two examples of sets of keys whose graphs are complete:



K=(ab, ac, bc) K=(ab, ac, ad, bc, bd, cd)

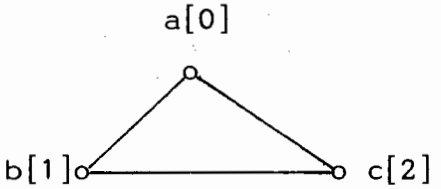
Figure B.1

For a complete graph on n letters the ordering of variables is immaterial because all letters in the graph have the same frequency of occurrence

(all vertices in the complete graph have the same degree, n-1). It is worth noting that no two letters which are both in the same complete graph of more than two vertices can have the same associated value.

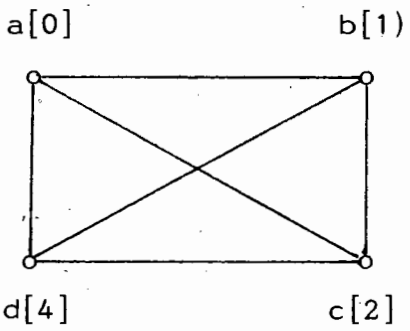
The series F_c, when assigned to the vertices in any order, forms minimal hash functions for complete (sub)graphs on n vertices for n=3 and n=4:

$a := F_c(1) = 0$
$b := F_c(2) = 1$
$c := F_c(3) = 2$
$H(ab, bc, ac) = (1, 3, 2)$



Perfect minimal solution for K_3

$a := F_c(1) = 0$
$b := F_c(2) = 1$
$c := F_c(3) = 2$
$d := F_c(4) = 4$



$H(ab, ac, ad, bc, bd, cd) = (1, 2, 4, 3, 5, 6)$

Perfect minimal solution for K_4

Figure B.2

As the number of letters involved grows larger, the utility of this method declines because the loading factor decreases rapidly. If we assign values from F_c when n=5 (five letters shared among ten keys), the resulting mapping places the ten keys into a range of eleven hash addresses for a loading factor of 0.90909. For n=6, fifteen keys are mapped into a range of eighteen addresses for a barely acceptable laoding factor of 0.7894.

Grahame and Sloane give the following result:

"The complete graph on v nodes is harmonious if and only if v. $\leq$ 4."

This conclusion is stated in a different form by G. Jaeschke [1980] in showing that Cichelli's method cannot find a minimal perfect hash function for every set of keys. We acknowledge that this is true, but complete graphs are relatively rare.

For small sets of keys, an interesting method of approaching the problem of performing associated value assignments would be to search out the maximal clique (complete subgraph) of the graph of relations between keys, then assign values from the series F_c if the clique is of degree less than seven. I have not tried to implement such an algorithm, so there may be unforeseen problems, but it seems to be a reasonable approach until we consider the complexity of finding the maximal clique in a graph. This problem turns out to be difficult. Given an integer k, the problem of finding a k-clique in a grpah $G = (V,E)$, $0 < k < \text{card}(V)$, is NP-complete.[1]

---

[1] Garey, Michael R. and David S. Johnson, Computers and Intractability, W.H. Freeman and Company, San Francisco, 1979.

Appendix C: Output

ALGORITHM 0

Starting at 15:39:08 on Oct. 30, 1981

THIRTY SIX PASCAL RESERVED WORDS

Starting search at 15:39:08

	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	AND	17	14	0
17	REPEAT	18	9	3
18	PROCEDURE	19	10	0
19	FOR	20	8	9
20	MOD	21	18	0
21	GOTO	22	18	0
22	FUNCTION	23	8	7
23	DIV	24	0	21
24	CASE	25	21	0
25	NIL	26	7	16
26	SET	27	21	3
27	WITH	28	3	21
28	CONST	29	21	3
29	IN	30	21	7
30	IF	31	21	8
31	BEGIN	32	20	7
32	VAR	33	21	9
33	UNTIL	34	13	16
34	PROGRAM	35	10	18
35	ARRAY	36	14	17
36	LABEL	37	16	16

PRINTING AT 15:39:10 OCT 30, 1981
addword called 1087 times.
try called 5086 times.
588 milliseconds of CPU time elapsed.

ALGORITHM 1

number of keys: 36
number of subsets: 8
total search time: 29 msecs.

key length	2	3	4	5	6	7	8	9
number of keys	6	9	7	6	4	1	1	2
chosen position	1,2	1,2	1,2	1	4	1	1	1
offset	0	6	15	22	28	32	33	34
search time (msecs)	4	6	7	5	3	1	1	2

KEY	HASH VALUE	CHOSEN LETTERS
OF	0	FO
TO	1	OT
OR	2	OR
IF	3	FI
DO	4	DO
IN	5	IN
NOT	6	NO
END	7	EN
NIL	8	IN
AND	9	AN
FOR	10	FO
MOD	11	MO
VAR	12	AV
DIV	13	DI
SET	14	ES
FILE	15	FI
THEN	16	HT
TYPE	17	TY
WITH	18	IW
CASE	19	AC
ELSE	20	EL
GOTO	21	GO
LABEL	22	L
UNTIL	23	U
WHILE	24	W
ARRAY	25	A
BEGIN	26	B
CONST	27	C
DOWNTO	28	N
RECORD	29	O
REPEAT	30	E
PACKED	31	K
PROGRAM	32	P
FUNCTION	33	F
PROCEDURE	34	P
OTHERWISE	35	O

ASSOCIATED VALUES

	subset 2	3	4	5	6	7	8	9
A	–	3	4	3	–	–	–	–
B	–	–	–	4	–	–	–	–
C	–	–	0	5	–	–	–	–
D	4	5	–	–	–	–	–	–
E	–	1	5	–	2	–	–	–
F	0	4	0	–	–	–	0	–
G	–	–	6	–	–	–	–	–
H	–	–	1	–	–	–	–	–
I	3	2	0	–	–	–	–	–
J	–	–	–	–	–	–	–	–
K	–	–	–	–	3	–	–	–
L	–	–	0	0	–	–	–	–
M	–	5	–	0	–	–	–	–
N	2	0	–	0	0	–	–	–
O	0	0	0	0	1	–	–	1
P	–	–	–	0	–	0	–	0
Q	–	–	–	0	–	–	–	–
R	2	–	–	0	–	–	–	–
S	–	7	–	0	–	–	–	–
T	1	–	0	0	–	–	–	–
U	–	–	–	1	–	–	–	–
V	–	3	–	0	–	–	–	–
W	–	–	3	2	–	–	–	–
X	–	–	–	–	–	–	–	–
Y	–	–	2	–	–	–	–	–
Z	–	–	–	–	–	–	–	–

ALGORITHM 2

Thirty-six PASCAL Reserved Words

Calling 'try' at 01:57:37 on Oct. 16, 1981
Finished at 01:57:54

5712 milliseconds of CPU time elapsed.

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

ALGORITHM 3

WORDS TO BE HASHED

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

LETTERS TO BE USED. I.E. 1 2 FOR FIRST AND SECOND LETTERS
1 2 4
IS BLANK TO BE A CHARACTER. Y/N? : N
IS LENGTH TO BE PART OF FUNCTION Y/N? : Y
ORDER BY PRODUCT OR MINIMUM P/M? : P
CPU SECONDS USED IN HASH IS 1.624

LASHING STARTED AT 1981 10 15 16 44 16 193
CPU SECONDS USED IN LASH IS 1.085

LETTERS USED 1 2 4

LETTER VALUES

'A'	0	13	20
'B'	0	0	0
'C'	0	0	7
'D'	0	0	7
'E'	1	0	1
'F'	0	0	1
'G'	22	0	23
'H'	0	3	15
'I'	0	4	25
'K'	0	0	14
'L'	13	19	6
'M'	12	0	0
'N'	0	5	4
'O'	1	0	0
'P'	0	0	0
'R'	0	4	5
'S'	6	0	26
'T'	3	26	14
'U'	0	21	0
'V'	3	0	14
'W'	5	0	0
'Y'	0	19	0

HASH TABLE

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

APPENDIX D

Programs for Algorithm 1 and Algorithm 3

```
(**-------------------------------------------------------------**)
(**                                                             **)
(**  This PASCAL/UBC program is an implementation of our        **)
(**  Algorithm 1.  The user should note that two features       **)
(**  not found in all versions of PASCAL are employed in        **)
(**  our program:                                               **)
(**                                                             **)
(**   (1) McCarthy evaluation of boolean expressions,           **)
(**         which terminates evaluation as soon as the          **)
(**         value of the expression is determined;              **)
(**   (2) functions are allowed to return a value of            **)
(**         any defined type, not just scalar values.           **)
(**                                                             **)
(**       The user must provide the name of an                  **)
(**  MTS file as the value of the constant fnamefile.           **)
(**  The MTS file of this name must contain the name of         **)
(**  the MTS file which contains the data set (keys).           **)
(**  This first file name, which will be identified with        **)
(**  the standard system file INPUT, must begin in column       **)
(**  one of line one of fnamefile.  On line two we must         **)
(**  place the name of the MTS file which will be ident-        **)
(**  ified with the standard file OUTPUT.                       **)
(**       Before using this program one must verify that        **)
(**  the constants maskeylength, maxnumberofkeys, and           **)
(**  maxtableloc are large enough for the intended set of       **)
(**  keys.  At present, the keys must be provided in            **)
(**  upper case English letters, one to a line, starting        **)
(**  in column one.  A more sophisticated version of            **)
(**  the procedure getkeys is needed.                           **)
(**       It will be somewhat difficult to change the           **)
(**  alphabet for this program.  Because we are working         **)
(**  on IBM equipment with the EBCDIC character code, the       **)
(**  preferred method of indexing for letter information        **)
(**  (using the PASCAL transfer functions) would require        **)
(**  every array so indexed to contain over 150 elements        **)
(**  of which only 26 would be used.  We defined our own        **)
(**  transfer functions, chartonum and numtochar, to get        **)
(**  around this, although rather clumsily.                     **)
(**       The main data structures are keys, an array           **)
(**  of information on each key, and subset, an array of        **)
(**  information on each set of keys of equal length.           **)
(**  The array taken represents the hash table; taken[i]        **)
(**  is true just in case for some key keys[j],                 **)
(**            keys[j].hashval = i.                             **)
(**       The output of this program is a list of letters       **)
(**  and their associated values.  It also prints a list        **)
(**  of keys and their hash addresses, calculated by the        **)
(**  the form:                                                  **)
(**    keys[j].hashval = assoc[keys[j].word[r]] +               **)
(**                      assoc[keys[j].word[s]] +...+           **)
(**                      assoc[keys[j].word[t]]                 **)
(**  where r,s,...,t are the chosen letter positions.           **)
(**                                                             **)
(**-------------------------------------------------------------**)
```

```
CONST maxkeylength = 20;              (* maximum key length *)
      maxnumberofkeys = 300;          (* limit to number of keys *)
      maxtableloc = 400;              (* maximum size of hash table *)
      blank = ' ';                    (* name that symbol *)
      fnamefile = 'FNAMES.ALG1 ';     (* I/O file names in this file *)

TYPE aword = ARRAY[1..maxkeylength] OF CHAR;
     keyinfo = RECORD
                 word : aword;   (* the key *)
                 hashlets : aword;  (* letters from chosen pos'ns *)
                 length :  INTEGER; (* key length *)
                 sortval: INTEGER; (* used for sorting keys *)
                 link :  INTEGER;   (* for threading lists *)
                 unused : aword;  (* letters with no assigned values *)
                 numun : INTEGER (* number of unused letters *)
               END;
     lettervalues = ARRAY[1..26] OF INTEGER;
     boolposn = ARRAY[1..maxkeylength] OF BOOLEAN; (* chosen pos'ns *)
     boolarray = ARRAY[1..26] OF BOOLEAN; (* properties of letters *)
     listrecord = RECORD first, last : INTEGER END;
     setinfo = RECORD
                 exists : BOOLEAN; (* any keys of this length? *)
                 highest, lowest : INTEGER; (* array bnds of this set *)
                 queue : listrecord; (* threaded list of subset *)
                 chosen : boolposn; (* letter pos'ns for this set *)
                 offset : INTEGER; (* low bound of hash addresses *)
                 assoc : lettervalues (* array of letter values *)
               END;
     filename = ARRAY[1..12] OF CHAR;
     alfa1 = ARRAY[1..8] OF CHAR;
     alfa2 = ARRAY[1..12] OF CHAR:

VAR keys : ARRAY [1..maxnumberofkeys] OF keyinfo;
    subset : ARRAY[1..maxkeylength] OF setinfo;
    freq, zfreq, zassoc : lettervalues;
    queue : listrecord;
    used, zuesed : boolarray;
    chartonum : ARRAY['A'..'Z'] OF INTEGER; (*like PASCAL ord fcn *)
    numtochar : ARRAY[1..26] OF CHAR; (* like PASCAL chr function *)
    maxlen, numkeys : INTEGER; (*max. key length, number of keys *)
    infile, outfile : filename; (* names of MTS I/O files *)
    ptime : alfa1; (* for MTS TIME function *)
    pdate : alfa2; (* for MTS TIME function *)
    fx, fpr, msecs : INTEGER; (* for MTS TIME function *)
    zbool : boolposn;
    taken : ARRAY[0..maxtableloc] OF BOOLEAN; (* prop. hash addr's *)
    i : INTEGER;
    wordcount,trycount,addcount,varycount : INTEGER; (* counters *)
    done : BOOLEAN; (*triggers recursive ascent when sol'n found *)
    totmsecs : INTEGER; (* total search time *)
    allblank : aword; (* an array of blanks *)
    firstopen : INTEGER; (* first untaken hash addr.  = offset *)
    maxcharval : INTEGER; (* upper bound on assoc. letter values *)
    linecount : INTEGER;
```

```
  (*-------------------------------------------------------*)
  (* A procedure which reads the names of the input       *)
  (* and output files and then opens them.                *)
  (*-------------------------------------------------------*)

PROCEDURE initio;
  BEGIN
    RESET(INPUT,fnamefile);
    READLN(infile);
    READLN(outfile);
    RESET(INPUT,infile);
    REWRITE(OUTPUT,outfile);
    REWRITE(screen,'*MSINK* ')
  END;  (* initio *)

(*=============================================================================*)

  (*-------------------------------------------------------------*)
  (* A set of procedures which call MTS (FORTRAN) routines       *)
  (* for, respectively, the time, date and elapsed cpu time      *)
  (*-------------------------------------------------------------*)

PROCEDURE time (fx,fpr : INTEGER; VAR ptime : alfa1);
  FORTRAN 'TIME';

PROCEDURE date (fx,fpr : INTEGER; VAR pdate : alfa2);
  FORTRAN 'TIME';

PROCEDURE cpu (fx,fpr : INTEGER; VAR msecs : INTEGER);
  FORTRAN 'TIME';
```

```
(*-------------------------------------------------------------*)
(* Give initial values to chartonum, zbool, zused, zfreq,    *)
(*                        and allblank.                        *)
(*-------------------------------------------------------------*)

PROCEDURE initialise;
  VAR i,j : INTEGER;
      ch  : CHAR;
  BEGIN
    (* start firstopen at 0 *)
    firstopen := 0;
    (* initialise allblank *)
    FOR i := 1 TO maxkeylength DO allblank[i] := blank;
    (* initialise chartonum *)
    i := 0;
    FOR ch := 'A' TO 'I' DO
      BEGIN
        i := i+1;
        chartonum[ch] :=i;
        numtochar[i] := ch
      END
    FOR ch := 'J' TO 'R' DO
      BEGIN
        i := i+1;
        chartonum[ch] := i;
        numtochar[i] := ch
      END
    FOR ch := 'S' TO 'Z' DO
      BEGIN
        i := i+1;
        chartonum[ch] := i;
        numtochar[i] := ch
      END
    totmsecs : = 0;
    (* initialise ZBOOL *)
    FOR i := 1 TO maxkeylength DO zbool[i] := FALSE;
    (* initialise TAKEN *)
    FOR i := 0 TO maxtableloc DP taken[i] := FALSE;
    (* initialise ZUSED,ZFREQ and ZASSOC *)
    FOR i := 1 TO 26 DO
      BEGIN
        zused[i] := FALSE;
        zfreq[i] := 0;
        zassoc[i] := 0
      END
  END; (* initialise *)
```

```
(*-------------------------------------------------------*)
(* Prints the names of the chosen positions for a subset *)
(*-------------------------------------------------------*)

PROCEDURE princhospos (chos : boolposn);
  TYPE ptr = @pos;
       pos = RECORD
               ord : INTEGER;
               next : ptr
             END;
  VAR index,numchosen,temp : INTEGER
      arrow,poslist,tail : ptr;

  BEGIN
    (* empty list *)
    poslist := NIL;
    (* print first part of line *)
    WRITE (' Letter position');
    (* locate first chosen position *)
    index := 0;
    REPEAT index := index+1
    UNTIL index  maxlen OR chos[index];
    IF index   maxkeylength
      THEN WRITELN('s were not chosen. WHY?')
      ELSE BEGIN
             numchosen := 1;
             NEW(poslist);
             poslist@.ord := index;
             poslist@.next := NIL;
             tail := poslist;
             (* add remaining chosen positions to the list *)
             WHILE index   maxlen DO
               BEGIN
                 index := index+1;
                 IF chos[index]
                   THEN BEGIN
                          numchosen := numchosen+1;
                          NEW(arrow);
                          arrow@.ord := index;
                          arrow@.next := NIL;
                          tail@.next := arrow;
                          tail := arrow
                        END  (* then *)
               END  (* while *)
```

```
        (* we now have a list of chosen positions *)
        (* if only one chosen, the subject NP of  *)
        (* our output is singular.                *)
        IF numchosen = 1
          THEN BEGIN
                 WRITE(' ');
                 WRITE(poslist@.ord:1);
                 WRITELN (' was chosen.')
               END  (* then *)
          ELSE BEGIN
                 WRITE ('s ');
                 WRITE(poslist@.ord:1);
                 poslist := poslist@.next;
                 WHILE poslist     tail DO
                   BEGIN
                     WRITE (', ');
                     WRITE(poslist@.ord:1);
                     poslist := poslist@.next
                   END  (* while *)
                 (* handle last item in list *)
                 WRITE (' and ');
                 WRITE(poslist@.ord:1);
                 WRITELN (' were chosen.')
               END  (* if *)
    END  (* if *)
END; (* princhospos *)
```

```
    (*-------------------------------------------------*)
    (* A procedure to print the keys in order by hash *)
    (*-------------------------------------------------*)

PROCEDURE printkeys (len : INTEGER);
  VAR index,gap,j : INTEGER;
  BEGIN
    WITH subset[len] DO
      BEGIN
        WRITELN;
        WRITE ('   Keys of length ');
        WRITE(len:1);
        WRITELN ('.');
        princhospos(chosen);
        WRITELN:
        WRITELN (' ':maxlen+2,'HASH  CHOSEN');
        WRITELN (' KEY':maxlen,' VALUE LETTERS');
        WRITELN (' ---':maxlen,' ------ --------');
        gap := maxlen - len;
        FOR index := lowest TO highest DO
          WITH keys[index] DO
          BEGIN
            WRITE(word:len);
            IF gap   0 THEN WRITE(' ':gap);
            WRITE(sortval:6,'  ');
            j := 1;
            REPEAT
              WRITE(hashlets[J]);
              j := J+1
            UNTIL hashlets[j] = blank;
            WRITELN
          END (* for *)
      END  (* with *)
  END;  (* printkeys *)
```

```
    (*-----------------------------------------*)
    (* print value of offset and associated    *)
    (* values of letters in chosen positions   *)
    (*-----------------------------------------*)

PROCEDURE showletvalues (len : INTEGER);
  VAR 1 : INTEGER;
  BEGIN
    WITH subset[len] DO
      BEGIN
        WRITELN('The offset is ', offset:3);
        WRITELN;
        WRITELN(' Letter Value');
        WRITELN(' -----  ---- ');
        FOR i := 1 TO 26 DO
          IF used[i]
            THEN WRITELN('   ',numtochar[i],'.    ',assoc[i]:2);
        WRITELN;
      END (* with *)
  END;  (* showletvalues *)

(*=============================================================================*)

  (*-------------------------------------------------*)
  (* A function to test a vector of Boolean values   *)
  (* which returns TRUE if all values are FALSE.     *)
  (*-------------------------------------------------*)

FUNCTION allfalse (bvector : boolposn) : BOOLEAN;
  VAR i : INTEGER;
      result : BOOLEAN;
  BEGIN
    i := 0;
    result := TRUE;
    WHILE 1   maxkeylength AND result DO
      BEGIN
        i := i+1
        IF bvector[i] THEN result := FALSE
      END (* while *)
    allfalse := result
  END;  (* allfalse *)
```

```
  (*--------------------------------------*)
  (* A function which returns the number *)
  (* of characters in a left-justified   *)
  (* array of them.                      *)
  (*--------------------------------------*)

FUNCTION findlen (letters : aword) : INTEGER;
  VAR i : INTEGER;
  BEGIN
    i := 0;
    WHILE i   maxkeylength AND letters[iÑ1]    blank DO
      i := i+1;
    findlen := i
  END;  (* findlen *)

(*=========================================================================*)

  (*-----------------------------------------------------------*)
  (* Read the set of keys from input file, set length value, *)
  (* set sortval to length, and determine maximum length     *)
  (*-----------------------------------------------------------*)

PROCEDURE getkeys;
  VAR i : INTEGER;
  BEGIN
    maxlen := 0;
    i := 0;
    WHILE (NOT EOF) AND (i   maxnumberofkeys) DO
      BEGIN
        i := i+1;
        WITH keys[i] DO
        BEGIN
          READLN(word);
          length := findlen(word);
          sortval := length;
          IF length   maxlen
            THEN maxlen := length;
          link := MAXINT
        END  (* with *)
      END  (* while *)
    numkeys :=i
  END:  (* getkeys *)
```

```
(*-------------------------------------------------------------*)
(* Quicksort, useful for sorting subranges of arrays in place *)
(*-------------------------------------------------------------*)

PROCEDURE quicksort (low,high : INTEGER);

  PROCEDURE sort (l,r : INTEGER);
    VAR i,j : INTEGER;
        x,w : keyinfo;
    BEGIN (* sort *)
      i := 1;
      j := r;
      x := keys[(l+r) DIV 2];
      REPEAT
        WHILE keys[i].sortval   x.sortval DO i := i+1;
        WHILE x.sortval   keys[j].sortval DO j := j-1;
        IF i  =j
          THEN BEGIN
                 w := keys[i];
                 keys[i] := keys[j];
                 keys[j] := w;
                 i := i+1;
                 j := j-1
               END
        UNTIL i   j;
        IF l   j THEN sort(l,j);
        IF i   r THEN sort(i,r)
      END;  (* sort *)

    BEGIN (* quicksort *)
      sort(low,high)
    END;  (* quicksort*)
```

```
  (*-----------------------------------------*)
  (* Partition keys into subsets by length. *)
  (* Link lists within subsets.             *)
  (*-----------------------------------------*)

PROCEDURE partition;
  VAR index,next,len : INTEGER;
  BEGIN
    FOR index := 1 TO maxkeylength DO subset[index].exists := FALSE;
    (* set up first subset *)
    index := 1;
    len := keys[index].length;
    WITH subset[len] DO
      BEGIN
        exists  := TRUE;
        lowest  := index;
        queue.first := index;
        offset  := 0
      END  (* with *)
    (* scan sorted keys, comparing neighbors *)
    WHILE index   numkeys DO
      BEGIN
        NEXT := index + 1;
        IF keys[next].length = len
          THEN keys[index].link := next
          ELSE (* end this subset and start new one *)
            BEGIN
              keys[index].link := MAXINT;
              subset[len].queue.last := index;
              subset[len].highest := index;
              len := keys[next].length;
              WITH subset[len] DO
                BEGIN
                  offset := index;
                  lowest := next;
                  queue.first := next;
                  exists := TRUE
                END  (* with *)
            END  (* if *)
        index := next
      END  (* while *)
    subset[len].queue.last := index;
    subset[len].highest := index
  END;  (* partition *)
```

```
(*-----------------------------------------*)
(* A version of quicksort using hashlets *)
(*-----------------------------------------*)

FUNCTION sortalpha (list : listrecord) : listrecord;
  VAR low,high,index : INTEGER;

  PROCEDURE sorta (l,r : INTEGER);
    VAR i,j : INTEGER;
        x,w : keyinfo;
    BEGIN
      i := 1;
      j := r;
      x := keys[(l+r) DIV 2];
      REPEAT
        WHILE keys[i].hashlets   x.hashlets DO i := i+1;
        WHILE x.hashlets   keys[j].hashlets DO j := j-1;
        IF i  =j
          THEN BEGIN
                 w := keys[i];
                 keys[i] := keys[j];
                 keys[j] := w;
                 i := i+1;
                 j := j-1
               END
      UNTIL i   j;
      IF l   j THEN sorta(l,j);
      IF i   r THEN sorta(i,r)
    END;  (* sorta *)

  BEGIN  (* sortalpha *)
    low := list.first;
    high := list.last;
    sorta(low,high);
    index := low;
    WHILE index   high DO
      BEGIN
        keys[index].link := index+1;
        index := index+1
      END  (* while *)
    keys[high].link := MAXINT;
    list.first := low;
    list.last  := high;
    sortalpha := list
  END;  (* sortalpha *)
```

```
(*-------------------------------------*)
(* Sort selected letters for one word *)
(*-------------------------------------*)

FUNCTION bubblesort (hlets : aword; num : INTEGER) : aword;
  VAR posn : INTEGER;
    temp : CHAR;
    unsorted : BOOLEAN;
  BEGIN
    unsorted := TRUE;
    WHILE unsorted DO
      BEGIN
        unsorted := FALSE;
        posn := 1;
        WHILE posn   num DO
          BEGIN
            IF hlets[posn]   hlets[posn+1]
              THEN  (* switch *)
                BEGIN
                  temp := hlets[posn];
                  hlets[posn] := hlets[posn+1];
                  hlets[posn+1] := temp;
                  unsorted := TRUE
                END
              posn := posn+1
          END  (* while *)
      END  (* while *)
    bubblesort := hlets
  END;  (* bubblesort *)
```

```
  (*-------------------------------------------------------------*)
  (* Extracts chosen letters for this subset, then sorts the     *)
  (* subset lexicographically on chosen letters; returns TRUE    *)
  (* if no neigboring keys have the same set of chosen letters   *)
  (*-------------------------------------------------------------*)

FUNCTION check (len : INTEGER; vector : boolposn) : BOOLEAN
  VAR i,j,ptr,next,numpos : INTEGER;
      nomatch : BOOLEAN;
  BEGIN
    WITH subset[len] DO
      BEGIN
        Chosen := vector;
        (* extract hashlets *)
        ptr := queue.first;
        WHILE ptr     MAXINT DO
          WITH keys[ptr] DO
          BEGIN
            j := 0;
            hashlets := allblank;  (* clean slate to start *)
            FOR i := TO len DO
              IF vector[i]
                THEN (* add letter to hashlets *)
                  BEGIN
                    j := j+1;
                    hashlets[j] := word[i]
                  END
            numpos := j; (* number of positions used *)
            (* sort selected letters *)
            hashlets := bubblesort(hashlets,numpos);
            (* go on to next key *)
            ptr := link
          END  (* while *)
        (* sort queue on hashlets *)
        queue := sortalpha(queue);
        (* make one pass comparing neighbors for conflicts *)
        ptr := queue.first;
        next := keys[ptr].link;
        nomatch := TRUE:
        WHILE nomatch AND next    MAXINT DO
          IF keys[ptr].hashlets = keys[next].hashlets
            THEN nomatch :=  FALSE
            ELSE BEGIN
                   ptr := next;
                   next := keys[ptr].link
                 END  (* while *)
        check := nomatch
      END  (* with *)
END;  (* check *)
```

```
(*-------------------------------------------*)
(*     A method for finding independent      *)
(*  letter positions which ensures finding the *)
(*  combination with the fewest chosen positions *)
(*-------------------------------------------*)

FUNCTION findcombo (len : INTEGER) : boolposn
  TYPE rootptr = @rootnode;
       rootnode = RECORD
                    BITS : boolposn;
                    minpower : INTEGER;
                    next : rootptr
                  END;
       listrec = RECORD head,last : rootptr END;
  VAR rootlist : listrec;
      arrow : rootptr;

  (*=============================================================*)

    (* A function which returns *)
    (*  2 to the x power in     *)
    (*    binary, reversed.     *)

    FUNCTION binpower (x : INTEGER) : boolposn;
      BEGIN
        binpower := zbool;
        binpower(x+1) := TRUE
      END;  (* binpower *)

  (*=============================================================*)

    (* A function which returns the boolean sum of two vectors *)

    FUNCTION boolsum (term1,term2 : boolposn) : boolposn;
      VAR i : INTEGER;
      BEGIN
        FOR i := TO maxkeylength DO
          boolsum[i] := term1[i] OR term2[i]
      END;  (* boolsum *)

  (*=============================================================*)
```

```
(*================================================================*)

(* ---------------------------------------------------------------*)
(* A function which tries each descendent of the nodes on         *)
(* roots, the list of candidates, until a solution is             *)
(* found or no more candidates exist.                             *)
(* ---------------------------------------------------------------*)

FUNCTION testlist (lenth : INTEGER; roots : listrec) : boolposn;
  VAR i : INTEGER;
    trial : boolposn;
    found : BOOLEAN;

   (*================================================================*)

    (*---------------------------------------------------------- *)
    (* A procedure which adds a new candidate to the end         *)
    (* of the queue of candidate combinations.                   *)
    (*---------------------------------------------------------- *)

   PROCEDURE addroot (combo : boolposn; x : INTEGER);
     VAR rptr : rootptr;
     BEGIN
       NEW(rptr);
       WITH rptr@ DO
         BEGIN
           bits := combo;
           minpower := x;
           next := NIL
         END  (* with *)
       roots.last@.next := rptr;
       roots.last := rptr
     END;  (* addroot *)

    (================================================================ *)
```

```
(*=============================================================================*)

BEGIN  (* testlist *)
  found := FALSE;
  WHILE NOT found and roots.head    NIL DO
    BEGIN
      (* expand node at head of queue *)
      WITH roots.head@ DO
        BEGIN
          i := 0;
          WHILE NOT found AND i   minpower DO
            BEGIN
              trial := boolsum(binpower(i),bits);
              found := check(lenth,trial);
              IF found
                THEN testlist := trial
                ELSE IF i   0 THEN addroot(trial,i);
              i := i+1
            END  (* while *)
        END  (* with *)
      roots.head := roots.head@.next
    END  (* while *)
  IF NOT found THEN testlist := zbool
END;  (* testlist *)

(*=============================================================================*)

BEGIN  (* findcombo *)
  (* create initial root of tree *)
  NEW(arrow);
  rootlist.head := arrow;
  rootlist.last := arrow;
  WITH arrow@ DO
    BEGIN
      bits := zbool;
      minpower := len;
      next := NIL
    END  (* with *)
  findcombo := testlist(len,rootlist)
END;  (* findcombo *)
```

```
    (*-------------------------------------------------------------*)
    (* Count letter frequencies in chosen positions for this *)
    (*                        subset.                              *)
    (*-------------------------------------------------------------*)

  PROCEDURE countfreq (lenth : INTEGER);
    VAR index,posn : INTEGER;
    BEGIN
      freq : = zfreq;  (* zero frequency array *)
      WITH subset(lenth) DO
        BEGIN
          index := queue.first;
          REPEAT
            WITH keys[index] DO
              FOR posn := 1 TO lenth DO
                IF chosen[posn]
                  THEN freq[chartonum[word[posn]]] :=
                         freq[chartonum[word[posn]]] + 1;
            index := keys[index].link
          UNTIL index = MAXINT
        END  (* with *)
    END;  (* countfreq *)

  (*=============================================================================*)

    (*-----------------------------------*)
    (* Sort keys on sum of frequencies. *)
    (*-----------------------------------*)

  PROCEDURE firstorder (len : INTEGER);
    VAR index,posn,sym : INTEGER;
    BEGIN
      WITH subset[len] DO
        BEGIN (* calculate product of frequencies for each key *)
          FOR index := lowest TO highest DO
            WITH keys[index] DO
              BEGIN
                sortval := 1;
                posn := 1;
                REPEAT
                  sym := chartonum[hashlets[posn]];
                  sortval := sortval * freq[sym];
                  posn := posn+1
                UNTIL hashlets[posn] = blank;
                (* want decreasing order *)
                sortval := -(sortval)
              END  (* for *)
          quicksort(lowest,highest)
        END  (* with *)
    END;  (* firstorder *)
```

```
(*-------------------------------------------- *)
(* A function which sorts letters in an array  *)
(*  into order of increasing frequency.        *)
(*--------------------------------------------*)

FUNCTION freqsort (letts : aword; numlets : INTEGER) : aword;
  VAR posn : INTEGER;
      temp:  CHAR;
      unsorted : BOOLEAN;
  BEGIN
    unsorted := TRUE;
    WHILE unsorted DO
      BEGIN
        unsorted := FALSE;
        posn := 1;
        WHILE posn   numlets DO
          BEGIN
            IF freq[chartonum[letts[posn]]]
                freq[chartonum[letts[posn+1]]]
              THEN BEGIN  (* switch *)
                     temp := letts[posn];
                     letts[posn] := letts[posn+1];
                     letts[posn+1] := temp;
                     unsorted := TRUE
                   END  (* then *)
            posn := posn+1
          END  (* while *)
      END  (* while *)
    freqsort := letts
  END;  (* freqsort *)
```

```
(*-------------------------------------------------------------------*)
(* This procedure receives the length of the subset which is to      *)
(* be re-ordered.  The keys are in order by decreasing sortval at    *)
(* this point.  Here we see that keys whose hash values are deter-   *)
(* mined by addition of some other key get inserted in the order     *)
(* as soon as all their letters are determined.  We will finish      *)
(* with the keys in order in the array KEYS.  This includes the      *)
(* Slingerland and Waugh ordering.  This procedure calls the sub-    *)
(* routines DETFINDER, MAXDETCARD, ADDTOORDER, DELREMKEYS,           *)
(*                          and DELCANDLIST.                         *)
(*-------------------------------------------------------------------*)

PROCEDURE reorder (len : INTEGER);
  TYPE knodeptr = @knode;
       knode = RECORD
                  kinfo : keyinfo;  (* key and assoc. information *)
                  nextnode : knodeptr; (* link to next key *)
                  prevnode : knodeptr (* link to preceding key *)
                END;
       detptr = @detnode;
       detnode = RECORD
                   keyaddr : knodeptr;
                   nextdet  : detptr
                 END;
       candptr = @candnode;
       candnode = RECORD
                    kaddr : knodeptr;
                    prevcand,
                    nextcand : candptr;
                    card : INTEGER;
                    detlist : detptr
                  END;
  VAR index,sval,mark : INTEGER;
      candlist,cptr,prevc : candptr;
      remkeys,rptr,prevr : knodeptr;
      ordlist,ordptr,ordend : knodeptr;
```

```
(*=================================================================*)

  (*-----------------------------------------------------*)
  (* A function which takes as parameters a candidate    *)
  (* record and a current copy of USED and finds and     *)
  (* counts keys whose values are determined if this key *)
  (* is chosen next for inclusion in the solution set.   *)
  (* It returns the updated candidate record.            *)
  (* ----------------------------------------------------*)

FUNCTION detfinder (candrec : candptr; letsused : boolarray)
  VAR index,ch : INTEGER;
      dtptr,listptr,endptr : detptr;
      remptr : knodeptr;

  (*=================================================================*)

    (*-------------------------------------------------------------*)
    (* A Boolean function which returns TRUE when all hashlets are *)
    (* marked USED. Returns FALSE when first unused letter is found*)
    (*-------------------------------------------------------------*)

FUNCTION allused (VAR hletters : aword) : BOOLEAN;
  VAR i : INTEGER;
      ok : BOOLEAN;
  BEGIN
    i := 1;
    ok := TRUE;
    REPEAT
      IF letsused(chartonum(hletters[i]))
        THEN ok := TRUE
        ELSE ok := FALSE;
      i := i+1
    UNTIL (NOT ok) OR (i    maxlen) OR (hletters[i]= blank);
    allused := ok
  END;  (* allused *)

  (*=================================================================*)
```

```
(*=============================================================================*)

 BEGIN (* detfinder *)
   WITH candrec@ DO
     BEGIN
       listptr := NIL;
     (* update used letters to include those in candidate *)
     index := 1;
     WITH kaddr@.kinfo DO
       REPEAT
         ch := chartonum[hashlets[index]];
         letsused(ch) := TRUE;
         index := index +1
       UNTIL index    maxlen OR hashlets[index] = blank;
     (* scan REMKEYS for determined hash values *)
     remptr := remkeys;
     WHILE remptr     NIL DO
       IF remptr    kaddr AND allused(remptr@.kinfo.hashlets)
         THEN BEGIN  (* found one! *)
                card := card+1;
                NEW(dtptr);
                IF listptr = NIL
                  THEN listptr := dtptr
                  ELSE endptr@.nextdet := dtptr;
                endptr := dtptr;
                dtptr@.keyaddr := remptr;
                dtptr@.nextdet := NIL;
                remptr := remptr@.nextnode
              END  (* then *)
         ELSE remptr := remptr@.nextnode
    END  (* with *)
    detfinder := listptr
 END;  (* detfinder *)

(*=============================================================================*)
```

```
(*=========================================================================*)

 (*-------------------------------------------------------------*)
 (* A function which returns a pointer to the candidate       *)
 (* which will bring in with it the largest number of words   *)
 (*-------------------------------------------------------------*)

FUNCTION maxdetcard (cand : candptr) : candptr;
  VAR max : INTEGER;
      select : candptr;
  BEGIN
    max := 0;
    select := cand;
    WHILE cand   NIL DO
      BEGIN
        IF cand@.card   max
          THEN BEGIN
                 select := cand;
                 max := select@.card
               END
        cand := cand@.nextcand
      END  (* while *)
    maxdetcard := select
  END;  (* maxdetcard *)

(*=========================================================================*)

  (*-----------------------------------------*)
  (* Add the contents of kptr@ to ordlist.   *)
  (*-----------------------------------------*)

PROCEDURE add (kptr : knodeptr);
  (* global : ordptr,ordend,ordlist *)
  BEGIN
    NEW(ordptr);
    IF ordlist = NIL
      THEN ordlist := ordptr
      ELSE ordend@.nextnode := ordptr;
    WITH ordptr@ DO
      BEGIN
        nextnode := NIL;
        prevnode := ordend;
        kinfo := kptr@.kinfo
      END
    ordend := ordptr
  END;  (* add *)
```

```
(*================================================================*)

  (*----------------------------------------------------------------*)
  (* A procedure to delete a key from the list of remaining keys *)
  (*----------------------------------------------------------------*)

PROCEDURE delremkeys (rmptr : knodeptr);
  BEGIN
    WITH rmptr@ DO
      BEGIN
        IF prevnode = NIL
          THEN remkeys := nextnode
          ELSE prevnode@.nextnode := nextnode;
        IF nextnode  NIL
          THEN nextnode@.prevnode := prevnode
      END  (* with *)
  END; (* delremkeys *)

(*================================================================*)

(*----------------------------------------------------------*)
(* A procedure to delete a node from the list of candidates *)
(*----------------------------------------------------------*)

PROCEDURE delcandlist (cdptr : candptr);
  BEGIN
    WITH cdptr@ DO
      BEGIN
        IF prevcand = NIL
          THEN candlist := nextcand
          ELSE prevcand@.nextcand := nextcand;
        IF nextcand    NIL
          THEN nextcand@.prevcand := prevcand
      END  (* with *)
  END;  (* delcandlist *)

(*==========================================================*)
```

```
(*================================================================*)

(*----------------------------------------------------------------*)
(* Adds a candidate and its determined keys to the ordering.      *)
(* deleting each key's occurrence in various lists.  Also         *)
(* ~ updates the set of used letters.                             *)
(*----------------------------------------------------------------*)

PROCEDURE addtoorder (cand : candptr);
  VAR index,ch : INTEGER;
      dptr : detptr;
      cptr,prevc : candptr;
  BEGIN
    (* mark numun to show this is a candidate *)
    cand@.kaddr@.kinfo.numun := 1;
    (* place candidate in ordering *)
    add(cand@.kaddr);
    (* place detlist keys in ordering *)
    dptr := cand@.detlist;
    WHILE dptr    NIL DO
      BEGIN
        (* mark numun = 0 to show no new letters *)
        dptr@.keyaddr@.kinfo.numun := 0;
        add(dptr@.keyaddr);
        dptr := dptr@.nextdet
      END  (* while *)
    (* update used *)
    index := 1;
    WITH cand@.kaddr@.kinfo DO
      REPEAT
        ch := chartonum[hashlets[index]];
        used(ch) := TRUE;
        index := index+1
      UNTIL index    maxlen OR hashlets[index] = blank;
```

```
  (* remove candidate from remkeys *)
  delremkeys(cand@.kaddr);
  (* remove keys in detlist from *)
  (* both remkeys and candlist  *)
  dptr := cand@.detlist;
  WHILE dptr   NIL DO
    BEGIN
      delremkeys(dptr@.keyaddr);
      cptr := candlist;
      prevc := NIL
      (* scan candlist for match *)
      WHILE cptr    NIL DO
        IF cptr@.kaddr = dptr@.keyaddr
          THEN BEGIN
                 delcandlist(cptr);
                 cptr := NIL
               END
          ELSE BEGIN
                 prevc := cptr;
                 cptr := cptr@.nextcand
               END
        dptr := dptr@.nextdet
      END  (* while *)
    (* remove candidate from candlist *)
    delcandlist(cand)
  END;  (* addtoorder *)

(*=============================================*)
```

```
(*=====================================================*)

BEGIN  (* reorder *)
  WRITELN; WRITELN('reordering subset ',len:2);
  used := zused;  (* no letters used yet *)
  (* create linked list of remaining keys *)
  WITH subset[len] DO
    BEGIN
      index := lowest;
      NEW(remkeys);
      prevr := NIL;
      WITH remkeys@ DO
        BEGIN
          prevnode := prevr;
          nextnode := NIL;
          kinfo := keys[index]
        END  (* with *)
      rptr := remkeys;
      (* add the rest of the keys to the list *)
      WHILE index   highest DO
        BEGIN
          index := index+1;
          prevr := rptr;
          NEW(rptr);
          prevr@.nextnode := rptr;
          WITH rptr@ DO
            BEGIN
              prevnode := prevr;
              nextnode := NIL;
              kinfo := keys[index]
            END  (* with *)
        END  (* while *)
    END  (* with *)
    (* start the reordering process *)
    ordlist := NIL;
    WHILE remkeys   NIL DO
      BEGIN
        (* create a list of the next candidates for *)
        (* inclusion in the ordering (all those    *)
        (* which have the largest sortvalue).      *)
        (* add the first key *)
        NEW(candlist);
        WITH candlist@ DO
          BEGIN
            kaddr := remkeys;
            prevcand := NIL;
            nextcand := NIL;
            detlist := NIL;
            card := 0
          END  (* with *)
        sval := remkeys@.kinfo.sortval;
        rptr := remkeys@.nextnode;
        prevr := remkeys;
        prevc := candlist;
```

```
        (* now add all keys with the same sortval as the *)
        (* first candidate to the candlist.               *)
        WHILE rptr   NIL AND sval = rptr@.kinfo.sortval DO
          BEGIN
            NEW(cptr);
            WITH cptr@ DO
              BEGIN
                kaddr := rptr;
                prevcand := prevc;
                nextcand := NIL;
                detlist := NIL;
                card := 0
              END  (* with *)
            prevc@.nextcand := cptr;
            prevc := cptr;
            rptr := rptr@.nextnode
          END  (* while *)
        (* process each member of candlist to *)
        (* find those keys each determines.   *)
        cptr := candlist;
        WHILE candlist   NIL DO
          BEGIN
            cptr := candlist;
            REPEAT
              cptr@.detlist := detfinder(cptr,used);
              cptr := cptr@.nextcand
            UNTIL cptr = NIL;
              (* choose candidate with largest detlist *)
            cptr := maxdetcard(candlist);
              (* add candidate and dependents to ordering *)
              (* and remove them from other lists.        *)
            addtoorder(cptr)
          END  (* while candlist *)
      END  (* while remkeys *)
      (* all keys are now in ordlist *)
      (* put them into array 'keys'  *)
      mark := subset[len].lowest - 1;
      REPEAT
        mark := mark+1;
        keys(mark) := ordlist@.kinfo;
        ordlist := ordlist@.nextnode
      UNTIL ordlist = NIL OR mark = subset[len].highest
END:  (* reorder *)
```

```
  (*-------------------------------------------------------*)
  (* This function deletes those letters in the array which *)
  (* already have the condition that USED(letter) = TRUE.  *)
  (*-------------------------------------------------------*)

FUNCTION delused (hletters : aword) : aword;
  VAR i,j,k : INTEGER;
      present : BOOLEAN;
      newlets : sword;
  BEGIN
    newlets := allblank;  (* clean slate *)
    i := 1;
    j := 1;
    REPEAT
      k := 1;
      present := FALSE;
      WHILE k   j AND NOT present DO
        IF hletters[i] = newlets [k]
          THEN present := TRUE
          ELSE k := k+1;
      IF NOT present AND NOT used(chartonum(hletters[i]))
        THEN BEGIN
               newlets[j] := hletters[i];
               j := j+1
             END
      I := i+1
    UNTIL 1   maxlen OR hletters[i] = blank;
    delused := newlets
  END;  (* delused *)

(*=========================================================== *)

  (*-----------------------------------------*)
  (* This function returns the index of the *)
  (* next open hash table location after x. *)
  (*-----------------------------------------*)

FUNCTION nextopen (x : INTEGER) : INTEGER;
  BEGIN
    While x   maxtableloc AND taken[x] DO
      x := x+1;
    IF taken [x] THEN nextopen := MAXINT
                 ELSE nextopen := x
  END;  (* nextopen *)
```

```
(*-------------------------------------------------------------*)
(* For each key in the subset which introduced a new letter, *)
(* isolate and count the new letters.  If numun = 0 then     *)
(* there are no new letters;  assign allblanks to unused.    *)
(*-------------------------------------------------------------*)

PROCEDURE setunused (len : INTEGER);
  VAR i,j : INTEGER;
  BEGIN
    used := zused;
    WITH subset[len] DO
      FOR i := lowest TO highest DO
        WITH keys[i] DO
          IF numun = 0
            THEN unused := allblank
            ELSE BEGIN
                   (* isolate new letters *)
                   unused := delused(hashlets);
                   (* count new letters *)
                   numun := findlen(unused);
                   (* mark new letters used *)
                   FOR j := 1 TO numun DO
                     used[chartonum[unused[j]]] := TRUE;
                   (* sort letters by decreasing frequency *)
                   unused := freqsort(unused,numun)
                 END  (* with *)
  END;  (* setunused *)

(*=============================================================*)

  (* returns a copy of lets with symbol deleted *)

FUNCTION delsym (symbol : INTEGER; VAR lets : aword) : aword;
  VAR newlets : aword;
      ch   : CHAR;
      i,j  : INTEGER;
  BEGIN
    ch := numtochar[symbol];
    i  := 1;
    j  := 1;
    newlets := allblank;
    WHILE lets[i]    blank DO
      BEGIN
        IF lets[i]    ch
          THEN BEGIN
                 newlets[j] := lets[i];
                 j := j+1
               END  (* if *)
        i := i+1
      END  (* while *)
    delsym := newlets
  END;  (* delsym *)
```

```
(*-------------------------------------------------------------------*)
(*   Addword is the procedure which varies the associated values     *)
(* for untried letters. For each possible assignment of values,      *)
(* the procedure TRY is called.  TRY attempts to place the current   *)
(* key in the hash table, given the current offset and associated    *)
(* letter values.  If the key is successfully placed, then TRY       *)
(* calls ADDWORD.  Actually, ADDWORD calls VARY, which calls TRY.    *)
(*-------------------------------------------------------------------*)

PROCEDURE addword (wcount : INTEGER);

  (*=================================================================*)

  (*--------------------------------------*)
  (*  TRY calls ADDWORD if the current    *)
  (* key creates no hash value conflict.  *)
  (*--------------------------------------*)

 PROCEDURE try (wdcount : INTEGER);
   VAR hsh,i,sym : INTEGER;
   BEGIN
     trycount := trycount+1;
     WITH keys[wdcount],subset[length] DO
       BEGIN (* calculate key's  hash value *)
         hsh := offset;
         i := 1;
         WHILE i  = maxlen AND hashlets[i]    blank DO
           BEGIN
             sym := chartonum[hashlets[i]];
             hsh := hsh + assoc[Sym];
             i := i+1
           END  (* while *)
         IF NOT taken[hsh] THEN
           BEGIN  (* add key to hash table *)
             taken[hsh] := TRUE;
             IF hsh = firstopen
               THEN firstopen := nextopen(firstopen);
             sortval := hsh;
             IF wdcount = highest
               THEN done := TRUE
               ELSE addword(wdcount+1);
             IF NOT done THEN
               BEGIN  (* backtrack *)
                 taken[hsh] := FALSE;
                 firstopen := MIN(firstopen,hsh)
               END  (* if *)
           END  (* if *)
       END  (* with *)
   END;  (* try *)

(*===================================================================*)
```

```
(*=============================================================*)

(*-------------------------------------------------------------*)
(* VARY modifies the associated value in position COUNT.      *)
(* If there are no letters left, then TRY is called.          *)
(* Otherwise, VARY is called (recursively) to deal with       *)
(* the next unused letter.                                    *)
(*-------------------------------------------------------------*)

PROCEDURE vary (posn,wrdcount : INTEGER);
  VAR sym,subtotal,i,x : INTEGER;
      sublets : aword;
  BEGIN
    varycount := varycount+1;
    WITH keys[wrdcount],subset[length] DO
      BEGIN
        sym := chartonum[unused[posn]];
        IF posn = 1
          THEN (* adjust initial value *)
            BEGIN
              sublets := delsym(sym,hashlets);
              subtotal := offset;
              i := 1;
              WHILE sublets [i]    blank DO
                BEGIN
                  x := chartonum[sublets[i]];
                  subtotal := subtotal + assoc[x];
                  i := i+1
                END  (* while *)
              IF subtotal    firstopen
                THEN assoc[sym] := (firstopen - subtotal) - 1
                ELSE assoc[sym] := -1
            END  (* then *)
          ELSE assoc[sym] := -1;
        WHILE NOT done AND assoc[sym]  maxcharval D)
          BEGIN
            assoc[sym] := assoc[sym] + 1;
            IF posn = 1
              THEN try(wrdcount)
              ELSE vary(posn-1,wrdcount)
          END  (* while *)
      END  (* with *)
  END;  (* vary *)

(*=============================================================*)

BEGIN (* addword *)
  addcount := addcount+1;
  WITH keys[wcount] DO
    IF numun = 0
      THEN try(wcount)
      ELSE vary(numun,wcount)
END;  (* addword *)
```

```
  (*-------------------------------------------------------------*)
  (* This procedure calls the others which carry out the trial *)
  (* integer assignments to letters until a solution is found.  *)
  (*-------------------------------------------------------------*)

PROCEDURE assignvalues (len : INTEGER);
  BEGIN
    linecount := 0;
    WITH subset[len] DO
      BEGIN
        maxcharval := highest-lowest+1;
        WRITELN('AFTER REORDER');
        wordcount := lowest;
        WRITE('start with ',wordcount:3,'th item in keys,);
        WRITELN(keys[wordcount].word:len);
        done := FALSE;
        addcount := 0;
        trycount := 0;
        varycount := 0;
        assoc := zassoc;
        date(5,0,pdate);
        time(4,0,ptime);
        WRITELN('Beginning search at ',ptime,' on ',pdate);
        cpu(0,0,msecs);
        addword(wordcount);  (* plunge into black hole! *)
        cpu(1,0,msecs);
        time(4,0,ptime);
        WRITELN(' Search completed at ',ptime,'.');
        WRITELN(msecs,' milliseconds of elapsed cpu time.');
        totmsecs := totmsecs + msecs;
        IF NOT done THEN WRITELN('No solution found!!!!!!');
        WRITELN('Addword called ',addcount:1,' times.  Try called ',
                  trycount:1,' times.');
        WRITELN('Vary called ',varycount:1,' times.')
      END  (* with *)
  END;  (* assignvalues *)
```

```
  (*-------------------------------------------------------------*)
  (* This procedure adjusts the offset values for the subset   *)
  (* whose associated values have just been calculated and     *)
  (* then sets offset for the next existent subset so that its *)
  (* first element will follow the last of the current set.    *)
  (*-------------------------------------------------------------*)

PROCEDURE adjustnextoffset (len : INTEGER);
  VAR 1 : INTEGER;
  BEGIN
    (* sort current subset *)
    WITH subset[len] DO quicksort(lowest,highest);
    (* find next subset *)
    WHILE len   maxlen DO
      IF subset[len+1].exists
        THEN WITH subset[len+1] DO
          BEGIN
            WHILE taken[offset] DO
              offset := offset+1;
            firstopen := offset;
            len := maxlen
          END
        ELSE len := len+1
  END;  (* adjustnextoffset *)

(*=============================================================*)

PROCEDURE letorder (len : INTEGER);
  VAR i,j,count : INTEGER;
  BEGIN
    WRITELN;
    WRITELN('   Letter order for subset ',len:1);
    WRITELN;
    WRITE('  ');
    count := 0;
    WITH subset[len] DO
      FOR i := lowest TO highest DO
        WITH keys[i] DO
          IF numun    0
            THEN BEGIN
                   count := count + numun;
                   FOR j := numun DOWNTO 1 DO
                     WRITE(unused[j],' ')
                 END
    WRITELN('  ',count:1,' letters');
    WRITELN
  END;  (* letorder *)
```

```
BEGIN  (* main program *)
  initio;
  initialise;
  getkeys;
  quicksort(1,numkeys);     (* increasing by length *)
  partition;    (* create subsets by length *)
  FOR i := TO maxkeylength DO
    IF subset[i].exists
      THEN BEGIN
             subset[i].chosen := findcombo(i);
             IF allfalse(subset[i].chosen)
               THEN BEGIN
                      WRITELN;
                      WRITE('  The keys in subset ',i:1);
                      WRITE('  cannot be distinguished');
                      WRITELN(' by our method.');
                      WRITELN('  BAIL OUT ');
                      HALT
                    END;  (* if *)
             countfreq(i);
             firstorder(i);
             reorder(i);
             setunused(i);
             letorder(i);
             assignvalues(i);
             adjustnextoffset(i);
             princhospos(subset[i].chosen);
             printkeys(i);
             showletvalues(i);
             WRITELN('TOTAL SEARCH TIME: ',totmsecs);
           END  (* then *)
END.  (* of the whole thing *)
```

```
     ∇HASH;AI;ANS;LCOMP;LENGTH;NSYMBOLS;ROWS;F;FTOL;L;EX;S;ONE;STCK;C
[1]  ⍝ HASH IS THE MAIN PROCEDURE THAT CALLS ALL THE OTHER FUNCTIONS.
[2]  ⍝
[3]  ⍝ SET UP THE CONSTANTS FOR USE IN THE FUNCTIONS.
[4]   ALPH←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'' '
[5]  START:OP∆ 'WORDS TO BE HASHED'
[6]   →START×⍳0=1↑⍴DATA←IN ⍞
[7]   AI←⎕AI[2]
[8]  RTE:OP∆ ' LETTERS TO BE USED. I.E. 1 2 FOR FIRST AND SECOND LETTERS'
[9]   LENGTH←,1↓⍴DATA
[10]  LETTER←,⍕OP∆ ' ',▼LENGTH⌊1⌈⌊,⎕
[11] TRB:⍞←EX←' IS BLANK TO BE A CHARACTER.  Y/N? :  '
[12]  →TRB×⍳3=ONE←'YN'⍳¯1↑OP∆ EX,¯1↑⍞
[13]  LENGTH←1⌈LENGTH×ONE-1
[14]  INFORM←LETTERS SET DATA
[15] TRL:⍞←EX←' IS LENGTH TO BE PART OF FUNCTION  Y/N? :  '
[16]  →TRL×⍳3=LCOMP←'YN'⍳¯1↑OP∆ EX,¯1↑⍞
[17]  →TCL×⍳2|LCOMP
[18]  INFORM[;3]←1
[19] TCL:INFORM←COR INFORM
[20]  →PSS×⍳(1↑⍴INFORM)=1↑⍴DATA
[21]  ⍞←EX←' WOULD YOU LIKE TO TRY A DIFFERENT ROUTE? :  '
[22]  →ROUTE×⍳'Y'=¯1↑OP∆ EX,¯1↑⍞
[23] PSS:L←⍳0
[24]  F←⍳1↑⍴INFORM
[25]  NSYMBOLS←COUNT INFORM
[26] ⍝ NSYMBOLS IS A COUNT OF THE USE OF EACH SYMBOL.
[27] ⍝ WSYM INDICATES WHEN THE SYMBOL IS FIRST USED.
[28]  WSYM←(⍴NSYMBOLS)⍴0
[29] PM:⍞←EX←' ORDER BY PRODUCT OR MINIMUM  P/M?  :  '
[30]  →PM×⍳2=ONE←¯1+'PM'⍳¯1↑OP∆ EX,¯1↑⍞
[31] DROP:INFORM[F;]←INFORM[F;] VAL NSYMBOLS
[32]  →DOWN×⍳(⍴F)=+/~FTOL←0=INFORM[;1]
[33]  L←(EX←(~EX∈L)/EX←FTOL/⍳⍴FTOL),L
[34]  NSYMBOLS←NSYMBOLS-COUNT INFORM[EX;]
[35]  →DROP×⍳0<⍴F←(~FTOL)/⍳⍴FTOL
[36] ⍝ ALL WORDS ARE FILLERS
[37]  INFORM←INFORM[L;]
[38]  CHECK←STCK←(0 ¯3 +⍴INFORM)⍴0
[39]  C←0
[40]  →DOWN2
[41] DOWN:INFORM←INFORM[F,L;]
[42] ⍝ SORT THE NONE FILLERS BY VALUE
[43]  INFORM[⍳⍴F;]←INFORM[▼INFORM[⍳⍴F;1];]
[44] ⍝ PUT THE NON-FILLERS IN PREORDER FORM
[45]  INFORM[⍳⍴F;]←INFORM[PRED INFORM;]
[46] DOWN2:
[47] ⍝ FINISH MAKING THE STCK FOR THE FILLERS
[48]  PUSH INFORM
[49]  OP∆ ' CPU SECONDS USED IN HASH IS ',▼0.001×⎕AI[2]-AI
[50]  OP∆ ' THE DATA IN CORRECTED PREORDER FORM'
[51]  OP∆ '  '
[52]  OP∆ PF DATA[INFORM[;2];]
```

```
     ∇AY←N SET AR
[1]  ⍝ CREATES AN ARRAY OF INFORMATION ABOUT THE DATA
[2]  ⍝    VALUE    INDEX    LENGTH    LETTERS TO BE USED
[3]  ⍝      0        1        3         3        20
[4]   AY←ALPH⍳((AR+.='  ')⌽AR,AR[;LENGTH⍴LENGTH])[;N]
[5]  ⍝ CHANGES '   CAT' TO '   CATTTTTTT' TO 'CATTTTTTT   '
[6]  ⍝ AND PULLS OFF THE NEEDED LETTERS.
[7]   AY←0,(⍳1↑⍴AR),(AR+.≠'  '),AY
```

```
     ∇ARRAY←COR AR;J;VEC
[1]  ⍝ IF ANY 2 WORDS IN THE LIST WILL HAVE AN UNAVOIDABLE COLLISION
[2]  ⍝ ONE OF THE WORDS WILL BE REMOVED FROM THE LIST.
[3]  ⍝
[4]  ⍝ ORDER THE WORDS BY LENGTH AND THEN ALPHABETICALLY
[5]  ⍝ LENGTH IS ONE IF NOT IMPORTANT.
[6]   ARRAY←AR←AR[⍋(⍴ALPH)⊥⍉1+ 0 2 ↓AR;]
[7]  ⍝ COMPARE EACH WORD TO THE NEXT ONE
[8]   →0×⍳~∨/VEC←∧/J=1⊖J← 0 2 ↓AR
[9]  ⍝ REMOVE CONFLICTING WORDS
[10]  J←DATA[VEC/1⊖AR[;2];]
[11]  ⎕←DATA[VEC/AR[;2];],(((+/VEC),16)⍴' CONFLICTS WITH '),J
[12]  ARRAY←(~VEC)⌿AR
```

```
     ∇ARRAY←COUNT INFORM;I;VALUE
[1]  ⍝ COUNTS THE NUMBER OF OCCURENCES OF EACH
[2]  ⍝ LETTER IN EACH POSITION.
[3]   ARRAY←((I←⍴ALPH),⍴LETTERS)⍴0
[4]   VALUE← 0 3 ↓INFORM
[5]  GO:ARRAY[I;]←I+.=VALUE
[6]   →(0≠I←I-1)/GO
```

```
     ∇AR←ARRAY VAL CNT;L;I
[1]  ⍝GIVES A VALUE TO EACH WORD WHICH IS THE PRODUCT OF ONE LESS
[2]  ⍝ THEN THE NUMBER OF TIMES EACH LETTER IN THE WORD IS USED.
[3]   AR←ARRAY
[4]   CNT←CNT-1
[5]  ⍝ IF ONE OF THE LETTERS IN THE WORD IS ONLY USED ONCE
[6]  ⍝ THEN A ZERO RESULTS INDICATING A FILLER. A WORD THAT CAN
[7]  ⍝ BE PLACED ANYWHERE WITHOUT AFFECTING OTHER WORDS.
[8]   S←⍴LETTERS
[9]   AR[;1]←1+ONE×1↑⍴INFORM
[10]  →M×⍳ONE
[11] P:AR[;1]←AR[;1]×,CNT[AR[;3+S];S]
[12]  →P×⍳0<S←S-1
[13]  →0
[14] M:AR[;1]←AR[;1]⌊,CNT[AR[;3+S];S]
[15]  →M×⍳0<S←S-1
```

```
     ∇PORD←PRED INFORM;I;J;VEC;EX;S;GROUP;P;LINE
[1]  ⍝ PRED SET INFORM IN AN ORDER THAT AS FEW NEW
[2]  ⍝ LETTERS ARE USED IN EACH SUCCESSIVE WORD
[3]   INFORM←⍉3↑⍉INFORM
[4]  ⍝ P  AND  LINE  ARE CONSTANTS
[5]   P←⍴LETTERS
[6]   LINE←⍳1↑⍴INFORM
[7]  ⍝ STCK KEEPS TRACK OF WHEN THE LETTERS IN THE
[8]  ⍝ WORDS ARE FIRST USED
[9]  ⍝ EACH ROW OF CHECK HAS THE SAME ELEMENTS AS THE SAME
[10] ⍝ ROW IN STACK BUT IN DESCENDING ORDER.
[11]  EX←+/STCK←INFORM=(⍴INFORM)⍴INFORM[1;]
[12]  CHECK←STCK←STCK×(⍴INFORM)⍴⌽⍳P
[13] ⍝ EX PREVENTS THE SAME WORD FROM BEING USE TWICE.
[14] ⍝ THE INDICES TO FILLERS ARE FLAGED FOR NO USE
[15]  EX[(⍴F)+⍳⍴L]←-P
[16] ⍝ PORD IS THE ORDERING INDICES
[17]  PORD←(EX=P)/LINE
[18] ⍝ FLAG PORD OF EX
[19]  EX[PORD]←¯1
[20] ⍝ C IS THE NEW LETTER INDICE
[21]  S←C←P
[22] LOOP:WSYM[INFORM[1;S];S]←S
[23]  →LOOP×⍳0<S←S-1
[24] NEXT:→END×⍳(⍴PORD)=⍴F
[25] ⍝ FIND THE WORD CONTAINING THE LEAST NEW LETTERS
[26] ⍝ SET THE NEW LETTERS TO THE VALUE OF C
[27]  I←EX⍳J←⌈/EX
[28]  S←STCK[I;]⍳0
[29] NEXTS:EX←EX+VEC←INFORM[;S]=INFORM[I;S]
[30]  C←C+1
[31]  WSYM[INFORM[I;S];S]←C
[32]  STCK[;S]←STCK[;S]+C×VEC
[33]  →NEXTS×⍳P≥S←STCK[I;]⍳0
[34] ⍝ FIND THE NEW GROUP
[35]  GROUP←(P=EX)/LINE
[36] ⍝ ENSURE THE NEW GROUP WILL NOT BE USED AGAIN
[37]  EX[GROUP]←¯1
[38]  S←⍴GROUP
[39] ⍝ SORT EACH MEMBER OF THE GROUP
[40] NS:CHECK[GROUP[S];]←VEC[⍒VEC←,STCK[GROUP[S];]]
[41]  →NS×⍳0<S←S-1
[42] ⍝ ORDER THE GROUPS INDICES AND ADD TO PORD
[43]  PORD←PORD,GROUP[⍋C⊥⍉0 1⌽CHECK[GROUP;]]
[44] →NEXT
[45] END:STCK[⍳⍴PORD;]←STCK[PORD;]
[46]  CHECK[⍳⍴PORD;]←CHECK[PORD;]
```

```
     ∇PUSH INFORM;S;VEC;I;J
[1]  ⍝ PUSH FINISHES THE STCK FOR THE FILLERS.
[2]  ⍝ ALL GROUPS ARE SINGLE WORDS
[3]   →0×⍳0=⍴L
[4]   INFORM←0 3 ↓INFORM
[5]   J←(I←1↑⍴INFORM)-⍴L
[6]   L←J+⍳⍴L
[7]  NEXTJ:S←(,STCK[J←J+1;])⍳0
[8]  NEXTS:C←C+1
[9]   STCK[L;S]←STCK[L;S]+C×INFORM[L;S]=INFORM[J;S]
[10]  WSYM[INFORM[J;S];S]←C
[11]  →NEXTS×⍳(⍴LETTERS)≥S←(,STCK[J;])⍳0
[12]  CHECK[J;]←STCK[J;⍒,STCK[J;]]
[13]  →NEXTJ×⍳I>J


     ∇A←PF D;C;F;S;VEC
[1]  ⍝ PRINT FORMAT CREATES AN ARRAY WITH ONE SPACE BETWEEN
[2]  ⍝ EACH WORD AND NO WORDS SPLIT AT THE END OF THE LINE.
[3]   D←(~A←D∈' ')/D←,D,' '
[4]   A←+\1+(A≠0)/A←A-1+0,¯1↓A←A/⍳⍴A
[5]   S←1
[6]   F←⍳0
[7]  NEXT:S←S+⎕PW
[8]   F←F,(~VEC←S≤A)/A
[9]   F←(¯1↓F),(¯1↑F)+¯1+⍳C←S-¯1↑F
[10]  A←VEC/A+C-1
[11]  →NEXT×⍳0≠A
[12]  VEC←((⍴F)+⍴D)⍴1
[3]  VEC[F]←0
[14]  A←(((⍴VEC)≠⎕PW),¬PW)⍴VEC\D


     ∇AR←IN D;A
[1]  ⍝  IN TAKES VECTOR D WHICH CONTAINS A LIST OF WORDS AND
[2]  ⍝ PLACES THESE WORDS INTO A RIGHT JUSTIFIED ARRAY.
[3]  ⍝
[4]  ⍝ GET RID OF UNWANTED CHARACTERS BY CHANGING THEM TO BLANKS.
[5]   →PASS×⍳2≠⎕NC 'ALPH'
[6]   D←(ALPH,' ')[ALPH⍳D]
[7]  ⍝ 'A' IS A CHARACTERISTIC VECTOR WITH ONES INDICATING BLANKS.
[8]  ⍝ 'D' IS THE DATA WITH BLANKS REMOVED.
[9]  PASS:D←(~A←D∈' ')/D←,D,' '
[10] ⍝ 'A' BECOMES A LIST OF THE LENGTHS OF THE WORDS.
[11]  A←(A≠0)/A←A-1+0,¯1↓A←A/⍳⍴A
[12] ⍝ 'A' BECOMES AN ARRAY WITH '1'S TO THE RIGHT
[13]  A←A∘.≥⌽⍳1+⌈/0,A
[14] ⍝ WHICH IS USED TO IMAGE THE WORDS INTO THE ARRAY,
[15] ⍝ WITH A BLANK FOR EACH 0 AND A LETTER FOR EACH 1.
[16]  AR←(⍴A)⍴(,A)\D
```

```
     ∇LASH;BFUNCT
[1]  ⍝ THIS IS A NONE BACKTRACKING VERSION.
[2]   FUNCT←' LASH'
[3]  ⍝ CALL MAIN AND INDICATE NO BACKTRACKING.
[4]   BFUNCT←1
[5]   MAIN
[6]   PRINT
```

```
     ∇BASH;BFUNCT;LF;COL
[1]   ⍝ THIS FUNCTION BACKTRACKS WHEN THE SPECIFIED
[2]   ⍝ LOADING FACTOR IS NOT BEING OBTAINED.
[3]    FUNCT←' BASH'
[4]   LD:S←⍞←' LOADING FACTOR  0.5 TO 1 RANGE : '
[5]    →LD×⍳LF≠0.5⌈⌊1↑LF←,⎕
[6]    S←OPΔ S,▼LF
[7]    LF←⌈(1↑⍴INFORM)÷LF
[8]   ⍝
[9]   BT:S←⍞←' NUMBER OF ALLOWABLE BACKTRACKS : '
[10]   →BT×⍳1≠⍴COL←⌊,⎕
[11]   S←OPΔ S,▼COL
[12]  ⍝
[13]  ⍝ CALL MAIN INDICATING BACKTRACKING.
[14]   BFUNCT←0
[15]   MAIN
[16]   PRINT
```

```
     ∇MAIN;CC;C;M;CHAR;T;S;HSTCK;VALUES;GROUP;INCR
[1]  ⍝ THIS FUNCTION BACKTRACKS WHEN THE SPECIFIED
[2]  ⍝
[3]   TS1←⎕TS
[4]   AI←⎕AI[2]
[5]  ⍝
[6]   TABLE←(2×⌈/⍴DATA)⍴0
[7]   VL←((⍴ALPH),⍴LTTERS)⍴0
[8]   K←C←COLS←0
[9]  ⍝
[10]  CC←⌈/,WSYM
[11] ⍝
[12]  HSTCK←CHECK[;1]
[13]  INFORM[;1]←INFORM[;3]
[14] ⍝
[15]  SETINCR
[16] ⍝
[17] NEXTC:→STOP×⍳CC<C←C+1
[18]  K←K+1
[19]  INCR←,-/INFORM[NCI[NCS[C;1]+⍳NCS[C;2];];1]
[20]  INCR←(INCR≥0)/INCR
[21]  CHAR←NCS[C;3]
[22]  S←NCS[C;4]
[23]  →STEP×⍳0≠NCS[C;6]
[24]  M←⌈/0,INCR
[25]  M←¯1+((0,⍳M)∈INCR)⍳0
[26]  VL[CHAR;S]←VL[CHAR;S]+M
[27]  INFORM[;1]←INFORM[;1]+M×CHAR=INFORM[;3+S]
[28]  →NEXTC
[29] ⍝
[30] STEP:GROUP←NCS[C;5]+⍳NCS[C;6]
[31] ⍝
[32]  VALUES←INFORM[GROUP;1]
[33] ⍝ TERMINATE IF VALUES ARE EQUAL, BUG!
[34]  →STOP×⍳1≠⌈/,+/VALUES∘.=VALUES
[35] ⍝ LOOK FOR AN OPENING FOR THE SMALLEST VALUE.
[36]  M←⌊/VALUES
[37]  T←M-1
[38] NT:T←T+(T↓TABLE)⍳0
[39] ⍝ LOOP UNTIL THE INCREMENT ALLOWS ALL WORDS
[40] ⍝ IN THE GROUP TO FALL INTO FREE SPACES.
[41]  →NT×⍳0∨.≠TABLE[VALUES+T-M]
[42] ⍝ LOOP WHILE T-M IS A NONE ACCEPTABLE INCREMENT.
[43]  →NT×⍳(T-M)∈INCR
[44] ⍝ INCREMENT THE CHARACTER VALUE.
[45]  VL[CHAR;S]←VL[CHAR;S]+M←T-M
[46]  TABLE[VALUES+M]←GROUP
[47] ⍝ ADD M TO THE AFFECTED WORDS VALUES
[48]  INFORM[;1]←INFORM[;1]+M×CHAR=INFORM[;3+S]
[49] ⍝ LOOP IF BFUNCT←1 INDICATING LASH
[50]  →NEXTC×⍳BFUNCT
[51] ⍝ LOOP IF THE TABLE IS NOT GETTING TO LARGE.
[52]  →NEXTC×⍳LF>(⌈/INFORM[;1])-(0≠TABLE)⍳1
[53]  BACKTRACK
[54]  →NEXTC
[55] STOP:AI←⎕AI[2]-AI
[56]  TS2←⎕TS
```

```
     ∇SETINCR;L;S;S0;T;R;C;BCK;CH;LT
[1]  ⍝ THIS FUNCTION PREPROCESS SOME OF THE CALCULATIONS
[2]  ⍝ REQUIRED ONLY ONCE FOR NO BACKTRACKING AND REQUIRED
[3]  ⍝ FOR EACH BACKTRACKING BUT NEED ONLY BE DONE ONCE.
[4]  ⍝
[5]  ⍝ FOR ANY LETTER  C
[6]  ⍝   NCS[C;1] AND NCS[C;2] GIVEN INDEXES INTO NCI -
[7]  ⍝    NCI WILL REFERENCE INFORM VALUES FOR SUBSTRACTIONS
[8]  ⍝    IN FINDING NON USABLE INCREMENTS.
[9]  ⍝   NCS[C;3] IS THE LETTER
[10] ⍝   NCS[C;4] IS THE LETTER POSITION
[11] ⍝   NCS[C;5 6] INDICATE START AND SIZE OF LETTERS GROUP
[12] ⍝
[13]  NCS←((CC←⌈/,WSYM),6)⍴0
[14]  NCI← 0 2 ⍴0
[15]  L←⍴LETTERS
[16]  LT←1↑⍴INFORM
[17]  S←⍳⍴ALPH
[18] LOOPL:NCS[S0←S0/,WSYM[;L];3]←(S0←,WSYM[;L]≠0)/S
[19]  NCS[S0;4]←L
[20]  →LOOPL×⍳0≠L←L-1
[21] ⍝
[22]  CH←⌈/CHECK×CHECK≠¯1⊖CHECK
[23]  BCK←(⍴CH)⍴0
[24] RESET:BCK←BCK⌈(0=CH)×1+¯1⌽BCK
[25]  →RESET×⍳0∨.=CH←CH⌈(¯1⌽CH)×0=CH
[26] ⍝
[27]  C←0
[28] NEXTC:→BOTTOM×⍳CC<C←C+1
[29]  NCS[C;1]←1↑⍴NCI
[30]  NCS[C;5]←L
[31]  T←L←L+NCS[C;6]←C+.=HSTCK
[32] ⍝
[33] NEXTT:→NEXTC×⍳LT<T←T+(T↓CH)⍳C
[34]  S←(T↓BCK)⍳0
[35]  R←BCK[T-1]+1
[36]  NCI←NCI,[1](,⍉(S,R)⍴(T-⍳R)),[1.5],(R,S)⍴T+¯1+⍳S
[37] ⍝
[38]  BCK[T]←1
[39]  BCK[T+S-1]←S+BCK[T-1]
[40]  CH[T+S-1]←CH[T-1]
[41]  T←T+S
[42]  →NEXTT
[43] ⍝
[44] BOTTOM:NCS[;2]←((1↓NCS[;1]),1↑⍴NCI)-NCS[;1]
```

```
     ∇BACKTRACK
[1]  ⍝ INCREASE COLISION COUNT.
[2]   →0×⍳BFUNCT←COL<COLS←COLS+1
[3]  ⍝ TYPE DETERMINES HOW FAR BACK TO GO.
[4]   TYPE1
[5]  ⍝ RESET MORE RESENT LETTER VALUES TO ZERO.
[6]   VL←VL×WSYM≤C
[7]  ⍝ REMOVE MORE RESENT WORDS FROM THE TABLE.
[8]   TABLE←TABLE×NCS[C;5]≥TABLE
[9]   INFORM[;1]←INFORM[;3]
[10]  D←⍴LETTERS
[11] ⍝ RESET WORD VALUES.
[12] NEXTD:INFORM[;1]←INFORM[;1]+,VL[INFORM[;3+D];D]
[13]  →NEXTD×⍳0<D←D-1
[14]  C←C-1
```

```
     ∇TYPE1;A;B;S;CHAR
[1]  ⍝ DETERMINE HOW FAR BACK TO BACKTRACK.
[2]  ⍝
[3]  ⍝ FIND THE HIGHEST VALUE WORD.
[4]   A←INFORM[;1]⍳⌈/,INFORM[;1]
[5]  ⍝ FIND THE LOWEST VALUE WORD IN THE GROUP.
[6]   B←GROUP[VALUES⍳⌊/VALUES]
[7]  ⍝ FIND A LETTER IN B NOT IN A.
[8]  ⍝ INCREMENT IT BY ONE.
[9]   C←⌈/,(~CHECK[B;]∈CHECK[A;])/CHECK[B;]
[10]  CHAR←NCS[C;3]
[11]  S←NCS[C;4]
[12]  VL[CHAR;S]←VL[CHAR;S]+1
```

```
     ∇PRINT;A;P;ROWS;WORDSACROSS
[1]  ⍝ PRINTS THE INFORMATION
[2]  ⍝ OPΔ SEND THE LINES TO A FILE IF A FILE HAS BEEN
[3]  ⍝ SPECIFIED. THE LINES ARE STILL PRINTED.
[4]   OPΔ FUNCT,'ING STARTED AT  ',▼TS1
[5]   P←0>A←TS2-TS1
[6]   OPΔ ' TIME DURATION WAS ',▼(-1φP)+A+P× 0 12 31 24 60 60 1000
[7]   OPΔ ' CPU SECONDS USED IN',FUNCT,' IS ',▼0.001×AI
[8]   OPΔ ' NUMBER OF TIMES THOUGH',FUNCT,' MAIN LOOP IS ',▼K
[9]   OPΔ '  '
[10]  OPΔ((¯17+⌊⎕PW÷2)⍴' '),'TERMINATION AFTER BACKTRACK ',▼COLS
[11]  OPΔ((⌊(⎕PW-2+⍴A)÷2)⍴' '),A←'LETTERS USED ',▼LETTERS
[12]  A←(P←WSYM∨.≠0)/ALPH
[13]  P←P⌿VL
[14]  OPΔ '  '
[15]  OPΔ((¯8+⌊⎕PW÷2)⍴' '),'LETTER VALUES'
[16]  OPΔ(((⍴A),⌊(⎕PW-5×1+1↓⍴P)÷2)⍴' '),'''',A,'''', 5 0 ▼P
[17]  OPΔ '  '
[18]  OPΔ((¯8+⌊⎕PW÷2)⍴' '),'HASH TABLE'
[19]  OPΔ '  '
[20]  A←(P←TABLE≠0)/⍳⍴TABLE
[21]  P←(▼((⍴A),1)⍴A),(¯1+P+.=' ')φP←DATA[INFORM[(P/TABLE);2];]
[22]  ROWS←⌈(1↑⍴P)÷WORDSACROSS←⌊⎕PW÷1↓⍴P
[23]  OPΔ(ROWS,WORDSACROSS×1↓⍴P)⍴(,P),⎕PW⍴' '
```