COMPARISON OF PERFECT HASHING METHODS

By

QIZHI TAO

Master of Science

Harbin Institute of Technology

Harbin, P R China

1991

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the degree of
MASTER OF SCIENCE
July, 1999

COMPARISON OF PERFECT HASHING METHODS

Thesis Approved:

J Chandler

Thesis Adviser

Jacques E. LaFrance

Wayne B. Powell

Dean of the Graduate College

PREFACE

This study was conducted to compare two minimal perfect hashing methods, Chang's method and Jaeschke's method. Since hashing is a widely used technique for store data in symbol table and the data are strings of characters, this study focuses on the performance of these methods with the letter-oriented set and gives their run time performance curves. Through the analysis of run time and space complexity, an optimal method is given to make each algorithm performance well.

I sincerely thank my M. S. Committee—Drs. J. P. Chandler, J. Lafrance, and H. K. Dai---for guidance and support in the completion of this research.

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to my advisor, Dr. J. P. Chandler, for his intelligent supervision, constructive guidance, inspiration and friendship. My sincere appreciation extends to my other committee members Dr. J. Lafrance and Dr. H. K. Dai, whose guidance, assistance, encouragement, and friendship are also invaluable.

I also like to give my special appreciation to my parents Prof. Chongde Tao and Ms. Aihua Zhou for their support of my studies, strong encouragement at times of difficulty, love and understanding throughout the whole process.

Finally, I would like to thank the Department of Computer Science for support during these two years of study.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUTION

Hashing is a well-known technique for storing data. With this technique, a key is transformed into a pseudorandom number and this number provides us with a good guess where the key and its associated information are located. Using hashing as a data organization and data retrieving method may cause the key-collision problem.

To handle the key-collision problem, there are several perfect hashing methods proposed by some researchers. Much work has been done to develop perfect hashing functions.

Among these methods, there are about five classic algorithms: Sprugnoli's algorithm, Jaeschke's algorithm, Chang's algorithm, Cichelli's algorithm, and Cook's algorithm [11]. Most of their methods have focused on solving perfect hashing problems on Pascal reserved words and abbreviated symbols for the twelve months.

The goal of this project is to compare some of the methods in details. First I use the C programming language to implement the algorithm calculation, and then I give the minimal perfect hashing function for the reserved words of C programming language. Based on these results, this project will analyze the time and space complexity, discuss the advantages and disadvantages of each method, and give some advice and suggestions about improving the efficiency of these perfect hashing methods.

Chapter 2

LITERATURE REVIEW

2.1 Hashing and its Application

Often a computer program needs to accept all or part of its input as a sequence of character strings and decide, for each string, whether that string is a member of some finite set of known strings. The set of known strings may be nonempty when the program starts and may change as the program receives input. The strings, both known and otherwise, are generally referred to as *keys.* Testing a key for membership in the set of known keys is called a *search,* adding a key to the set of known keys is called an *insertion,* and removing a key from the set is a *deletion.*

Many different schemes have been developed to handle this computational task. These include linear search of an unordered table, binary search of an ordered table, B-trees, tries, various forms of string pattern matching, and hashing. By using a binary search tree, we will have the worst case complexity for these operations of O(n). If we use some refinements of the binary search tree, that would be O(log n). But can it be better? Yes, hashing is the solution for this.

Hashing refers to schemes that use some simple arithmetic function of a key as the location in the table at which the key should be stored. With this technique, implementing insertion, deletion and finding operations on ADT (abstract data type) can be accomplished in constant average time. Unlike the search tree method that relies on

identifier comparisons to perform a search, hashing relies on a formula called the *hash function*. The table in which identifiers are stored is the *hash table.*

Hashing applications are abundant. Compilers use hash tables to keep track of declared variables in source code. Since hashing can be used to implement searching, inserting and deleting in constant average time, hashing is the ideal application for implementation of the *symbol table*. The other reason is the identifiers are typically short, so the hash function can be computed quickly [43].

Hashing is useful for any graph theory problem where the nodes have real names instead of numbers. Here, as the input is read, vertices are assigned integers from one onward by order of appearance. Again, the input is likely to have large groups of alphabetized entries. If a search tree is used, there could be a dramatic decrease in efficiency.

A third common use of hashing is in programs that play games. As the program searches through different lines of play, it keeps track of positions it has seen by computing a hash function based on the position (and storing its move for that position). If the same position reoccurs, usually by a simple transposition of moves, the program can avoid expensive re-computation. This general feature of all game-playing programs is known as the *transposition table.*

Another use of hashing is in on-line spelling checkers. If misspelling detection (as opposed to correction) is important, an entire dictionary can be prehashed and words can be checked in constant average time [6].

Currently, hashing is widely used in natural language understanding systems, programming system such as compilers and interpreters and other application systems where data are stored and retrieved frequently.

2.2 The Hashing Table and Hashing Function

2.2.1 The Hashing Table

The hashing table is a sequentially mapped data structure that makes use of the random-access capability afforded by sequential mapping. We use an arithmetic function, f, to determine the address, or location of an identifier in the table. The hash table ht is stored in sequential memory locations that are partitioned into b buckets, $ht[0]$,.....ht [b-1]. Each bucket has s slots. Usually s =1 which means that each bucket holds exactly one record. The important part of hashing table is the size of the table that is referred to as TableSize (denoted as m in Fig. 1) since each key is mapped into some number in the range 0 to TableSize-1 and placed in an appropriate cell.

2.2.2 The Hashing Function

The hashing function is the function used to transform the identifier into an address in the hash table. Using hashing function f, we can compute a hashed value for each identifier $h(k_i)$. That is k_i hashes to slot $T[h(k_i)]$ in hash table T.

The advantages of this approach are that, if we pick the hash function properly, TableSize can be chosen so as to be proportional to the number of elements actually stored in table T [44].

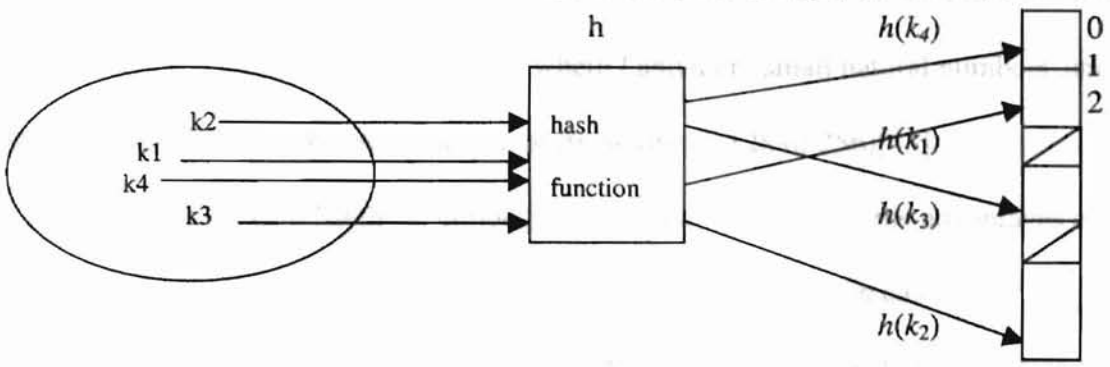


Figure 1 Hash Table Implement of the DICTIONARY ADT

Criteria for a good hash function:

- The hash address is easily calculated.
- The loading factor (LF) of the hash table is high for a given set of keys. (The LF is the fraction of used or occupied hash table locations in the total hash table locations).
- The hash addresses of a given set of keys are distributed uniformly in the hash table.

There are a wide variety of hash functions. Here are a number of specific techniques used to create hash functions [22].

Division Method Hash functions that make use of the division method generate hash values by computing the remainder of k divided by m:

$$h(k) = k \bmod m \qquad (1)$$

With this hash function, $h(k)$ will always compute a value that is an integer in the range 0, 1,....., m-1.

The choice of m is critical to the performance of the division method. For instance choosing m as a power of 2 is usually ill-advised, since $h(k)$ is simply the p least significant bits of k whenever $m=2^p$. In this case the distribution of keys in the hash table is based on only a portion of the information contained in the keys.

In general, the best choices for m when using the division method turn out to be prime numbers that do not divide $r^l \pm a$, where l and a are small natural numbers, and r is the radix of the character set we are using (typically r =128 or 256)[43].

Multiplication Method Although the division method has the advantages of being simple and easy to compute, its sensitivity to the choice of m can be overly restrictive. The principal advantage of the multiplication method is that the choice of m is not critical----in fact, m is often chosen to be a power of 2 in fixed-point arithmetic implementations.

Hash functions that make use of the multiplication method generate hash values in two steps. First the fractional part of the product of k and a real constant A, where $0< A < 1$, is computed. This result is then multiplied by m before applying the floor function to obtain the hash value:

$$h(k) = \lfloor m(kA \mod 1) \rfloor. \qquad (2)$$

Note that kA mod 1 means $kA - \lfloor kA \rfloor$ yields the fractional part of the real number kA. Since the fractional part must be greater than or equal to 0, and less than 1, the hash values must be integers in the range $0,1,\ldots, m-1$. One choice of A that often does a good job of distributing keys throughout the hash table is the inverse of the golden ration:

$$A = \Phi^{-1} \approx 0.61803399 \qquad (3)$$

The multiplication method exhibits a number of nice mathematical features. Because the hash values depend on all bits of the key, permutations of a key are no more likely to collide than any other pair of keys [43].

Universal Hashing If a malicious adversary chooses the keys to be hashed, then he can choose n keys that all hash to the same slot, yielding an average retrieval time of

$\Theta(n)$. Any fixed hash function is vulnerable to this sort of worst-case behavior; the only effective way to improve the situation is to choose the hash function randomly in a way that is independent of the keys that actually going to be stored. This approach, called universal hashing, yields good performance on the average [17, 25, 26].

Let H be a finite collection of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m-1\}$. Such a collection is said to be functions $h \in H$ for which $h(x) = h(y)$ is precisely $|H| / m$. In other words, with a hash function randomly chosen from H, the chance of a collision between x and y when $x \neq y$ is exactly $1/m$, which is exactly the chance of a collision if $h(x)$ and $h(y)$ are randomly chosen from the set $\{0, 1, \ldots\ldots, m-1\}$. Universal hashing has not been used much, if any, in practice.

2.3 Collision Resolution Strategies

A problem we must deal with when we use hashing is deciding what to do when two keys hash into the same value (this is known as a *collision*). Although we should strive to construct hash functions that minimize collisions, in most applications it is reasonable to assume that collisions will occur. Therefore the manner in which we resolve collisions will directly affect the efficiency of the operations on the ADT.

2.3.1 Separate Chaining

One of the simplest collision resolution strategies, called separate chaining, involves placing all elements that hash to the same slot into a linked list. In this case the slots in the hash table will no longer store data elements, but rather pointers to linked lists, as shown in Figure 2. This strategy is easily extended to allow for any dynamic data

structure. Note that with separate chaining, the number of items that can be stored is only limited by the amount of available memory. The disadvantage is that each linked list can only be searched sequentially, and this is very slow if a list is at all long. Also, the links occupy valuable space [44].

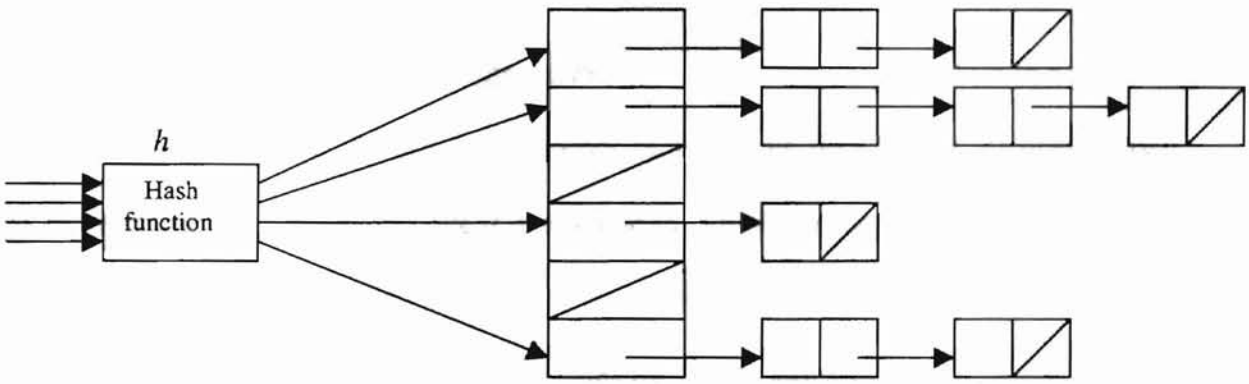


Figure2 The Collision Resolution by Separate Chaining

2.3.2 Open Addressing

In open addressing all data elements are stored in the hash table itself. In this case, collisions are resolved by computing a sequence of hash slots. This sequence is successively examined, or probed, until an empty hash table slot is found in the case of insertion, or the desired key is found in the case of searching or deletion. The memory saved by not storing pointers can be used to construct a larger hash table if necessary. Thus, using the same amount of memory we can construct a larger hash table, which potentially leads to fewer collisions and therefore faster operation implementations.

In open addressing, the ordinary hash functions which perform a mapping from the universe of keys U to slots in the hash table $T[0..m-1]$ will be modified so that they use both a key and a probe number when computing a hash value. This additional information is used to construct the probe sequence. More specifically, in open addressing, hashing functions perform the mapping:

$H: \quad U \times \{0, 1, \ldots, \infty\} \rightarrow \{0,1, \ldots, m\text{-}1\}$ and produces the probe sequence

$$< h(k,0),\ h(k,1),\ h(k,2), \ldots\ldots >$$

Because the hash table contains m slots, there can be at most m unique values in a probe sequence. Note, however, that for a given probe sequence we are allowing the possibility of $h(k, i) = h(k, j)$ for $i \neq j$. Therefore it is possible for a probe sequence to contain more than m values.

There are three main probing strategies for open addressing.

1) Linear Probing. This is one of the simplest probing strategies to implement; however, its performance tends to decrease rapidly with an increasing load factor (LF).

If the first location probed is j, and c_1 is a positive constant, the probe sequence generated by linear probing is:

$$< j,\ (j + c_1 \times 1) \bmod m.\ (j + c_1 \times 2) \bmod m,\ \ldots\ldots >.$$

Given any ordinary hash function $h': U \rightarrow \{0, 1, \ldots, m-1\}$, a hash function that uses linear probing is easily constructed using:

$$h(k, i) = (h'(k) + c_1 i) \bmod m \qquad (4),$$

where $i = 0,1,\ldots m\text{-}1$ is the probe number. Thus the argument supplied to the module operator is a linear function of the probe number.

The use of linear probing leads to a problem known as *clustering*----elements tend to clump (or cluster) together in the hash table in such a way that they can only be accessed via a long probe sequence.

There are two factors in linear probing that lead to clustering. First, every probe sequence is related to every other probe sequence by a simple cyclic shift. Specifically, if we interpret a given probe sequence as a q-permutation ($q \leq m$) of q shift of this permutation, this leads to a specific form of clustering called *primary clustering*. Because any two probe sequences are related by a cyclic shift, they will overlap after a sufficient number of probes. A less severe form of clustering, called secondary clustering, results from the fact that if two keys have the same initial hash value $h(k_1, 0) = h(k_2, 0)$, then they will generate the same probe sequence---$h(k_1, i) = h(k_2, i)$, for $i = 1, 2\ldots\ldots$, m-1. Primary clustering results if the resolution method follows an established chain of collisions no matter where it enters the chain; secondary clustering results if an established chain of collisions is followed only if it is entered at the beginning of the chain.

2) Quadratic Probing. This is a simple extension of linear probing in which one of the arguments supplied to the mod operation is a quadratic function of the probe member. More specifically, given any ordinary hash function h', a hash function that uses quadratic probing can be constructed using:

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \quad (5),$$

where c_1 and c_2 are positive constants. Once again, the choices for c_1, c_2, and m are critical to the performance for this method. Since the left-hand argument of the mod operation in equation (5) is a nonlinear function of the probe number, probe sequences

cannot be generated from other probe sequences via simple cyclic shifts. This eliminates the primary clustering problem and tends to make quadratic probing work better than linear probing. However, as with linear probing, the initial probe $h(k, 0)$ determines the entire probe sequence, and the number of unique probe sequences is m. Thus, secondary clustering is still a problem.

3) Double Hashing. Given two ordinary hash functions h'_1 and h'_2, double hashing computes a probe sequence using the hash function

$$h(k, i) = (h'_1(k) + i\, h'_2(k)) \mod m \qquad (6)$$

Note that the initial probe $h(k, 0) = h'_1(k) \mod m$, and that successive probes are offset from previous probes by the amount $h'_2(k) \mod m$. Thus the probe sequence depends on k through both h'_1 and h'_2. This approach avoids both primary and secondary clustering by making the second and subsequent probes in a sequence independent of the initial probe. The probe sequences produced by this method have many of the characteristics associated with randomly chosen sequences, which makes the behavior of double hashing a good approximation to uniform hashing [45].

2.3 Table Overflow

In practice, if there is an insertion operation on a full table, that will cause table overflow. If separate chaining is being used, this is typically not a problem since the total size of the chains is only limited by the amount of available memory in the free store. Thus the discussion to table overflow in open address hashing is needed.

Two techniques that circumvent the problem of table overflow by allocating additional memory will be considered. In both cases, it is best not to wait until the table

becomes completely full before allocating more memory; instead, memory will be allocated whenever the load factor a exceeds a certain threshold which is denoted as a_{td}.

1) *Table Expansion*: The simplest approach for hashing table overflow involves allocating a larger table whenever an insertion causes the load factor to exceed a_{td}, and then moving the contents of the old table to the new one. The memory of the old table can then be reclaimed. Using this technique with hash tables is complicated by the fact that the output of hash functions is dependent on the table size. This means that after the table is expanded (or contracted), every data element needs to be "rehashed" into the new table. The additional overhead due to rehashing tends to make this method too slow.

2) *Extendible Hashing*: An alternative approach for the problem above is using extendible hashing. Extendible hashing limits the overhead due to rehashing by splitting the hashing table into blocks. The hashing proceeds in two steps: The low-order bits of a key are first checked to determine which block a data element will be stored in, and then the data element is actually hashed into a particular slot in that block using the methods discussed previously. The addresses of these blocks are stored in a directory table. In addition, a value b is stored with the table---this gives the number of low-order bits to use during the first step of the hashing process [44].

Table overflow can now be handled as follows. Whenever the load factor a_{td} of any one block d is exceeded, an additional block d' the same size as d is created, and the elements originally in d are rehashed into both d and d' using $b + 1$ low-order bits in the first step of the hashing process. Of course, the size of the directory table must be doubled at this point, since the value of b is increased by one.

If the block sizes are kept relatively small, the extendible hashing approach will greatly reduce the overhead due to rehashing. Of course, this comes at the expense of the additional time that is spent on comparing low-order bits in the directory table during the first step of the hashing process [41,42].

2.4 Perfect Hashing

In order to overcome the collision problem there was developed a kind of hashing method in the 1970's, which is called perfect hashing [27].

2.4.1 Notation

Definition 2.1 A refinement of hashing which allows retrieval of an item (=key) in a static table with a single probe is called perfect hashing.

Definition 2.2 A hashing function is a *perfect hashing function* for a set of keys if and only if the function is one-to-one on that set of keys, i.e., this is a collision-free hashing function.

Definition 2.3 A hashing function is a *minimal perfect hashing function* for a set of keys if and only if the function maps the keys one-to-one onto the buckets 0, 1, ..., k-1,whe k is the number of keys in the set. That is, it is perfect and it completely fills the table [44].

2.5.2 Development of perfect hashing

Since using hashing as a data organization and data retrieving method may cause the key-collision problem, some collision resolution strategies must be applied to handle them. One strategy of solving key-collision problem is to construct a perfect hashing

function. With this function, a one-to-one mapping from the key set into the address space is established. Therefore, a retrieval operation can be executed in a single step.

Theoretically, it is not difficult to construct a perfect hashing function for an arbitrary given set of keys if the memory space used by the hashing function is not restricted. For example, assume that the values of the keys are all positive and the maximum value is L, then $h(k)= k$ is a perfect hashing function. However, it may lead to a very small loading factor. In order to avoid sparse hash tables, there are several perfect hashing methods that have been developed:

1) Sprugnoli's method: Sprugnoli proposed two simple functions (1) $h(k)=(k + s)/N$ where s and N are integers, and (2) $h(k)=\lfloor((d + kq) \bmod M)/N\rfloor$, where d, q, M, N are integers, as the candidates for constructing perfect hashing functions. There are two algorithms for finding s and N for (1) and d, q, M and N for (2) [8].

2) Jaeschke's method: Jaeschke proposed a method for establishing minimal perfect hashing functions. If $K=\{k_1, k_2,..., k_n\}$ is a set of positive integers, Jaeschke's method attempts to find integer constants C, D and E such that for each k_i in K, $h(k_i)=\lfloor C/(Dk_i+ E)\rfloor \bmod n$ is a minimal perfect hashing function. He gave two algorithm, called Algorithm C and Algorithm DE, to find C and D, E respectively [5].

3) Chang's method: Chang proposed a minimal perfect hashing scheme based on the Chinese remainder theorem. His hashing function is of the form: $h(k)= C \bmod p(k)$, where k belongs to a set $K=\{k_1, k_2,....., k_n\}$ of positive integers and p(k) is a prime number function on K [1,9,13,20, 32].

4) Cichelli's method: Cichelli proposed a heuristic method to build tables and associated hashing functions for a number of particular data sets. In his method, each

character is assigned a value. The form of hashing function is defined as h(word) = length (word) + value (first letter) + value (last letter). That is, the table position can be calculated as the sum of the word length plus the associated values of the first and last letter of the word [2, 3, 4, 6, 16].

5) Cook's method: Cook proposed several algorithms to improve Cichelli's backtracking algorithm for assigning suitable associated values for characters [10,14,15, 34].

Perfect hashing is frequently used for memory efficient storage and fast retrieval of items from a static set, such as reserved words in programming languages, command names in operating system, commonly used words in natural language, etc. Therefore, in the following chapters, we will choose two methods from these five methods mentioned above and analyze their performance for the letter-oriented input sets since most of the input sets are string of characters.

2.5 Other Hashing Methods

There are other hashing methods: non-obvious hashing [30] and spiral hashing [45]. Since they do not have much relation with perfect hashing, they are not mentioned here.

Chapter 3

CHANG'S METHOD: A MINIMAL PERFECT HASHING SCHEME

3.1 Theorems

The following theorems are quoted from [9].

LEMMA 1. [Chinese Reminder Theorem].

Let $r_1, r_2, \ldots, r_n$ be integers. There exists an integer C such that $C \equiv r_1 \pmod{m_1}$, $C \equiv r_2 \pmod{m_2}$, ..., and $C \equiv r_n \pmod{m_n}$, if m_i and m_j are relatively prime for all $i \neq j$.

Theorem 3.1

Given a finite set $K = \{ k_1, k_2, \ldots, k_n \}$ of positive integers, there exists an integer C such that $h(k_i) = C \bmod p(k_i)$ is a minimal perfect hashing function if $p(x)$ is a prime number for every k_i in K.

Corollary 1

Given a finite set $K = \{ k_1, k_2, \ldots, k_n \}$ of positive integers, there exists a hashing function $h(k) = C \bmod p(k)$ such that the keys in K can be stored in ascending order by applying $h(x)$.

LEMMA 2.

Let m_i and m_j be relatively prime where $i \neq j$ and $1 \leq i, j \leq n$. Let $m_1 < m_2 < \ldots < m_n$.

$\sum_{i=1}^{n} b_i M_i i \bmod m_j = j$ if $M_i = \prod_{i \neq j} m_j$ and $b_i M_i \equiv 1 \pmod{m_i}$.

Theorem 3.2

Let m_i and m_j be relatively prime where $i \neq j$ and $1 \leq i, j \leq n$. Let $m_1 < m_2 < \ldots < m_n$.

$C = \sum_{i=1}^{n} b_i M_i i \bmod \prod_{i=1}^{n} m_i$ is the smallest positive integer such that $C \equiv i \pmod{m_i}$, if $M_i = \prod_{i \neq j} m_j$ and $b_i M_i \equiv 1 \pmod{m_i}$.

Theorem 3.3

Let $C = \sum_{i=1}^{n} b_i [\prod_{i \neq j} p(k_j)] i$, where $\prod_{i \neq j} p(k_i) b_i \equiv 1 \ [\bmod p(k_i)]$. The hashing function $h(k) = C \bmod p(k)$ is a minimal perfect hashing function if $p(k)$ is a prime number function for $K = \{ k_1, k_2, \ldots, k_n \}$.

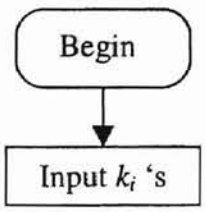
Theorem 3.4

Let $M'b \equiv 1 \pmod{m}$, $(M', m) = 1$, and $M' < m$. Then $b = B_k$, with $B_0 = 1$, $B_1 = -Q_k$ and $B_{j+1} = -B_j Q_{k-j} + B_{j-1}$, where $M' \equiv M \bmod m$.

3.2 Flowchart for Calculating C

Input: $k_1, k_2, \ldots, k_n$

Output: C



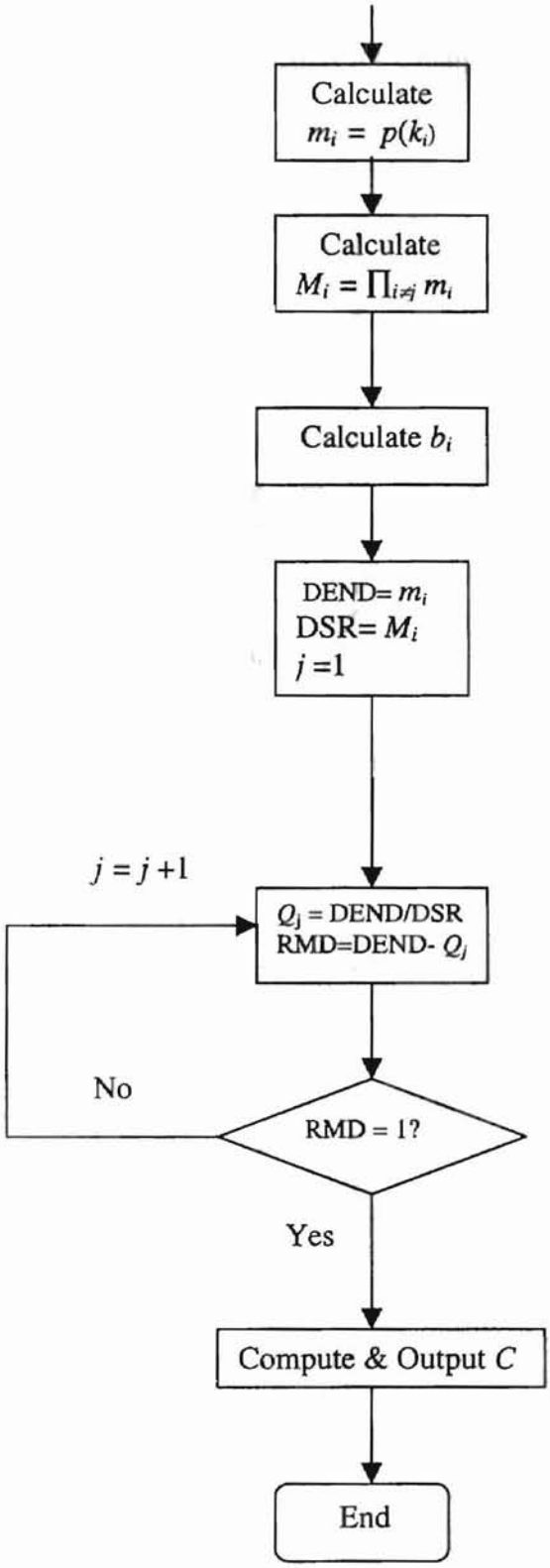


Figure 3 Flowchart for Calculating C value

3.3 Flowchart for Chang's Method

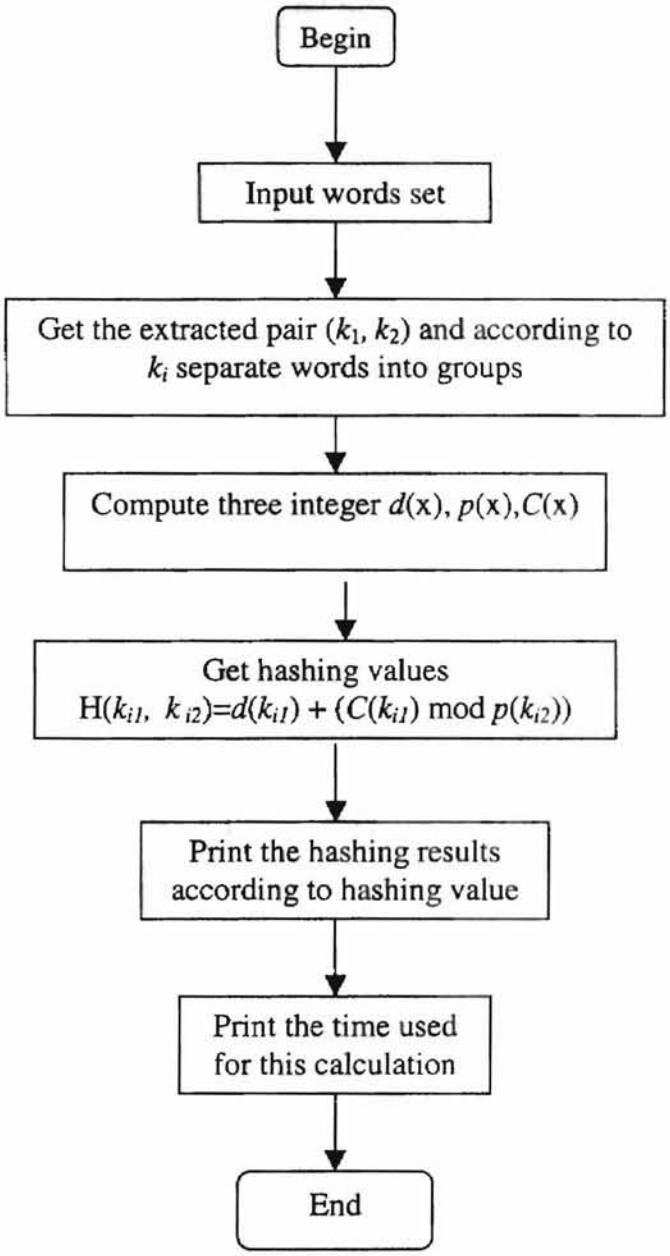


Figure 4 Flowchart for Calculating Hashing Value by Chang's Method

3.4 The C Programming Code for This Method

This is attached in Appendix A.

3.5 Test Sets and Test Results of Chang's Method

3.5.1 The Month Set

a) The input set is: January, February, March, April, May, June, July, August, September, October, November, December

b) The calculating values are:

x =	P	U	E	R	y	O	c
p(x) =	13	3	2	17	5	11	7

X =	A	D	F	J	M	N	O	S
d(x) =	0	2	3	4	7	9	10	11
C(x) =	28	1	1	23	36	1	1	1

Table 1 The Calculating Values of p(x), d(x), and C(x) of the Month Set

c) The test results are:

Group	Extracted Pair	Original Key	Location
1	(A,p)	April	2
	(A,u)	August	1
2	(D,e)	December	3
3	(F,r)	February	4

4	(J,u) (J,e) (J,y)	January June July	6 5 7
5	(M,r) (M,y)	March May	8 9
6	(N,o)	November	10
7	(O,c)	October	11
8	(S,e)	September	12

Table 2 Hashing Results on the Month Set

3.5.2 The Key Words Set of the C Programming Language

a) The input set is: Auto, Break, Case, Char, Const, Continue, Default, Do, Double, Else, Enum, Extern, Float, For, Goto, If, Int, Long, Register, Return, Short, Signed, Sizeof, Static, Struct, Switch, Typedef, Union, Unsigned, Void, Volatile, While

b) The calculating values are:

X	U	R	E	S	T	f	O	L	n	x	G	Z	a	i	y	h
$p(x)$	29	7	37	19	23	11	2	5	17	43	13	53	31	3	47	41

X	A	B	C	D	E	F	G	I	L	R	S	T	U	V	W
$d(x)$	0	1	2	6	9	12	14	15	17	18	20	26	27	29	31
$C(x)$	1	1	29604	409	2841	7	1	155	1	209	779159	1	40	7	1

Table 3 The Calculating Values of $p(x)$, $d(x)$, and $C(x)$ on the Key Words Set of the C Programming Language

c) The test results are:

Group	Extracted Pair	Original Key	Location
1	(A, u)	Auto	1
2	(B, r)	Break	2
3	(C, e)	Case	6
	(C, r)	Char	3
	(C, s)	Const	4
	(C, t)	Continue	5
4	(D, f)	Default	8
	(D, o)	Do	7
	(D, u)	Double	9
5	(E, l)	Else	10
	(E, n)	Enum	11
	(E, x)	Extern	12
6	(F, l)	Float	14
	(F, o)	For	13
7	(G, o)	Goto	15
8	(I, f)	If	16
	(I, n)	Int	17
9	(L, o)	Long	18
10	(R, g)	Register	19
	(R, t)	Return	20
11	(S, o)	Short	21

	(S, g)	Signed	24
	(S, z)	Sizeof	26
	(S, a)	Static	25
	(S, r)	Struct	23
	(S, i)	Switch	22
12	(T, y)	Typedef	27
13	(U, i)	Union	28
	(U, s)	Unsigned	29
14	(V, i)	Void	30
	(V, l)	Volatile	31
15	(W, h)	While	32

Table 4 Hashing Results on the Key Words Set of the C Programming Language

3.5.3 The Frequently Used Words Set

a) The input set is: And, Are, As, At, Be, But, By, From, For, Had, He, Her, His, Have, In, Is, It, Not, Of, On, Or, That, The, This, To, Which, Was, With, You

b) The calculating values are:

X	N	r	S	T	E	u	y	O	d	v	F	h	a	i
p(x)	11	3	5	13	7	37	43	2	19	41	23	29	17	31

X	A	B	F	H	I	N	O	T	W	Y

d(x)	0	4	7	9	14	17	18	21	25	1
C(x)	1642	5034	5	61792	211	1	739	17	13023	1

Table 5 The calculating value of p(x), d(x), and C(x) for the Frequently Used Words Set

a) The test results are:

Group	Extracted Pair	Original Key	Location
1	(A, n)	And	3
	(A, r)	Are	1
	(A, s)	As	2
	(A, t)	At	4
2	(B, e)	Be	5
	(B, u)	But	6
	(B, y)	By	7
3	(F, r)	From	9
	(F, o)	For	8
4	(H, d)	Had	13
	(H, e)	He	12
	(H, r)	Her	10
	(H, s)	His	11
	(H, v)	Have	14
5	(I, n)	In	16
	(I, s)	Is	15

	(I, t)	It	17
6	(N, o)	Not	18
7	(O, f)	Of	21
	(O, n)	On	20
	(O, r)	Or	19
8	(T, t)	That	25
	(T, e)	The	24
	(T, s)	This	23
	(T, o)	To	22
9	(W, h)	Which	27
	(W, a)	Was	26
	(W, i)	With	28
10	(Y, o)	You	29

Table 6 Hashing Results on the Frequently Used Words Set

Chapter 4

JAESCHKE'S METHOD: ANOTHER PEFECT HASHING SCHEME

4.1 Theorems

The following theorems are quoted from [5].

Theorem 4.1 [Reciprocal Hashing]

Given a finite set $W = \{w_1, w_2, \ldots, w_n\}$ of positive integers, there exist three integer constants C, D, E such that h defined by

$$h(w) = \lfloor C/(Dw + E) \rfloor \bmod n$$

is a minimal perfect hashing fuction.

LMMEA 4.1 For any set $W = \{w_1, w_2, \ldots, w_n\}$of positive integers, there exists two integer constants D, E such that

$Dw_1 + E, Dw_2 + E, \ldots, Dw_n + E$

are pairwise relatively prime.

4.2 The Algorithm for Calculating C

Let $W = \{w_1, w_2, \ldots, w_n\}$ consist of positive integers with $w_1 < w_2 < \ldots < w_n$. Then the algorithm to find an integer C such that the following condition is satisfied

$$\lfloor C/w_i \rfloor \neq \lfloor C/w_j \rfloor \bmod n, \qquad \text{for all } i, j, \text{ with } 1 \leq i \leq j \leq n \quad (4.3)$$

The algorithm starts with an arbitrary positive integer $C = C_0$. Then the residues of $\lfloor C/w_i \rfloor \bmod n$ are calculated. If they are all different from each other the algorithm terminates successfully. Otherwise the actual C is increased conveniently by a certain

amount $\alpha(C, W)$, and the new C is examined in the same way. The algorithm terminates unsuccessfully if C exceeds a prescribed limit L.

4.2.1 Starting Constant --- C_0

Usually we start from $C_0 = 1$. If the identifier sets W with a small difference $w_n - w_1$, in order to avoid the unnecessary calculating, we choose:

$$C_0 = [\ (n\text{-}2)\ w_1\ w_n\ /(\ w_n - w_1\ )] \qquad (4.1)$$

as a reasonable start.

4.2.2 Increment --- $\alpha(C, W)$

In order to get $\alpha\ (C, W)$, we examine only such integers C that are multiples of at least one element w_i of W. This is clear because a C value that is not a multiple of any element of W gives a remainder:

$r_i \equiv C \bmod w_i \quad (0 < r_i < w_i\ )$

and by taking the minimum of these r_i , referred as r_0, the quotients $\lfloor\ C/\ w_i\ \rfloor$ equals the quotients $\lfloor\ C'/\ w_i\ \rfloor$ where $C' = C - r_0$; C' is a multiple of w_0. That means $\alpha(C, W)$ should be one of the numbers:

$w_1 - r_1,\ w_2 - r_2\ ,\ \ldots,\ w_n - r_n,$

where $r_i = C - \lfloor\ C/\ w_i\ \rfloor\ w_i$. We choose

$$\delta_{ij} = \min\ \{\ w_i - r_i,\ w_j - r_j\},$$

$$K(C, W) = \{\ (i, j)\ |\ 1 \le i\ < j \le n \wedge \lfloor\ C/\ w_i\ \rfloor \equiv \lfloor\ C/\ w_j \rfloor \bmod n\} \text{ and}$$

$$a'(C, W) = \max_{(i,j) \in K(C,W)} \delta_{ij}$$

then $a'(C, W)$ should be an appropriate increment of C.

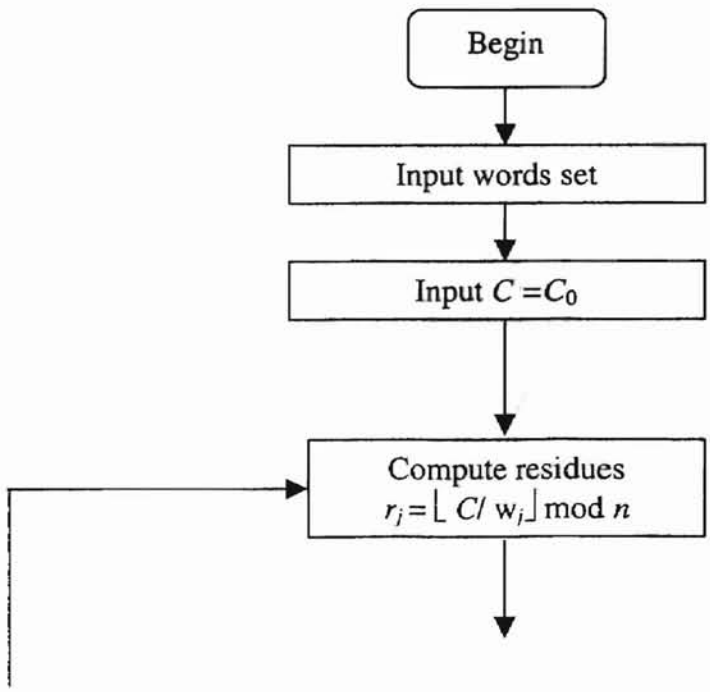
4.2.3 Limitation of Calculate C ---L

A natural limit for the C value to be inspected is:

$$L = n \cdot \mathrm{lcm}(w_1, w_2, \ldots, w_n), \qquad (4.2)$$

where scm means "smallest common multiple". Because if a $C > L$ of the desired kind exists, the $C - L$ is also a C value which satisfies Eq. (4.1). That means if no $C < L$ satisfies Eq. (4.2), then no C exists at all such that Eq. (4.2) holds. The number L determined by Eq. (4.2) is generally very large and therefore not adequate for the termination of Algorithm C. Therefore we have a upper bound value of L to avoid the C value to be examined getting too large.

4.3 Flowchart for Calculating C



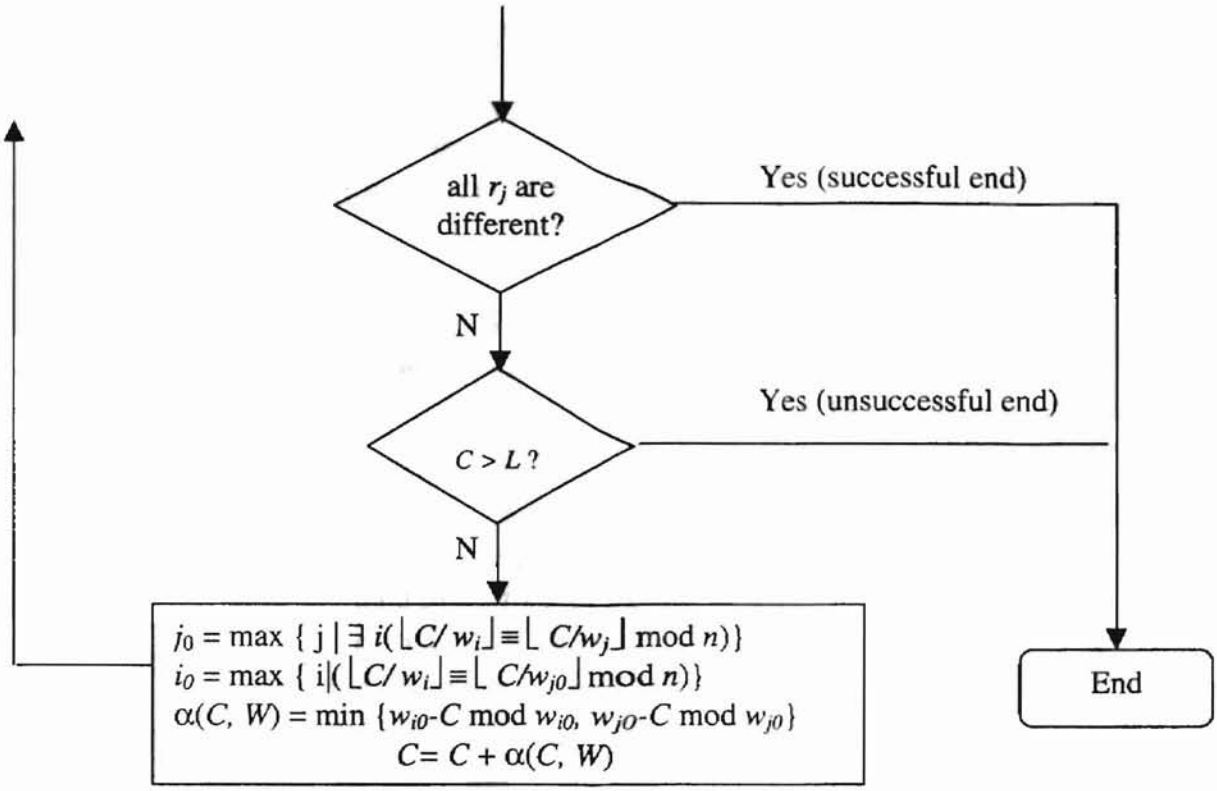
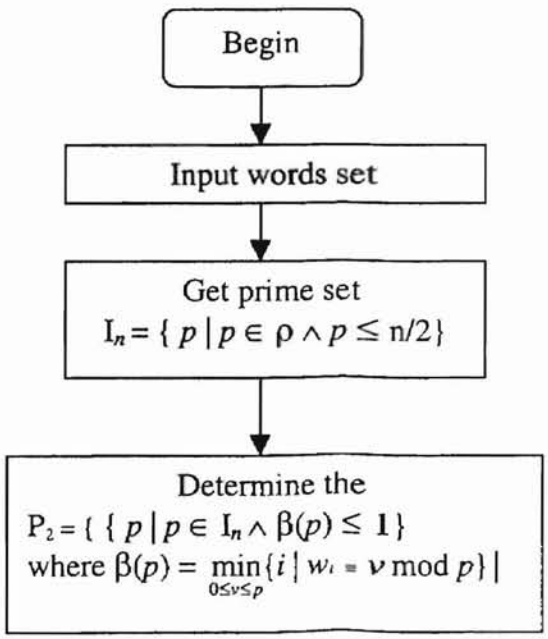


Figure 5 Flowchart for Calculating C

4.4 Flowchart for Calculating D and E



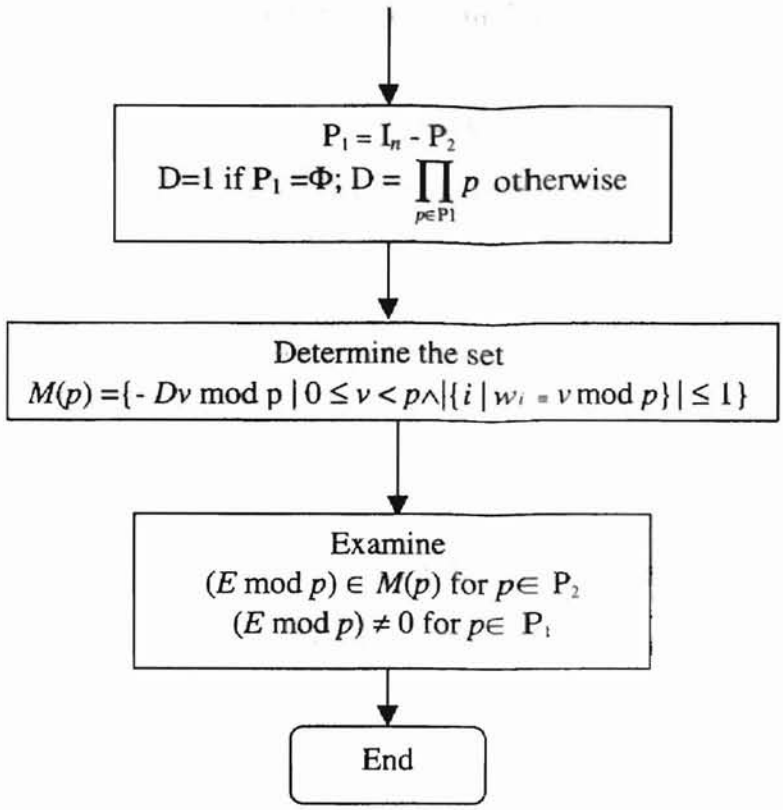


Figure 6 Flowchart for Calculating *D* and *E*

4.5 The C Programming Code for This Method

This is attached in Appendix A.

4.6 The Test Sets and Test Results of Jaeschke's Method

4.6.1 Twelve Months Set

a) The input set is: January, February, March, April, May, June, July, August, September, October, November, December

b) The calculating values are: *C*0= 4039; *C*=29952

4.6.2 The Key Words Set of the C Programming Language

a) The input set is: Auto, Break, Case, Char, Const, Continue, Default, Do, Double, Else, Enum, Extern, Float, For, Goto, If, Int, Long, Register, Return, Short, Signed, Sizeof, Static, Struct, Switch, Typedef, Union, Unsigned, Void, Volatile, While

b) The calculating values are: $C = 49329781$, $D = 10140585$, $E = 4137$

4.6.3 The Frequently Used Words Set

a) The input set is: And, Are, As, At, Be, But, By, From, For, Had, He, Her, His, Have, In, Is, It, Not, Of, On, Or, That, The, This, To, Which, Was, With, You

b) The calculating values are: $C = 78645213$, $D = 8541735$, $E = 5423$

Chapter 5

COMPARISON OF TWO METHODS

5.1 Run Time Analysis

There are many aspects that affect the run time of the algorithm. Here are three of them: number of words in the set, length of words in the set, and distribution of words. All these aspects will be discussed separately.

5.1.1 Number of Words in the Set and Their Impact

1) Run Time for Chang's Algorithm

a) Theoretical Analysis. Since the run time for getting the abstracted pair is $O(n)$, the run time for calculating the $p(x)$, $d(x)$ is $O(n)$. And for $C(x)$, it is $O(mn)$, where m is the number of iteration for calculating b. But because $m \leq n$, so $O(mn) \leq O(n^2)$.

b) Empirical result. The actual run time of Chang's algorithm is shown in Figure 7. From the test result we have the fit curve function is $y= 12 + 0.37x^{1.6}$. The actual run time of this method is $O(n^{1.6})$.

2) Run Time of Jaeschke's Algorithm

a) Theoretical Analysis. Run time for calculating C is $O(kn)$, where k is the upper bound of the value of the Input set as shown in Figure 8. The run time should add the run time of calculating D, which is $O(n)$, and run time for calculating E, which is $O(c^n)$,

where c is the maximum difference w_i - w_j . Thus the run time of this algorithm is $O(a^n)$, where a is a certain constant.

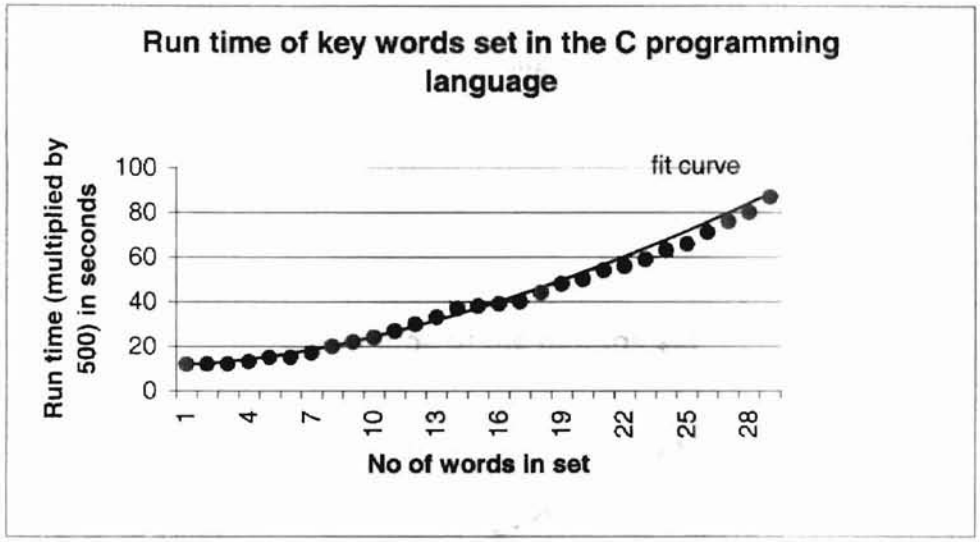


Figure 7 Run time of Chang's Algorithm

b) Empirical Result. Figure 8(a) shows the semi-log plot of a month set for which the C value is successful calculated. In the curve we get the y= 0.39x-3. Figure 8(b) the semi-log plot of the key words set of the C programming language for which C, D and E are calculated. We have y = 0.34x-0.5. Thus these prove the run time of this algorithm is $O(a^n)$, where in the month set, a is $e^{0.39}$ and in the key words set of the C programming language, a is $e^{0.34}$.

3) Comparison between the Two Algorithms

Since Chang's algorithm always costs $O(n^c) \leq O(n^2)$, where c is constant and less than 2, theoretically, it is better than Jaeschke's algorithm. But in the actual calculation this is not always the case. As shown in Figure10, we can see that for the small set ($n \leq 15$), Jaeschke's algorithm is always faster than Chang's algorithm. For a large set ($n>15$), Chang's algorithm is a better choice.

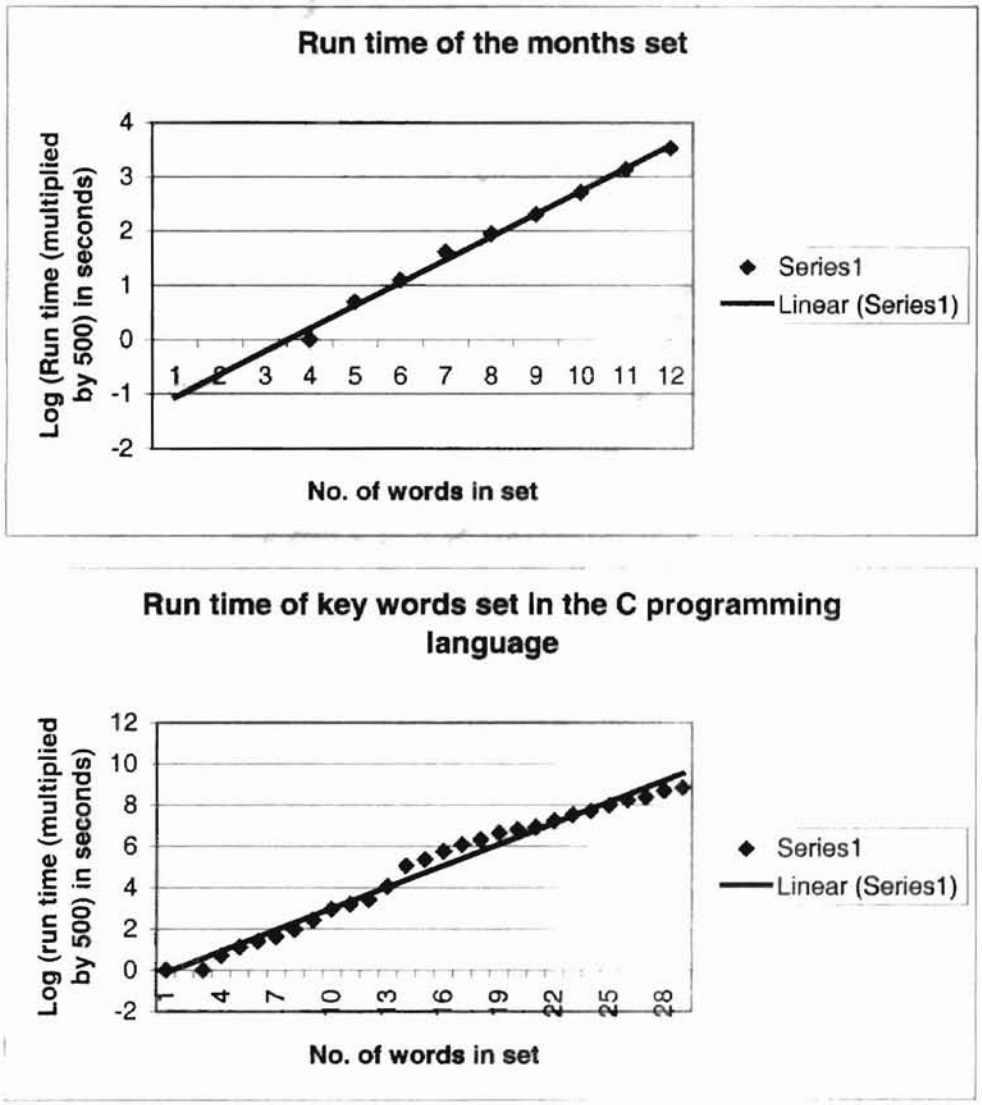


Figure 8. Run Time of Jaeschke's Algorithm (a) the Month Set
(b) Key words set of the C programming language

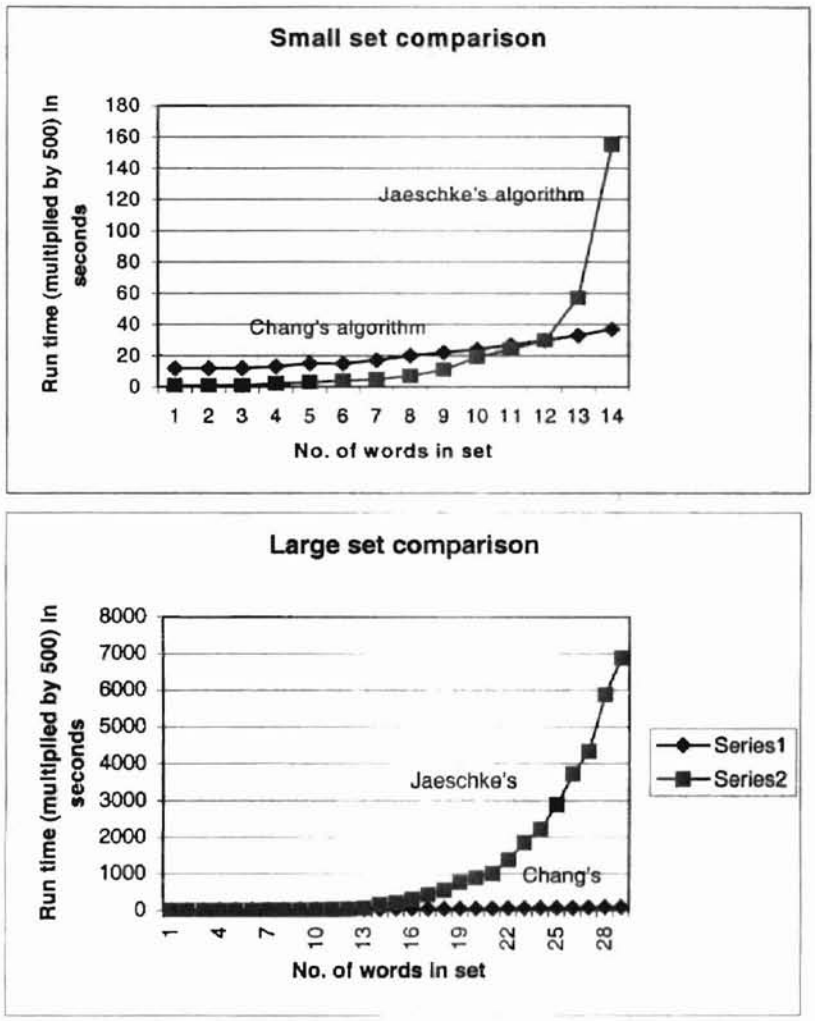


Figure 9. Comparison the Run Time between the Two Algorithms
(a) Small set (b) Large set

5.1.2 The Impact of the Length of Words in the Set on Run Time

1) Chang's Algorithm

The most important factor that affects Chang's algorithm's run time is the time of getting the abstracted pair. The length of words in the set does not affect the time of getting the abstracted pair. In order to test this, we choose five sets: 3-character set, 4-

character set, 5-character set, 7-character set and greater than 7-character set. Each word in one of these sets has same length except that in the greater than 7-character set. The results show the length of words has no effect on run time of this method as we can see in Figure 10(a).

2) Jaeschke's Algorithm

The run time of this algorithm depends on the number of iteration for calculating C, D and E, and the number of iteration is also determined by the value of each word in the set.

Since the value of the word increases as the length of words goes up, the length of the words does have an effect on the run time of this algorithm as we can see in Figure 10(b).

3) Test Sets for the Run Time of Each Algorithm

Here is the test sets of these algorithms. Their run time performance will be discussed later on.

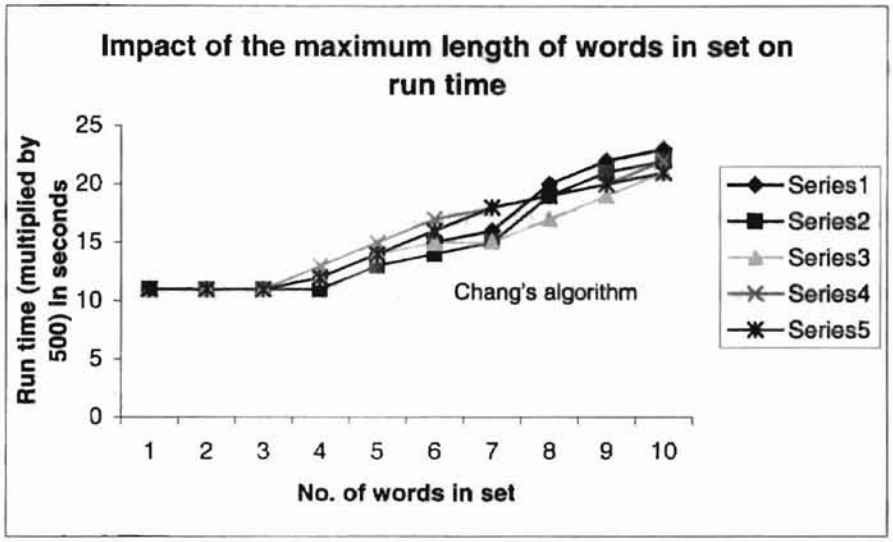
a) 3-character Set. And, bee, car, cob, cog, dip, din, dad, eat, foe

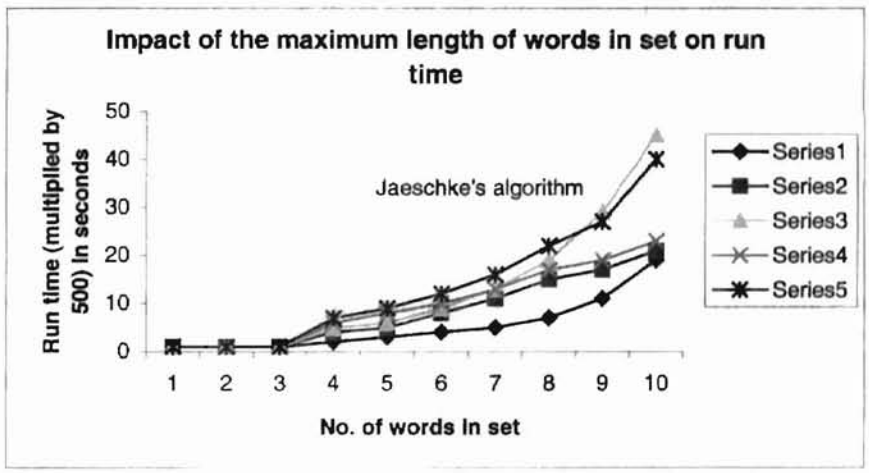
b) 4-character Set. Auto, beat, case, char, cone, deep, dome, dose, else, flag

c) 5-character Set. Among, break, crank, crazy, creep, dense, decoy, deceit, elect, false.

d) 7-character Set. Alumnus, bracket, creator, crazily, creeper, dancing, density, deduce, elastic, forgive.

e) Greater Than 7-character Set. Ambiguous, breakfast, creativity, craziness, cremation, deductive, departure, decorative, elasticity, fragment.





Series 1—curve for 3-character set; Series 2—curve for 4-character set;
Series 3 – curve for 5-character set; Series 4 –curve for 7-character set;
Series 5 curve for greater than 7-character set.

Figure 10 Impact of the Length of the Words in Set on the Two Algorithms
(a) Chang's Algorithm (b) Jaeschke's algorithm

5.1.2 Impact of the Distribution of Words on Run Time

1) Chang's Method

Since the run time of this algorithm is affected by the time of getting the abstracted pair, the distribution of words has an effect on its run time. If the words are all concentrated in one small region of the alphabet, it is difficult to get the abstracted pair, and this will cause the run time to go up very quickly. This is shown in Figure 11(a). On the other hand, if the words focus on a small range of characters, we sometimes need to separate them into groups to get a good performance of this algorithm.

2) Jaeschke's Method

The run time of Jaeschke's method depends only on the word value. The distribution does not have any considerable effect on the run time of this method as we can see from Figure 11(b).

3) Test sets

Here is the test sets of these algorithms. Their run time performance will be discussed later on.

a) Uniform Set. About, body, come, data, elect, flag, hear, jeans, kind, lady.

b) Concentrated Set. Easy, elect, eager, erect, establish, erupt, equal, engage, exercise, exit.

Figure 11shows the distribution of the words and their effects.

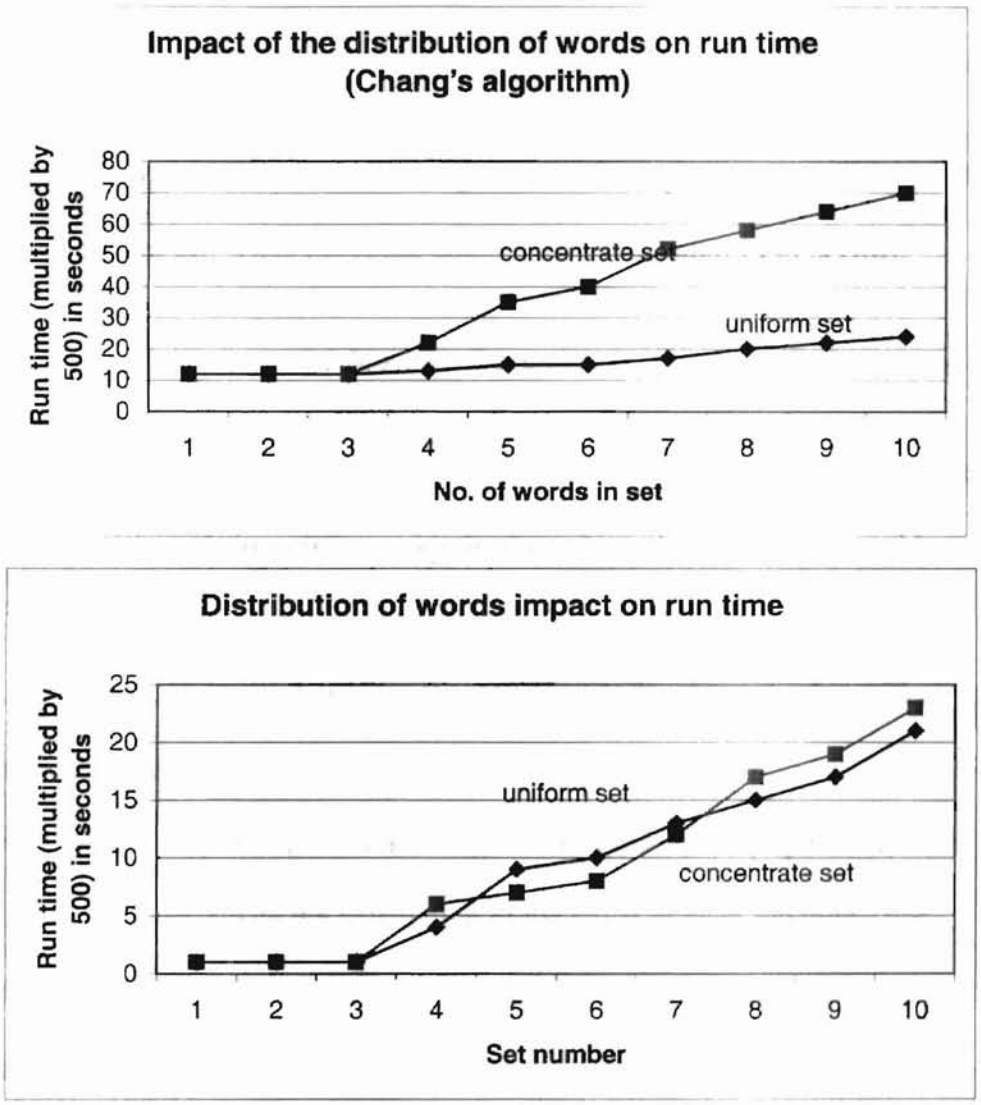


Figure 11 Impact of the Distribution of the Words in Set on the Two Algorithms
(a) Chang's Algorithm (b) Jaeschke's Algorithm

5.2 Space Complexity Analysis

5.2.1 Chang's Algorithm

Chang's algorithm uses space to store the input set and the abstracted pair set. The space used is O(n), and storage used to save C(x), p(x) and d(x) is also O(n). The storage to save b_i is less than O(n^2). So the space complexity of this algorithm is O(n^2).

5.2.2 Jaeschke's Algorithm

Jaeschke's algorithm uses O(n) storage to save the input set, I_n set, P_1 set and P_2 set which are O(n) for all. Thus the space complexity of this algorithm is O(n).

5.3 Machine Dependence

Both Chang's algorithm and Jaeschke's algorithm are machine-dependent if the input set is letter-oriented since different kinds of machines have different machine character code representations which are used to get appropriate values of the hash functions. Nowadays, most machines use the ASCII code for character that will make these algorithms depend less on machines.

For the numeric input set, Jaeschke's algorithm is machine independent, since the machine code never participate the calculation of the value of hash function. For Chang's method, we first should shift the input set into words set which make this method machine-dependent for the numeric input set. Or we can choose another approach also developed by Chang which uses a different formula as mentioned in [9], then the machine character code never be used for calculating the value of hash function, and this make Chang's algorithm machine-independent either.

5.4 Operation Time Comparison

After getting the perfect hash functions for the input set by using two algorithms, there comes the problem: Does the operation on the hash table established by those hash functions, such as searching and finding, consume same a mount of time?

Here is the analysis of this. We choose the hash table of the Month set which includes twelve slots. The test set is composed of each element of the Month set and other twelve words. The words are Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday, Day, Date, Year, Month and Week. The run time for finding if the element of the test set is in the hash table is shown in Figure 12.

From the figure, we can see that operation on the hash table built by Jaeschke's hashing is much faster than that of Chang's because the hashing value is easier to calculate for a given hash function by Jaeschke's algorithm than that of Chang's.

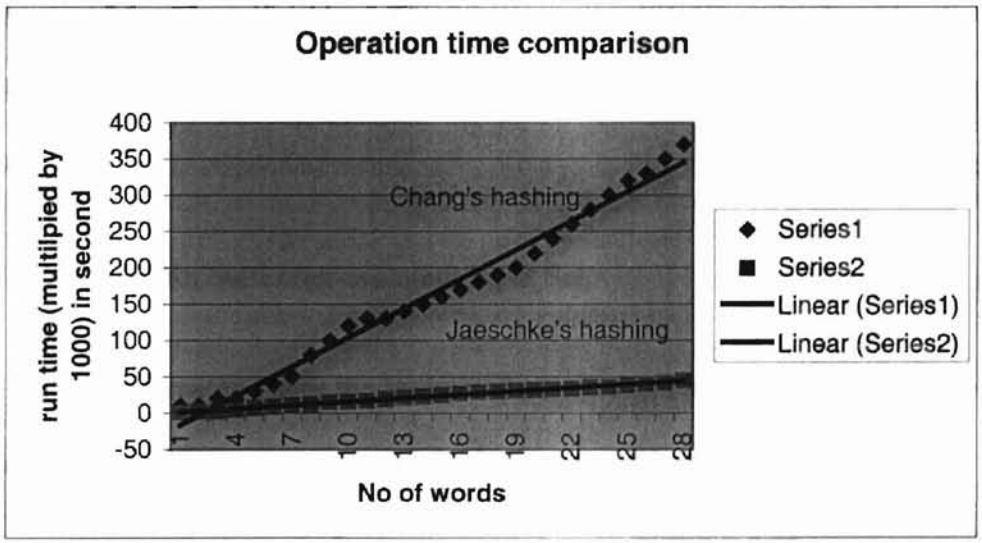


Figure 12 Operation Time Comparison on the Two Hash Tables Established by the Two Algorithms.

Chapter 6

CONCLUSIONS AND IMPROVEMENTS

From the last chapter, we can see that these two methods have their own time and space complexity. Each of them has their own advantages and disadvantages. Here are some discussions on them.

6.1 Advantages of Chang's Algorithm

6.1.1 Good Run Time Complexity

As shown in the last chapter this method has the time complexity of $O(n)$. It is also good for a large input set. For a large set (n>15), Chang's algorithm is a good choice since Chang's algorithm is $O(n^c)$, where c is less than 2, and Jaeschke's algorithm is $O(a^n)$.

6.1.2 Good for the Letter-oriented Input Set

This method uses the abstracted pair that is based on the input set to calculate the hashing value. It is very powerful for letter-oriented sets.

6.1.3 Easy Coding and High Efficiency

This method is based on the Chinese Remainder Theorem. It is easy to code, and the result is guaranteed to be a minimal perfect hashing function. So it is good to use this method to perform hashing.

6.2 Limitations of Chang's Method

Although Chang's method has good performance in run time for a letter-oriented input set, it has some disadvantages. Here are some of them.

6.2.1 Space Limitation

This algorithm's space complexity is $O(n^2)$, that means it will use a large space to hold the calculation values. If the input set is large, there will be a problem.

6.2.2 Distribution of the Input Set

This method uses the grouped abstracted pair (p_1, p_2). Each pair in the group has the same p_1, but different p_2. This means the maximum number of the pairs in the group must be less than 26. So if the input words is concentrated in a small range that will cause the run time to find the abstracted pair to go up, or even worse there will be no solution for this kind of set and some additional methods must be used to re-group the set.

6.2.3 Handles Only Letter-oriented Input Set Effectively

This method is powerful and fast only for coping with a letter-oriented set. For other kinds of sets there must be a different approach of Chang's method or it will need more complicated calculation to shift them to word sets.

6.3 Advantages of Jaeschke's Algorithm

6.3.1 Good Space Complexity

This method use only O(n) space for calculating hashing values, which saved more space than Chang's method.

6.3.2 No limitation for the input set

This method does not need the input to be words or letters. It can handle both characters and integers without much difference. Also the distribution of the input words has no effect on the run time of this method. Therefore, this method has a wide range of application than Chang's method.

6.3.3 Good Run Time for the Small Input Set

For a small set ($n \leq 15$), Jaeschke's algorithm is better than Chang's algorithm as we can see from the last chapter. It uses only as half time as that of Chang's algorithm on the average.

6.4 Disadvantages of Jaeschke's Algorithm

6.4.1 Run Time Complexity

This method is slower than Chang's method as n becomes large, because its run time is O(a^n). It is sometimes 100 times slower than Chang's method as number of elements in the set is greater than 30.

6.4.2 Run time Changes according to the Length of the Input Words or Value of the Input Data

If the input data are integers, the large input set will cause the run time go up. If the input set consists of words only, as the length of the input words goes up, it will cause the run time to increase as shown in the last chapter.

6.5 Suggestions

These two methods have their own advantages and disadvantages. It is better to use them properly and limit their drawbacks. Here are two suggestions

1) For the small input set use Jaeschke's algorithm, and for a large set use Chang's algorithm.

2) If the input set is concentrated in a small range of characters, first use Jaeschke's algorithm. If the length of the words is greater than seven characters, try Chang's algorithm first.

6.6 Improvements

There are also some methods to combine them and make them perform well.

1) For the concentrated word set, separate the input set into several groups and make sure each group has less than 15 words, then use Jaeschke's algorithm. In this way the only extra overhead is another storage to save the group table and increased run time because of calculating values for each group. Since they are all $O(n)$, it is not a big issue. So compare with the run time and space complexity of each method, it is still a

reasonable approach to separate the input set into subsets and use the Jaeschke's algorithm to hash each subset separately, then united the results together to get the hashing values.

2) For longer length word set or large integer set

First use Chang's method to get the abstracted pair set and then use Jaeschke's algorithm for longer length input words or just use some bits of input integers to build a new input set. In this way the calculation will be easy and simple and the extra time and space cost is only $O(n)$.

BIBLIOGRAPHY

[1] C. C. Chang, A letter-oriented minimal perfect hashing scheme. *The Computer J.*, 29(3), 277-281 (1986).

[2] R. J. Cichelli, Minimal perfect hash function made simple. *Commun. ACM,* 23(1), 17-19 (1980).

[3] V. G. Winters, Minimal perfect hashing in polynomial time. *Bit* 30, 235-244(1990).

[4] M. Gori amd G. Soda, An algebraic approach to Cichelli's perfect hashing. *BIT* 29, 2-13 (1989).

[5] G. Jaeschke. Reciprocal hashing: a method for generating minimal perfect hashing functions. *Commun. ACM*, 24(12), 829-833 (1981).

[6] N. Cercon, J. Boates and M. Krause, An interactive system for finding perfect hash functions. *IEEE Software,* 2(6), 38-53 (1985).

[7] F. Berman, M. E. Bock, et al. Collections of functions for perfect hashing. *SIAM J. Comput.,*15(2), 604-618 (1986).

[8] R. Sprugnoli, Perfect hashing functions: a single probe retrieving method for static sets. *Commun. ACM*, 20(11), 841-850 (1977).

[9] C. C. Chang, The study of an ordered minimal perfect hashing scheme. *Commun. ACM,* 27(4), 384-387 (1984).

[10] T. J. Sager, A polynomial time generator for minimal perfect hash functions. *Commun. ACM,* 28(5), 523-532 (1985).

[11] G. V. Cormack, R. N.S. Horspool and M. Kaiserswerth, Practical perfect hashing. *The Computer J.*, 28(1), 54-58 (1985).

[12] Z. J. Czech and B. S. Majewski, A linear time algorithm for finding minimal perfect hashing functions. *The Computer J., 36*(6), 579-587 (1993).

[13] C. C. Chang, An ordered minimal perfect hashing scheme based upon Euler's theorem. *Information Sciences,* 32(3), 165-172 (1984).

[14] C. R. Cook, A letter oriented minimal perfect hashing function. *Sigplan Notices,* 17(9),18-27 (1982).

[15] M. W. Du, K. F. Jea and D. W. Shieh, The study of a new perfect hash scheme. *Proceedings, the IEEE Computer Societies: International Computer Software & Application Conference'80, Chicago,* 341-347 (1980).

[16] G. Jaesche and G. Osterburg, On Cichelli's minimal perfect hash functions method. *Communications of the association for computing Machinery*, 23(12), 728-729 (1981).

[17] D. E. Knuth, *The Art of Computing Programming*. Vol. 3: Sorting and Searching. Addison-Wesley, Reading, Mass., 506-507 (1973).

[18] M. L. Fredman, J. Konlos and E. Szemeredi, Storing a sparse table with O(1) worst case access time. *Joural ACM*, 31(3), 538-544(1984).

[19] C. Bell and B. Floyd, A Monte Carlo study of Cichelli of hash-junction solvability. *Commun. ACM,* 26(11), 924-925 (1983).

[20] C. C. Chang, The study of an ordered minimal perfect hashing scheme with single parameter. *Information Processing Letters,* 27,79-83 (1988).

[21] W. P. Yang and M. W. Du, A backtracking method for constructing perfect hash functions from a set of mapping functions. *BIT*, 25, 148-164 (1985).

[22] G. D. Knott, Hashing functions. The *Computer J.*, 18, 265-278 (1975).

[23] C. Cook and R. Oldehoeft, More on minimal and almost minimal perfect hash function search. *Computers and Mathematics with Applications*. 9(1), 215-232 (1983).

[24] D.E. Knuth, Estimating the efficiency of backtrack programs. *Math. Comput.*,29(2), 121-136 (1975).

[25] J. L. Carter and M. N. Wegman, Universal classes of hash functions. *Proc. Ninth Annual Symposium on the Theory of Computing*, 106-112 (1977).

[26] D. Comer and M. J. O'Donnell, Geometric problems with application to hashing, *The Computer Journal*, 11,217-226 (1982).

[27] K. Mehlhorn, On the program size of perfect and universal hash functions. *Proc. 23rd Annual Symposium on the Foundations of Computer Science*. 170-175 (1982).

[28] R. E. Tarjan and A. C. Yao, Storing a sparse table. *Commun. ACM*, 22, 606-611 (1979).

[29] A. C. Yao, Should tables be sorted? *J. Assoc. Comput. Mach.*, 28, 615-628 (1981).

[30] W. D. Maurer and T. G. Lewis, Hash table methods. *Computing Surveys*. 7(1), 5-20 (1975).

[31] D. G. Severance, Identifier search mechanisms: A survey and generalized model. . *Computing Surveys*, 6(3), 175-194 (1974).

[32] C. C. Chang, The study of an ordered minimal perfect hashing scheme. *Commun. ACM*, 27(4), 384-387 (1984).

[33] B. Bollobas, *Random Graphs*. Academic Press, New York (1985).

[34] M. D. Brian and A. L. Tharp, Near-perfect hashing of large word sets. *Software—Practice and Experience,* 19, 967-978 (1990).

[35] Z. J. Czech, G. Havas and B. S. Majewski., An optimal alogrithm for generating minimal perfect hash functions. *Information Process. Lett.,* 43, 257-264 (1992).

[36] J. Ebert, A versatile data sturcture for edge-oriented graph algorithms. *Commun. ACM,* 30, 513-519 (1987).

[37] P. Flajolet, D. E. Knuth and B. Pittel, The first cycles in an evolving graph. *Discrete Mathematics,* 75, 167-215 (1989).

[38] E. A. Fox, L. S. Heath, et al., Practical minimal perfect has functions for large datbases. *Commun. ACM,* 35, 105-121 (1992).

[39] G. Haggard and K. Karplus, Finding minimal perfect hash functions. *ACM Special Interest Group on Individual Computing Environments Bull.,* 18, 191-193 (1986).

[40] G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures.* Addison-Wesley, Reading, *MA* (1991).

[41] T. G. Lewis and C. R. Cook, Hashing for dynamic and static internal tables. *IEEE Computer, 21, 45-56* (1988).

[42] E. M. Palmer. Graphical Evolution: *An Introduction to the Theory of Random Graphs.* New York, John Wiley & Sons (1985).

[43] M. A. Weiss, *Data Structures and Algorithm Analysis in C.* Menlo Park, Addison-Wesley Publishing Company (1997).

[44] B. S. Majewski, N. C. Worwald, etal, A family of perfect hashing methods. *The Computer J.,* 39(6), 547-554 (1996).

[45] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*. McGraw-HillCompanies (1997).

APPENDIX A C PROGRAMMING CODE FOR CHANG'S ALGORITHM

```
#include<stdio.h>
#include<string.h>
#include<math.h>
#include<time.h>

/*   Group Structure */
typedef struct group
{
      char G;
      int Pos[2];/* Pos[0] stands for pos, Pos[1] stands for length*/
      int size;
      int link[30];
      struct group *next;
}group;

/*  Abstracted Pair */
typedef struct pair
{
      char k1;
      char k2;
      int hx;
}pair;

/* d(x) and c(x) of input element */
typedef struct d_c
{     char k1;
      int d;
      int c;
}d_c;

/* Frequency of each character */
typedef struct fre{
      char k2;
      int n;
      int p;
}fre;
char tempy[30],input[50][10];/* input should be separated by comma.*/
pair x[32];
d_c dx_cx[26];
fre f[26]={{'0',0,0}};
int kk[26],size;
/* Functions */
group * sort_group(group *Group);
void get_abstract(group *Group);
int get_ab(group *node,int sign);
int  get_group(int group[]);
void get_dx(d_c dx_cx[], int group[],int g_Num);
int  get_fx(fre f[]);
void get_prime(int f_Num,int Prime[]);
```

```
void get_px(fre f[],int Prime[],int f_Num);
void print(group *Group,FILE *out);
void print_dx(FILE *out);
void print_px(fre f[],FILE *out);
int get_ff(char c);
int get_c(int kk[],int size);
void get_cx(group *Group);
void print_cx(FILE *out);
int c(char k1);
int d(char k1);
int p(char k2);
void cal_hx(FILE *out,group *Group);
void sort(int size);

void main(void){
      int i,j,k,n,loop;
      char temp;
      group *Group,*node;
      int g_Num,grou[26],f_Num;
      int d[26],Prime[26];
      FILE *out,*tout;
      time_t start,end,usetime;

      i=1;
      j=0;

      out=fopen("aa","w");
      tout=fopen("time","w");
      start=time(NULL);

      for(n=0;n<500;n++){
            fprintf(out,"********    n= %2d  **************\n\n",n);
            Group=(group *)malloc(sizeof(group));
            Group->next=(group *)malloc(sizeof (group));
            node=Group->next;
            node->size=0;
            node->next=NULL;
            for(i=0;i<50;i++){
                  input[i][0]='\0';
            }
            /*get input set and store it in char array temp */
            printf("Please in put the letter set:\n");
            printf("********************************\n");
            usetime=0;
            /* data set are stored in input array */
            temp=getchar();
            i=1;
            j=0;
            while (temp!='\n'){
                  if(temp!=','){
                        input[i][j++]=temp;
                  }
                  else{
                        input[i][j]='\0';
                        j=0;
                        i++;
                  }
```

```
                temp=getchar();
            }
            size=i;
            input[i++][j]='\0';

            input[i++][0]=0;/* show the input is finished */

            /*sort the set and group them */
                        Group=sort_group(Group);
            get_abstract(Group);
            print(Group,out);

            g_Num=get_group(grou);
            get_dx(dx_cx,grou,g_Num);
            print_dx(out);
            f_Num=get_fx(f);
            get_prime(f_Num,Prime);
            get_px(f,Prime,f_Num);
            print_px(f,out);
            get_cx(Group);
            print_cx(out);
            cal_hx(out,Group);
            end+=time(NULL);
            usetime=end-start;
            fprintf(tout,"-----------------------------------------------
--\n");
            fprintf(tout,"words number=%10d,  time
used=%10d\n",size,usetime);

            fprintf(out,"-----------------------------------------------
------\n");
                  free(Group);

      }
      fclose(out);
      fclose(tout);


}


void sort(int size){
      int i,j,temp;
      for(i=0;i<size-1;i++){
            for(j=size-1;j>i;--j){
                  if(kk[j-1]>kk[j]){
                        temp=kk[j-1];
                        kk[j-1]=kk[j];
                        kk[j]=temp;
                  }
            }
      }
}


void cal_hx(FILE *out,group *Group){
      group *node;
```

```
	int i,k,j;
	i=0;
	while(x[i].k1!=0){
		x[i].hx=d(x[i].k1)+(c(x[i].k1)%p(x[i].k2));

		i++;
	}

	fprintf(out,"          Location            Extracted Pair
Original Key     \n");
	fprintf(out,"-----------------------------------------------------
----------------------\n");
	i=1;
	j=0;
	k=0;
	node=Group->next;
	while(node!=NULL){
		k=0;
		fprintf(out,"%13d                    (%1c,%1c)
%20s\n",x[j].hx,x[j].k1,x[j++].k2,input[node->link[k++]]);
		while(k<node->size){
			fprintf(out,"%13d                    (%1c,%1c)
%20s\n",x[j].hx,x[j].k1,x[j++].k2,input[node->link[k++]]);
		}
		node=node->next;
	}

}
int d(char k1){
	int i=0;
	while(dx_cx[i].k1!=0){
		if(dx_cx[i].k1==k1)
			return dx_cx[i].d;
		else
			i++;
	}
	return 0;

}
int c(char k1){
	int i=0;
	while(dx_cx[i].k1!=0){
		if(dx_cx[i].k1==k1)
			return dx_cx[i].c;
		else
			i++;
	}
	return 0;

}
int p(char k2){
	int i=0;
	while(f[i].k2!=0){
		if(f[i].k2==k2)
			return f[i].p;
		else
			i++;
```

```
        }
        return 0;

}


void get_cx(group *Group){
        int i,k,j,z;

        group *node=Group->next;
        k=z=0;
        while(node!=NULL){
            if(node->size!=1){
                        i=node->size;
                        j=0;
                        while(i>0){
                            kk[j]=get_ff(x[k].k2);
                                        j++;
                                        k++;
                            i--;
                        }
                                kk[j]=0;
                                sort(node->size);
                        dx_cx[z++].c=get_c(kk,node->size);
                    }
                    else{
                          dx_cx[z++].c=1;
                          k++;
                    }
                    node=node->next;
            }
}

int get_ff(char c){
      int i=0;
      while(f[i].k2!='0'&&f[i].k2!=c){
            i++;
      }
      return f[i].p;
}
int get_c(int kk[],int size){


      int i,j,z,k;
      int *MM,DEND,RMD,DSR,*Q,*B,*b,*C,*M,c,m;
      int *y;
      M=(int *)malloc(sizeof(int)*size);
      MM=(int *)malloc(sizeof(int)*size);
      Q=(int *)malloc(sizeof(int)*size);
      B=(int *)malloc(sizeof(int)*size);
      b=(int *)malloc(sizeof(int)*size);
      C=(int *)malloc(sizeof(int)*size);
      y=(int *)malloc(sizeof(int)*size);
      for(i=1;i<=size;i++){
            M[i]=1;
            MM[i]=Q[i]=B[i]=b[i]=C[i]=0;
      }
```

```
for(i=1;i<=size;i++){
	for(j=1;j<=size;j++){
		if(i!=j)
			M[i] *=kk[j-1];
	}
}
for (i=1;i<=size;i++){
	MM[i]=M[i]%kk[i-1];
//	printf("%d\n",MM[i]);
}
for(z=1;z<=size;z++){
	DEND=kk[z-1];
	DSR=MM[z];
	j=1;
	Q[j]=DEND/DSR;
	RMD=DEND-Q[j]*DSR;
	if(RMD==0){
		b[z]=1;
	}
	else{
		while(RMD!=0&&RMD!=1){
			DEND=DSR;
			DSR=RMD;
			j=j+1;
			Q[j]=DEND/DSR;
			RMD=DEND-Q[j]*DSR;
		}
		i=j;
		B[0]=1;
		B[1]=-Q[i];
		if(i>2){
			for (j=1;j=i-1;j++){
				B[j+1]=-B[j]*Q[i-j]+B[j-1];
			}
		}
		else {
			if(i==2)
				B[j]=-B[j-1]*Q[i-j+1]+B[j-2];
		}
		b[z]=B[i];
	}

}
c=0;
m=1;
for(i=1;i<=size;i++){
	C[i]=b[i]*M[i]*i;
    c+=C[i];
	m*=kk[i-1];
}
if(c<0){
	c+=m;
}
c=c%m;
free(M);
free(MM);
```

```
        free(Q);
        free(b);
        free(B);
        free(C);
        free(y);

        return c;

}




void print_px(fre f[],FILE *out){
        int i;
        fprintf(out,"----------------------------\n");

        i=0;
        while(f[i].k2!=0){
                fprintf(out,"p[%c]= %2d\n",f[i].k2,f[i].p);
                i++;
        }
        fprintf(out,"----------------------------\n");
}
void print_dx(FILE *out){
        int i,j;
        fprintf(out,"----------------------------\n");
        fprintf(out,"d[%c] = %2d\n",x[0].k1,dx_cx[0].d);
        i=1;
        j=1;
        while(x[i].k1!=0){
                if(x[i].k1==x[i-1].k1)
                        i++;
                else{

                        fprintf(out,"d[%c] = %2d",x[i].k1,dx_cx[j++].d);
                fprintf(out,"\n");
                                i++;
                }
        }

    fprintf(out,"----------------------------\n");

}

void print_cx(FILE *out){
        int i,j;
        fprintf(out,"----------------------------\n");
        fprintf(out,"c[%c] = %2d\n",x[0].k1,dx_cx[0].c);
        i=1;
        j=1;
        while(x[i].k1!=0){
                if(x[i].k1==x[i-1].k1)
                        i++;
                else{
```

```
                fprintf(out,"c[%c] = %2d",x[i].k1,dx_cx[j++].c);
            fprintf(out,"\n");
                i++;
            }
        }

    fprintf(out,"---------------------------\n");

}

group *sort_group(group *Group){
    int i,k;
    group *node,*pnode;
    i=1;
    k=0;/* for link position */

    while(input[i][0]!=0){
        node=Group->next;
        if(i==1){/*first node*/
            node->G=input[i][0];
            node->link[0]=i;
            node->link[1]='\0';
            node->size=1;
        }
        else{

            if(node->G==input[i][0]){
                k=0;
                while(node->link[k]!='\0'){
                    k++;
                }
                node->link[k]=i;
                node->link[k+1]='\0';
               node->size++;
            }
          else{
                if(node->G>input[i][0]){
                    Group->next=(group
*)malloc(sizeof(group));
                    Group->next->G=input[i][0];
                    Group->next->link[0]=i;
                    Group->next->link[1]='\0';
                    Group->next->size=1;
                    Group->next->next=node;
                }
                else{
                    pnode=node;
                    while(node->G<input[i][0]&&node-
>next!=NULL){
                        pnode=node;
                        node=node->next;
                    }
                    if(node->G==input[i][0]){
                        k=0;
                        while(node->link[k]!='\0'){
                        k++;
```

```
                                    }
                                    node->link[k]=i;
                                    node->link[k+1]='\0';
                                    node->size++;
                              }
                              else{
                                    if(node->G>input[i][0]){
                                          pnode->next=(group
*)malloc(sizeof(group));
                                          pnode->next->G=input[i][0];
                                          pnode->next->link[0]=i;
                                          pnode->next->link[1]='\0';
                                          pnode->next->size=1;
                                          pnode->next->next=node;
                                    }
                                    else{
                                          node->next=(group
*)malloc(sizeof(group));
                                          node->next->G=input[i][0];
                                          node->next->link[0]=i;
                                          node->next->link[1]='\0';
                                          node->next->size=1;
                                          node->next->next=NULL;
                                    }
                              }

                        }
                  }
            }
            i++;
      }
      return Group;
}
void get_abstract(group *Group){
      group *node=Group->next;
      int k,j,z,i=0;
      j=0;
      while(node!=NULL){
            if(node->size==1){
                  node->Pos[0]=2;
                  node->Pos[1]=-1;
                  x[i].k1=node->G;
                  k=node->link[0];
                        x[i++].k2=input[k][1];
            }
            else{
                  j=get_ab(node,1);
                  if(j!=-1){
                        z=0;
                        while(z<node->size){
                              node->Pos[0]=j;
                              node->Pos[1]=-1;
                              x[i].k1=node->G;
                              k=node->link[z];
                              x[i++].k2=input[k][j];
                              z++;
                        }
```

```
                }
            }
            node=node->next;
        }
        x[i].k1='\0';
        x[i].k2='\0';
}
int get_ab(group *node,int sign){
        int i,j,k,z;
        k=0;
        if(sign==10)
            return -1;/*feilture */
        for(i=0;i<30;i++){
                tempy[i]='\0';
        }
        if(strlen(input[node->link[0]])>sign){
            tempy[k++]=input[node->link[0]][sign];
        }
        else{
            z=strlen(input[node->link[0]]);
            tempy[k++]=input[node->link[0]][z];
        }
        tempy[k]='\0';
        j=1;
        i=0;
        while(j<node->size){
            if(!strchr(tempy,input[node->link[j]][sign])){
                if(strlen(input[node->link[j]])>sign){
                    tempy[k++]=input[node->link[j]][sign];
                }
                else{
                    z=strlen(input[node->link[j]]);
                    tempy[k++]=input[node->link[j]][z];
                }
            }
            else{
            i=-1;
                break;
            }
            j++;
        }
        if(i!=-1){
            return sign;
        }
        else{
            get_ab(node,sign+1);
        }

}


void print(group *Group,FILE *out){
        group *node;
        int j,k,i=1;
        node=Group->next;
        j=k=0;
```

```
      fprintf(out,"**********************************************************
**********************\n");
      fprintf(out,"\n Input set is: \n\n");
      while(i<=size){
            fprintf(out,"%20s\n",input[i++]);
      }
      fprintf(out,"**********************************************************
**********************\n");

      fprintf(out,"          Group            Extracted Pair
Original Key     \n");
      fprintf(out,"----------------------------------------------------------
----------------------\n");
      i=1;
      while(node!=NULL){
            k=0;
            fprintf(out,"%13d                    (%1c,%1c)
%20s\n",i++,x[j].k1,x[j++].k2,input[node->link[k++]]);
            while(k<node->size){
                  fprintf(out,"
(%1c,%1c)     %20s\n",x[j].k1,x[j++].k2,input[node->link[k++]]);
            }
            node=node->next;
      }
      fprintf(out,"----------------------------------------------------------
----------------------\n");
}
void get_px(fre f[],int Prime[],int f_Num){
      int i,k,z=0;

      for(i=0;i<f_Num;i++){
            f[i].p=-1;
      }
      for(i=0;i<f_Num;i++){

            z=0;
            while(f[z].p!=-1){
                  z++;
            }
            for(k=z+1;k<f_Num;k++){
                  if(f[k].p==-1
&&((f[k].n>f[z].n)||(f[k].n==f[z].n&&(f[k].k2<f[z].k2))))
                        z=k;
            }
            f[z].p=Prime[i];
      }
}

void get_prime(int f_Num, int Prime[]){
      int i,k,z;
      Prime[0]=2;
    Prime[1]=3;
      Prime[2]=5;
      k=3;
      for(i=7;k<f_Num;i++){
            for(z=sqrt(i);z>1;z--){
                  if(i%z==0)
```

```
                        z=1;
                }
                if(z==1){
                        Prime[k++]=i;
                }
        }
}


int  get_fx(fre f[]){
        int i,z,k;
        f[0].n=1;
        f[0].k2=x[0].k2;
        k=1;
        i=1;
        while(x[i].k2!=0){
                for(z=k-1;z>=0;z--){
                        if(f[z].k2==x[i].k2){
                                f[z].n++;
                                z=-1;
                        }
                }
                /* if k2 first appearence */
                if(z!=-2){
                        f[k].n=1;
                        f[k++].k2=x[i].k2;
                }
                i++;
        }
        return k;
}



void get_dx(d_c dx_cx[], int grou[],int g_Num){
        int i,k,z;
        for(i=0;i<26;i++){
                dx_cx[i].d=0;
                dx_cx[i].c=0;
        }
        i=1;
        k=1;
        dx_cx[0].k1=x[0].k1;
        while(x[i].k1!=0){
                if(x[i].k1!=x[i-1].k1){
                        dx_cx[k++].k1=x[i].k1;
                }
                i++;
        }
        while(x[i].k1!=0){
                for(z=k-1;z>=0;z--){
                        if(dx_cx[z].k1==x[i].k1){
                          f[z].n++;
                          z=-1;
                        }
                }
                /* if k2 first appearence */
```

```
            if(z!=-2){
                  f[k].n=1;
                  f[k++].k2=x[i].k2;
            }
            i++;
      }
      for(i=1;i<g_Num;i++){

            dx_cx[i].d=dx_cx[i-1].d+grou[i-1];
      }
}
int get_group(int grou[]){
      int i,j;
      for(i=0;i<26;i++){
            grou[i]=0;
      }
      grou[0]=1;
      for(j=1,i=0;j<32;j++){
            if(x[j].k1==x[j-1].k1){
                  grou[i]++;
            }
            else{
                  i++;
                  grou[i]=1;
            }
      }
      return i+1;
}
```

APPENDIX B C PROGRAMMING CODE FOR JAESCHKE''S ALGORITHM

```
#include<stdio.h>
#include<string.h>
#include<math.h>

char input[50][10];
int w[50],ww[50],B[50];
long D,E,bound;
int Prime[50],P1[50],P2[50],M[50],P[50],P_temp[50];
int get_prime(int num);
void calcu_D(int num);
void calcu_P1(int num);
void calcu_B(int num,int size);
int Check_B(int size);
int multiple(int size);
void sort(int size);
int calcu_C(int size,long C,long L);
int check_same(int size,int same[]);
void printout(FILE *out,int size,int C);
void calcu_M(int num);
void calcu_bound(int num);
void calcu_E(int num);
int get_I(int num);
int get_T(void);
int i0,a,T[50];
void main(void){
                    int i,j,k,size,num;
                    long L,C,C0;
                    char temp;
                    FILE *out;
                    time_t start,end,usetime;

                    out=fopen("bb","w");

                    for(i=0;i<50;i++){
                          input[i][0]='\0';
                          w[i]=ww[i]=0;
                    }
                    printf("Please in put the letter set:\n");
                    printf("*********************************\n");

                    /* data set are stored in input array */
                    i=0;
                    j=0;
                    temp=getchar();
                    while (temp!='\n'){
                          if(temp!=','){
                                input[i][j++]=temp;
                                w[i]+=temp;
                                ww[i]+=temp;
                          }
                          else{
```

```
                        input[i][j]='\0';
                        j=0;
                        i++;
                    }
                    temp=getchar();
                }
                size=i+1;

                input[i++][j]='\0';
                w[i]=0;
                ww[i];
                input[i][0]=0;/* show the input is finished */
                start=time(NULL);

        for(i=0;i<500;i++){
                        sort(size);
                C0=(size-2)*w[0]*w[size-1]/(w[size-1]-w[0]);
         L=multiple(size);
         L*=size;
         j=calcu_C(size,C0,L);
         if(j!=-1){
            printout(out,size,j);
         }
         else{
            get_prime(size);
            num=get_I(size/2);
            for(i=0;i<num;i++)
                    P_temp[i]=P[i];
            calcu_B(num,size);
                     calcu_P1(num);
            calcu_D(num);
            calcu_M(num);
            calcu_bound(num);
            i=get_T();
            calcu_E(i,num);
         }
end+=time(NULL);
                        usetime=end-start;
                        fprintf(out,"-----------------------------------------
                -\n");
                        fprintf(out,"words number=%10d,  time
                used=%10d\n",size,usetime);

                        fprintf(out,"-----------------------------------------
                ------\n");

    }
    fclose(out);

}

/*get set I of prime <= n/2 */
int get_I(int num){
                int i;
                for(i=0;Prime[i]<=num;i++)
                        P[i]=Prime[i];
                return i;
```

```
}
int get_T(void){
                    int i,j,temp;
                    i=sqrt(bound);
                    temp=bound;
                    j=get_I(i);
                    for(i=0;i<50;i++){
                          T[i]=0;
                    }
                    for(i=0;i<j;i++){
                          while(P[i]!=0&&temp%P[i]==0){
                                T[i]=P[i];
                                temp /=P[i];
                          }
                    }
                    return j;
}



void calcu_E(int j,int num){
                    int i,flag;
                    long temp=1;
                    for(i=0;i<j;i++){
                          if(T[i]!=0)
                                temp *=T[i];
                          if(P_temp[i]!=0)
                                temp *= P_temp[i];
                    }
                    for(E=P[0];E<temp;E+=P[0]){
                          flag=0;
                          for(i=0;i<num&&flag!=1;i++){
                                if(P1[i]!=0 &&(E%P1[i])==0)
                                      flag=1;
                          }
                          if(flag!=1)
                                return;
                    }
}
void calcu_bound(int num){
                    int i,j;
                    long sq_d,deta=1;
                    for(i=0;i<num;i++){
                          for(j=i+1;j<num;j++){
                                deta *=(w[j]-w[i]);
                          }
                    }
                    bound=deta;

}


void calcu_M(int num){
                    int i;
                    for(i=0;i<num;i++){
                          M[i]=0;
                          if(P2[i]!=0)
```

```
                        M[i]=-D*(w[i]%P2[i])%P2[i];
                }
}

void calcu_D(int num){
                int i;
                D=1;
                for(i=0;i<num;i++){
                        if(P1[i]!=0)
                                D *=P1[i];
                }
}
/*get P1 set */
void calcu_P1(int num){
                int i,j=0;
                for(i=0;i<num;i++)
                        P1[i]=0;
                for(i=1;i<num;i++){
                        if(Prime[i]!=P2[i])
                                P1[j++]=Prime[i];
                }
}



void calcu_B(int num,int size){
                int i,j,z;
                for(i=0;i<50;i++){
                        B[i]=-1;
                        P2[i]=0;
                }
                for(j=0;j<num;j++){
                        for(i=0;i<size;i++){
                                /* get all resides in B*/
                                B[i]=w[i]%Prime[j];
                        }
                        z=Check_B(size);
                        if(z==1){
                                P2[j]=Prime[j];
                                /*if P2[j]!=0 that is element of P2 set*/
                        }
                }
}
/*check the times of v happens */
int Check_B(int size){
                int i,j,f;
                for (i=0;i<size;i++){
                        f=1;
                        for(j=i+1;j<size;j++){
                                if(B[i]==B[j])
                                        f++;
                        }
                        if(f==1)
                                return 1;
                }
                return 0;
}
```

```
int  get_prime(int size){
                int i,k,z;
                Prime[0]=2;
    Prime[1]=3;
                Prime[2]=5;
                k=3;
                for(i=7;i<size;i++){
                      for(z=sqrt(i);z>1;z--){
                            if(i%z==0)
                                  z=1;
                      }
                      if(z==1){
                            Prime[k++]=i;
                      }
                }
                for(i=k;i<50;i++){
                      Prime[i]=0;
                }
                return k;
}
void printout(FILE *out,int size,int C){
                int i;
                fprintf(out,"Input set is:\n");
                fprintf(out,"---------------------------\n");
                fprintf(out,"     word           w[i]\n");
                fprintf(out,"---------------------------\n");
                for(i=0;i<size;i++){
                      fprintf(out,"%10s %10d\n",input[i],w[i]);
                }
                fprintf(out,"\nC=%6d\n\n",C);
                fprintf(out,"-------------------------\n");
                fprintf(out,"     Location       word\n");
                fprintf(out,"---------------------------\n");
                for(i=0;i<size;i++){
                fprintf(out,"%10d %12s\n",(C/w[i])%size,input[i]);
                }
}


int calcu_C(int size,long C,long L){
                int i,j,same[50];
                if(C>L){
                      return -1;
                }
    for(i=0;i<50;i++){
                      same[i]=0;
                }
                for(i=0;i<size;i++){
                      same[i]=(C/w[i])%size;
                }
                j=check_same(size,same);
```

```
                if(j==-1)
                        return C;
                else{
                        i0=0;
                        for(i=0;i<j;i++){
                                if(((C/w[j])%size==(C/w[i])%size)&&i0<i)
                                        i0=i;
                        }
                        if((w[i0]-C%w[i0])>(w[j]-C%w[j]))
                                a=w[j]-C%w[j];
                        else
                                a=w[i0]-C%w[i0];
                        C=calcu_C(size,C+a,L);
                        return C;
                }

}
int check_same(int size,int same[]){
                int i,j,jj;
                jj=-1;
                for(i=0;i<size;i++){
                        for(j=i+1;j<size;j++){
                                if((same[i]==same[j])&&j>jj)
                                        jj=j;
                        }
                }
                return jj;
}



int multiple(int size){
                int i,j,flag;
                long L,m;

                int temp[50];
                m=L=1;
                for(j=0;j<size;j++){
                        temp[j]=w[j];
                }
                for(i=2;i<sqrt(w[size-1]);i++){
                        flag=0;
                        for(j=0;j<size;j++){
                                if((temp[j]-(temp[j]/i)*i)==0){
                                        temp[j]=temp[j]/i;
                                        if(flag==0){
                                                flag=1;
                                                m*=i;
                                        }
                                }

                        }
                }
                for(i=0;i<size;i++)
                        L*=temp[i];
                return L*m;
```

```
}

void sort(int size){
                    int i,j,temp;
                    for(i=0;i<size-1;i++){
                          for(j=size-1;j>i;--j){
                                if(w[j-1]>w[j]){
                                      temp=w[j-1];
                                      w[j-1]=w[j];
                                      w[j]=temp;
                                }
                          }
                    }
}
```

VITA

Qizhi Tao

Candidate for the Degree of

Master of Science

Thesis: COMPARISON OF PERFECT HASHING METHODS

Major Field: Computer Science

Biographical:

Personal Data: Born in Harbin, Heilongjiang Province of P R China, the daughter of Chongde Tao and Aihua Zhou.

Education: Graduate from the No. 3 middle school of Harbin in July 1984; received Bachelor of Science degree and Master of Science degree in Mechanical Engineering from Harbin Institute of Technology in July 1988 and March 1991, respectively. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in June, 1999.

Experience: Worked as an engineer and translator in Longda Company of Harbin from 1991 to 1993;employed as an executive editor of Journal of Harbin Institute of Technology (English Edition) for 1993 to 1997.