Application of Minimal Perfect Hashing in Main Memory Indexing

by

Yuk Ho

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

Author ....
Department of Electrical Engineering and Computer Science
May 20, 1994

Certified by ....
Jerome H. Saltzer
Thesis Supervisor

Accepted by ....
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

Application of Minimal Perfect Hashing in Main Memory Indexing

by

Yuk Ho

Submitted to the
Department of Electrical Engineering and Computer Science

May 20, 1992

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering and
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

With the rapid decrease in the cost of random access memory (RAM), it will soon become economically feasible to place full-text indexes of a library in main memory. One essential component of the indexing system is a hashing algorithm, which maps a keyword into the memory address of the index information corresponding to that keyword. This thesis studies the application of the minimal perfect hashing algorithm in main memory indexing. This algorithm is integrated into the index search engine of the Library 2000 system, a digital on-line library system. The performance of this algorithm is compared with that of the open-addressing hashing scheme. We find that although the minimal perfect hashing algorithm needs fewer keyword comparisons per keyword search on average, its hashing performance is slower than the open-addressing scheme.

Thesis Supervisor: Jerome H. Saltzer
Title: Professor, Department of Electrical Engineering and Computer Science

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank:

Professor Jerome Saltzer, my thesis supervisor, for his guidance and support to my research work.

Mitchell Charity, for teaching me everything I need to know about the Library 2000 system.

George Havas and Bohdan Majewski, for giving me access to the source code of their minimal perfect hashing algorithm implementation.

My parents, for their continuous support during my study at M.I.T.

Albert, Ali, Bernard, Esther and Jerome, for helping me to procrastinate my thesis work.

This work was supported in part by the IBM Corporation, in part by the Digital Equipment Corporation, and in part by the Corporation for National Research Initiatives, using funds from the Advanced Research Projects Agency of the United States Department of Defense under grant MDA972-92-J1029.

Chapter 1

Introduction

1.1 Library 2000 Project Overview

The research work on this thesis is a part of the Library 2000 project. As stated in the 1993 annual progress report of the project [2], “the basic hypothesis of the project is that the technology of on-line storage, display, and communications will, by the year 2000, make it economically possible to place the entire contents of a library on-line and accessible from computer workstations located anywhere. The goal of the Library 2000 project is to understand and try out the system engineering required to fabricate such a future library system.”

The vision of the project is that the future library system will allow users to browse any book, paper, thesis, or technical report from a standard office desktop computer. While reading a document, one can follow citations and references by point-and-click, and the selected document will pop up immediately in an adjacent window [1].

1.2 Main Memory Indexing

One essential module of an on-line digital library system is the index search engine. With an extremely large amount of information stored in the library system, it is very important that we can access these information efficiently. The most commonly used form of information retrieval is keyword search. With a given keyword, the index search engine will find all the documents that contain the keyword by searching in a full-text index.

One approach of achieving efficient information retrieval in a library system is to store the full-text index in main memory rather than in the disk. Random access memory (RAM) offers a clear performance advantage: RAM access time is typically faster than disk access time by five orders of magnitude. Looking ahead a few years, one can anticipate that with the rapid decrease in the cost of RAM, main memory indexing will soon become an economically feasible solution even for indexes with a size of a few gigabytes. Therefore the Library 2000 project is exploring the implications and mechanisms of using indexes stored completely in RAM.

In the Library 2000 system, the index is generated from the bibliographic records of all the documents in the library. It is stored in a database called the wordset. The wordset resides in main memory when the search engine is in operation. The wordset contains an array of word-location pairs. Each word-location pair consists of two parts: a keyword and a set of locations corresponding to that keyword. Each location record identifies a document and a place in the document in which the given word is located. The search engine's task is to find the set of word locations for a given keyword, or report that the keyword is not in the wordset.

1.3 Index Search Strategies

In this section, we will discuss some common strategies used in doing keyword searches, and compare their performances.

1.3.1 Binary Search

This algorithm uses an array of records, with each record having two fields: a pointer to a keyword and a pointer to the keyword's location set. It assumes that the key-location pointer pairs in the array are sorted by the keywords.

When searching a keyword, we can check the midpoint of the array against the keyword and eliminate half of the array from further consideration. Then we repeat this procedure, halving the size of the remaining array each time [10]. This algorithm runs in $O(\lg n)$ time.

1.3.2 B-tree

In a B-tree, each node contains t key-location pointer pairs in sorted order. The node also contains t+1 pointer to its children. The keys in the node are used as dividing points separating the range of keys handled by the node into t+1 sub-ranges, each handled by one child of the node [10].

During a keyword search, we make a (t+1)-way branching decision at each node of the B-tree. The time bound of this algorithm is $O(\lg n)$, but the base of the logarithm can be many times larger. Therefore a B-tree saves a factor of lg t over binary search.

1.3.3 Hashing

A hashing algorithm maps a keyword directly into the memory address which contains the word locations corresponding to that keyword. If we have a perfect hash function, then each keyword search takes only one probe, and the search time bound is $O(1)$.

If the hash function is not perfect, then keyword searches will have the extra overhead of collision resolution. The number of probes needed for keyword searches using a linear-probing open-addressing scheme is approximately: [11]

$$C' \approx \frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right) \qquad \text{(unsuccessful search)} \tag{1.1}$$

$$C \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \qquad \text{(successful search)} \tag{1.2}$$

α is defined as the ratio $\frac{n}{m}$, where n is the number of keywords, and m is the number of entries in the hash table. By applying Taylor series expansion to the above equations at $\alpha = 0$, we can get the first order approximation of the number of probes for small α:

$$C' \approx 1 + \alpha = 1 + \frac{n}{m} \qquad \text{(unsuccessful search)} \tag{1.3}$$

$$C \approx 1 + \frac{\alpha}{2} = 1 + \frac{n}{2m} \qquad \text{(successful search)} \tag{1.4}$$

Therefore the search time bound is *O(n)*. But we can lower the constant factor of this bound by increasing *m*. For very small α, i.e., when $m \gg n$, the number of probes needed for keyword searches is close to 1. Therefore if we use a large enough hash table, this hashing scheme can still outperform binary search and B-trees even though it has a worse theoretical time bound.

From the above analysis, we can conclude that both perfect hashing and non-perfect hashing can offer a performance advantage over binary search and B-trees.

1.4 Hashing Algorithms

The keyword searching process consists of two steps:

1. *Hashing* — Map a keyword into the memory address which contains the word locations corresponding to that keyword.
2. *Retrieving* — Read the word-locations from the wordset.

The Library 2000 search engine uses an open-addressing hashing algorithm. The hash function of this algorithm may map two or more keywords into the same value, causing the problem called collision. This algorithm uses a linear-probing collision resolution scheme to ensure that each keyword is mapped to a unique memory address.

This thesis studies the application of another hashing algorithm — the minimal perfect hashing algorithm — in the main memory index search engine. The minimal perfect hashing algorithm uses a hash function that can map each keyword to a unique value. Therefore it does not have any collision resolution overhead.

The minimal perfect hashing algorithm will first be integrated into the search engine, replacing the open-addressing hashing module. Then the keyword search performance of these two hashing algorithms will be measured and compared.

1.5 Implementation Overview

The Library 2000 search engine is implemented in the C language, compiled on IBM RISC/6000 workstations. The minimal perfect hashing code is adapted from the C language implementation by Bohdan S. Majewski [3,4].

Currently the search engine runs on two sets of full-text indexes: one for the Reading Room of the MIT Laboratory for Computer Science and Artificial Intelligence Laboratory, the other for the MIT Barton Library System. The LCS/AI Reading Room index consists of 50,841 keywords, and the wordset size is 3.6 MB. The MIT Barton Library index consists of 1,140,389 keywords, and the wordset size is 146.1 MB. Most of the performance measurements of this thesis will be conducted on the MIT Barton Library index.

Chapter 2

The Library 2000 Index Structure

2.1 The Wordset

The wordset is the database that stores the full-text index of the bibliographic records of all the documents in the library. The wordset always resides in the main memory when the search engine is in operation.

To generate the wordset, we first extract all the wordpairs from the bibliographic records. A wordpair is a record that consists of two different fields: the keyword and the location field. In the Library 2000 search engine implementation, the location field is further divided into four sub-fields.

1. Record ID: The ID of the bibliographic record which contains the keyword.
2. Field ID: The field in which the keyword appears. The field can be title, author or others.
3. Field count: This number distinguishes multiple occurrences of a field in a record. First occurrence is 0, next is 1, etc.
4. Word position: Position of the keyword in the field. First word is 1, second is 2, etc.

In the next step, we first sort all the wordpairs by the keyword, and then by the record id, field id, field count and word position. Then we collapse all the wordpairs of each keyword into a single word-location pair. The word-location pair consists of a keyword, and a set of location records corresponding to that keyword. Each word location record has four fields: record id, field id, field count, and word position.

Now we can generate the wordset by dumping all the word-location pairs to a file. Since the wordpairs have been sorted by the keyword, the word-location pairs in the wordset are also in sorted order. The total size of the set of location records varies for different keywords.

In order to reduce the size of wordset, two compression schemes are used when generating the wordset. The set of location records corresponding to each keyword are encoded in variable length integers. Delta encoding is applied to the record ids within each set of location records.[1] The indexing mechanism, together with the compression schemes, reduce the 650 MB raw text source of the MIT Barton library catalog into a wordset index of 146 MB.

2.2 The Wordset Hash Table and the Wordstore

The index search engine uses two auxiliary data structures to improve the efficiency of information retrieval from the wordset: the wordset hash table and the wordstore. Figure 2.1 on page 12 illustrates these two data structures and their roles in the index search mechanism.

The wordstore is a contiguous piece of memory which contains all the keywords of the wordset in sorted order. The keywords are separated from each other by null characters. The wordset hash table is an array of fixed length records. Each record consists of four fields:

1. Word pointer: This pointer points to the first character of a keyword in the wordstore. The pointer is used for doing word comparisons during keyword searches. Word comparison is done by the C function “strcmp”. This function scans a word from its first character, and it treats the null character as the end of the word. The word pointer’s function will be described in detail in Chapter 3 and 4.

1. These compression schemes were designed and implemented by Mitchell Charity.

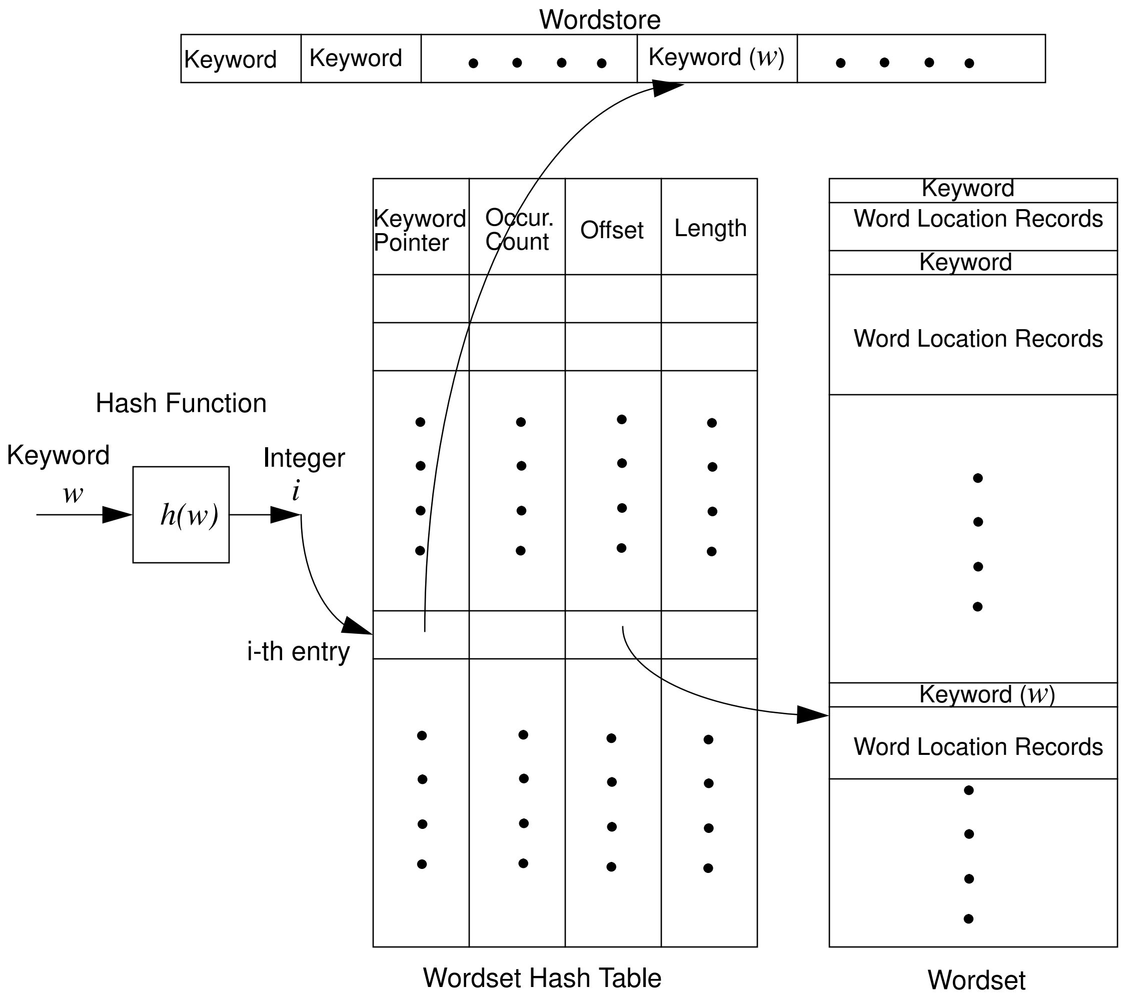

Figure 2.1: Data Structure of the Library 2000 Index Search Engine.

2. Occurrence count: The total number of occurrences of the keyword in all the bibliographic records.

3. Offset: The memory address offset of the set of location records corresponding to the keyword.

4. Length: The total number of bytes contained in the set of location records.

The order in which the records are stored in the hash table depends on the hash function. Consider a set **W** of m keywords, and a hash function h that maps **W** into some interval of integers **I**, say [0, k-1], where $k \geq m$. The wordset hash table will have k entries. For a particular keyword w, the record corresponding to that keyword will appear in the [$h(w)$]th entry of the hash table. If $k > m$, then some entries in the wordset hash table will be empty.

2.3 The Index Search Mechanism

When the index search engine starts up, it first loads the wordset into main memory. Then it looks for the wordset hash table file and the wordstore file. If these files do exist, they will be loaded into main memory.

However, if these two files do not exist, the search engine will generate them by scanning through the word-location pairs in the wordset. For each word-location pair, it copies the keyword into the wordstore. Then it passes the keyword to the hash function, which returns an integer, say i. A new hash table record is created and put into the i-th entry of the wordset hash table. The keyword pointer field of the record is set to point to the keyword in the wordstore. The occurrence count, offset and length fields are calculated from the location records in the wordset.

On doing a keyword search, the search engine first calls the hash function to map the keyword into an integer i. Then it fetches the i-th entry in the wordset hash table. The keyword pointer in that entry, which points to a word in the wordstore, is passed to the string comparison function. It compares the wordstore entry with the keyword being searched. This is necessary for collision detection if the hash function is not perfect. Even when we are using a perfect hash function, the string comparison is still needed, since it is possible for a word which is not a member of the index keyword set to be mapped to a number corresponding to another word which is in the keyword set.

Using the offset field of the hash table entry, we can calculate the memory address of the location records corresponding to the keyword. The length field tells the search engine the amount of data it should read from the wordset. The location records will then be decompressed and returned to the user.

The wordset generation process and the index search engine was designed and implemented by Mitchell Charity. Robert Miller built the part of the system which allowed sharing of a single search engine process by multiple clients. Manish Muzumdar had also contributed to the search engine implementation.

Chapter 3

The Open-addressing Hashing Algorithm

3.1 Implementation

The current implementation of the Library 2000 search engine uses a hashing algorithm called open-addressing. This algorithm was implemented by Mitchell Charity. Hashing a keyword set usually involves two steps. The first step is to map the keyword to an integer using a hash function. In some cases, the hash function may map two or more keys into the same number. This calls for the need of the second step — collision resolution. Open addressing is one of the common collision resolution schemes being employed. It uses an offset rule to find empty cells within the hash table [5].

3.1.1 Hash Function

The hash function is a function that maps a keyword into a hash table index. The function does this in two steps. It first converts the alphanumeric keyword into an integer, and then maps the integer into a hash table index. In converting an alphanumeric keyword into integer, some folding scheme is needed to prevent numerical overflow. The Library 2000 search engine implementation uses the radix K folding scheme. This scheme converts the keyword to a base K number by treating each character in the key as a digit in a base K number, where K is the total number of distinct characters. It then applies a modulo operation on this number with a large prime [5].

The character range of the keywords includes 26 alphabets, 10 digits, and the punctuation ".". Therefore the value of K is chosen to be 37. The large prime used to reduce the converted number is the hash table size. This scheme has the advantage that the converted integer can be mapped into a hash table index automatically as a by-product of the modulo operation. Using a prime number as the hash table size also has an additional benefit that will be addressed in the following section.

3.1.2 Collision Resolution

The hash function described above may map two keywords into the same hash table index. Therefore we need some method for resolving collision. The Library 2000 search engine uses the open-addressing scheme. This scheme searches through the table entries to find an empty cell. The search pattern is specified by an offset rule that generates the sequence of table locations to be searched [5].

The current implementation uses a simple linear offset search pattern to search forward from the a collision point. Consider a keyword *w* and a hash function *h*. The offset rule is give by the formula

$$h_i(w) = (h(w) + y) \bmod n \tag{3.1}$$

where y is the probe number, n is the size of hash table, and $h_i(w)$ is the hash table index of the i-th probe. This resolution scheme scans the hash table linearly until it finds an empty space. The hash table size n needs to be a prime number so that the offset will cause every possible table position to be considered [5].

When the search engine builds the hash table, it first copies the keyword from the word-location pair to the wordstore. Then it passes the keyword to the hash function, which returns an integer i. It looks up the i-th entry in the hash table and examines the value of the keyword pointer field of that entry. If the pointer is NULL, this means that the hash table entry is empty. Then the search engine will set that pointer value to point to the keyword in the wordset, and fill in the other fields of the hash table entry. On the other hand, if the pointer is not NULL, it means that the entry is already occupied. Then we have to examine the next entry in the probe sequence and see if it is empty. The search engine will give up and return failure if it cannot find an empty slot after 20 trials. In this case, we need to increase the size of the hash table and then rebuild the hash table. This scheme imposes 20 as an upper bound on the number of keyword comparisons needed for a keyword search, hence guaranteeing an $O(1)$ search time.

During a keyword search, the keyword is first passed to the hash function, which returns an integer i. Then we get the i-th entry record of the hash table, and compare the keyword with the word pointed to by the keyword pointer field of the record. If the word pointer is NULL, the search engine will return that the keyword is not found. If the original keyword matches with the one pointed to by the word pointer, then the search is successful. Otherwise we go to the next entry in the probe sequence and do the keyword comparison again. The word pointed to by the keyword pointer field is stored in the wordstore, which is already loaded into main memory. Thus the wordstore provides a very efficient mechanism for keyword comparisons. The search engine will return that the keyword is not found if it cannot find a matching hash table entry after 20 trials.

3.2 The Loading Factor

In this section, we study the effect of the loading factor of the hash table on the collision resolution process. The loading factor α of a hash table is defined as

$$\alpha \equiv \frac{Keyword\ \ Set\ \ Size}{Hash\ \ Table\ \ Size} \tag{3.2}$$

Recall that the hash function we use maps a keyword to a number by first converting the word to a base 37 number, and then applying a modulo operation on the hash table size. The smaller the hash table size, the larger will be the chance that two or more keywords being mapped to the same number. Therefore we would expect collision rate to rise with the increase in the loading factor.

We can measure the collision rate by finding the average number of keyword comparisons it takes to do a keyword search. The measurement is done in the following way. We first build a hash table for all the keywords in the MIT Barton library index. Then we run a test program that scans through the set of keywords and does a search on each of them. After that, we can use the UNIX profiling tool "gprof" to get an execution profile of the test program. The execution profile contains information such as the number of times a function has been called, and the total CPU time consumed by a function. In particular, we can find out the total number of string comparisons executed by the test program from the profiling data. The average number of keyword comparisons per keyword search can be calculated by dividing the total number of string comparisons by the total number of keywords.

The following table summarizes the experimental results for hash tables with various sizes.

Number of hash table entries	Number of keywords	Total number of keyword comparisons	Loading factor	Average number of comparisons per keyword search
3000017	1140389	1417158	0.38	1.24
4000037	1140389	1389255	0.29	1.22
5000011	1140389	1293801	0.23	1.13
6000011	1140389	1241504	0.19	1.09
7000003	1140389	1217349	0.16	1.07

Table 3.1: Average number of key comparisons for different hash table sizes.

Figure 3.1 on page 19 shows a plot of the average number of comparisons per keyword search versus the loading factor. We can clearly observe the trend that the number of collisions decreases as the hash table size increases. This illustrates an important characteristic of open-addressing hashing — its flexibility in trade-off between time and space. We can sacrifice keyword search performance to save memory consumption, or increase the hash table size to improve the search performance.

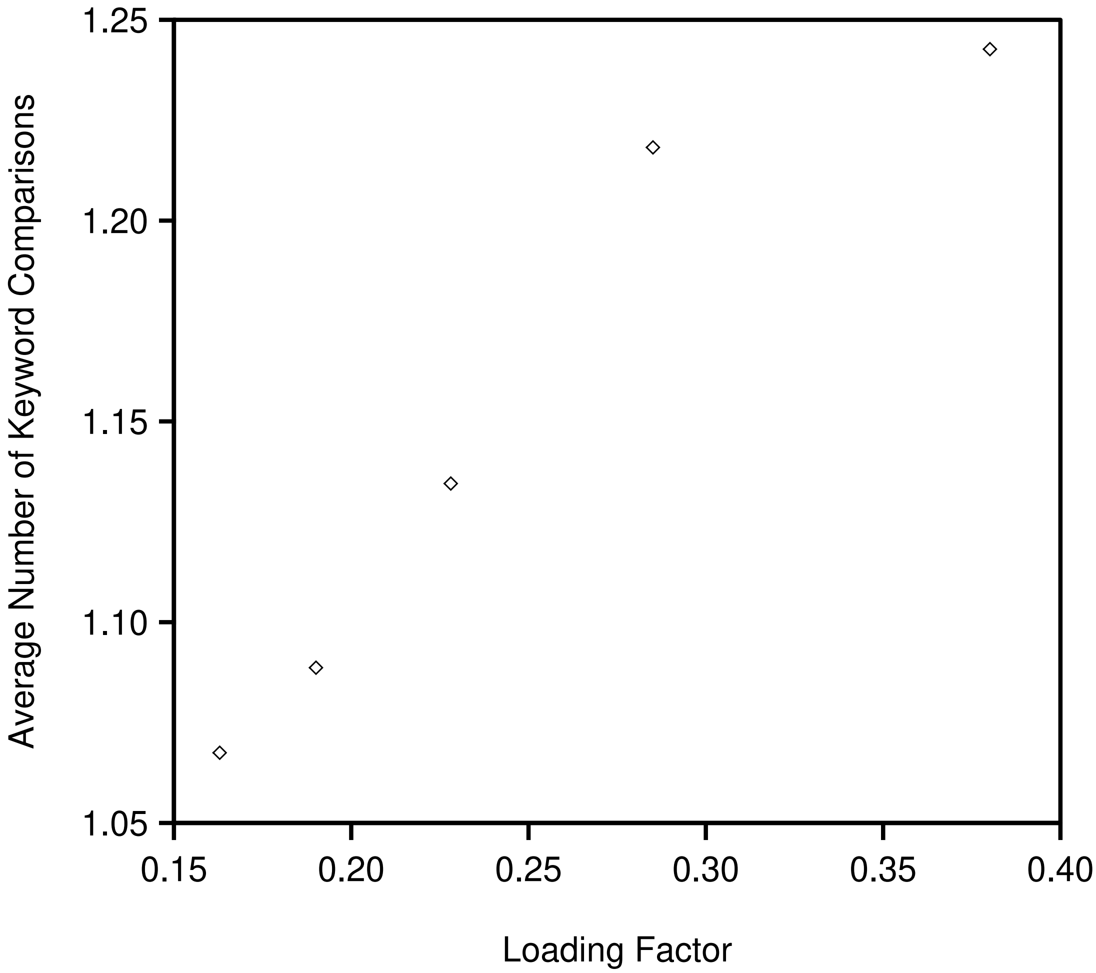

Figure 3.1: Average number of key comparisons vs. the loading factor.

Chapter 4

The Minimal Perfect Hashing Algorithm

4.1 Definition

Consider a set **W** of m keywords, and a set **I** of k integers. A hash function is a function h that maps **W** into **I**. For two distinct keywords w_1 and w_2, if $h(w_1) = h(w_2)$, then w_1 and w_2 are called synonyms. The existence of synonyms causes the problem called collision. One of the solutions to this problem is the open-addressing algorithm, which is described in Chapter 3.

Another solution is to avoid the problem, by using a hash function that does not cause collision. Such a function is called a perfect hash function. A perfect hash function is an injection mapping from **W** to **I**. This means that every member of **I** is a value $h(w)$ for at most one w in **W**. If **W** and **I** have the same number of elements, i.e. $m = k$, then h is a one-to-one mapping, or a bijection, from **W** to **I**. Such a function is called a minimal perfect hash function.

Using a minimal perfect hash function, each keyword search will take only one key comparison. We first map the keyword to an integer i, and then compare the keyword with the one pointed to by the i-th hash table entry. If the keywords match with each other, then the search is successful, otherwise we can conclude that the keyword is not in the hash table.

Another advantage of the minimal perfect hashing algorithm is its low memory consumption. In Chapter 3, we have seen that in order to reduce the collision rate, the open-addressing algorithm needs to use a hash table with size much larger the keyword set size. But since the minimal perfect hash function is a bijective function, the hash table size will be exactly equal to the keyword set size.

However, the minimal perfect hashing algorithm also has one drawback. Every time we add a new word to the keyword set, we need to regenerate the hash function. On the other hand, if we use the open-addressing scheme, we can insert new keywords into the hash table dynamically. Of course, if the loading factor of the hash table increases to such a level that a keyword search needs more than 20 key comparisons, we have to increase the hash table size and regenerate the hash table. But the open-addressing scheme does offer more flexibility for incremental keyword set updates than the minimal perfect hashing algorithm.

4.2 Implementation

The minimal perfect hashing algorithm studied in this thesis is developed by George Havas, Bohdan S. Majewski, Nicholas C. Wormald and Zbigniew J. Czech [3]. Bohdan S. Majewski has implemented this algorithm in the C language. Permission has been obtained from him to use the source code in the Library 2000 project for research purposes. This section will give an overview of this algorithm as described in [3,4].

A family of minimal perfect hashing algorithms based on random r-graphs is presented in [3]. An r-graph is a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, where each $e \in \mathbf{E}$ is an r-subset of $\mathbf{V}$. This means that each edge of the graph has r vertices. Majewski's C language code uses a 3-graph implementation. The hash function is of the form

$$h(w) = g(f_1(w)) \oplus g(f_2(w)) \oplus g(f_3(w)) \tag{4.1}$$

where $\oplus$ is an exclusive or operation.

Generating the hash function involves two steps: mapping and assignment.

4.2.1 Mapping

Consider a keyword set **W** with m keys. In the mapping step, **W** is mapped into a 3-graph with m edges and n vertices. The value of n depends on m, and the method to determine it will be explained later in this section.

The mapping has to satisfy one requirement: the edges of the 3-graph must be independent. The edge independence criterion is defined in [3] as:

> "Edges of an r-graph **G** = (**V**, **E**) are independent if and only
> if repeated deletion of edges containing vertices of degree 1
> results in a graph with no edges in it."

For a 2-graph, that means that the graph has to acyclic.

The mapping is done in a probabilistic manner. We generate a mapping randomly, and repeat that process until the edges of the graph are independent. For each keyword, we use three functions (f_1, f_2, and f_3) to map it into three vertices. Then these three vertices will form an edge of the graph. Therefore the graph **G** = (**V**, **E**) is defined by **V** = {0, ... , n-1}, **E** = $\{\{f_1(w), f_2(w), f_3(w)\} : w \in$ **W**$\}$. The f_i functions maps a keyword randomly into the range [0, n-1]. It is defined as:

$$f_i(w) = \left(\sum_{j=1}^{|w|} T_i(j, w[j])\right) \bmod n \tag{4.2}$$

where $|w|$ is the number of characters in the keyword w, and $w[j]$ is its j-th character. T_i is a two dimensional array of random integers, indexed by a character and the position of the character in a keyword.

The value of n determines the probability of generating an edge independent graph. Theorem 4 in [3] states that

> "For any r-graph three exists a constant c_{inv} depending only

on r such that if $m \le c_{inv}\, n$ the probability that a random r-graph has independent edges tends to 1 (as m goes to infinity)."

For a 3-graph, the value of c_{inv} is experimentally determined to be 1.23 [3].

4.2.2 Assignment

In the assignment step, we find a function g that maps **V** into the range [0, m-1]. The function g needs to satisfy the requirement that the function h defined as

$$h(e = \{v_1, v_2, v_3\} \in E) = g(v_1) \oplus g(v_2) \oplus g(v_3) \tag{4.3}$$

is a bijective mapping from **E** to the range of integers [0, m-1]. This means that we have to assign a value between 0 and m-1 for each of the vertices, so that when we apply the exclusive or operation on the values of the three vertices of each edge, we will get a unique number.

This assignment problem can be solved easily given that the graph **G** is edge independent. The solution is outlined in [3] as follows: "Associate with each edge a unique number $h(e) \in$ [0, m-1] in any order. Consider the edges in reverse order to the order of deletion during a test of independence, and assign values to each as yet unassigned vertex in that edge. As the definition implies, each edge (at the time it is considered) will have one (or more) vertices unique to it, to which no value has yet been assigned. Let the set of unassigned vertices for edge e be $\{v_1, v_2, \ldots, v_j\}$. For edge e assign 0 to $g(v_1)$, $g(v_2)$,..., $g(v_{j-1})$." Now we can find the value of $g(v_j)$ by applying the exclusive or operation to $h(e)$ and the $g(v)$ values for all the vertices of e except v_j.

Recall that each edge e is mapped from a keyword w by the functions f_1, f_2 and f_3. Therefore we can rewrite the minimal perfect hash function in the form

$$h(w) = g(f_1(w)) \oplus g(f_2(w)) \oplus g(f_3(w)) \tag{4.4}$$

4.3 Time Bound on Hash Table Generation

The expected time complexity of generating the minimal perfect hash function is proven to be $O(rm+n)$ in [3]. For the 3-graph implementation, $r = 3$ and $n = 1.23m$. Therefore the time bound on hash function generation is $O(m)$. In this section we will verify this result experimentally.

The hash function generation code is run on an IBM RISC/6000 530H workstation, running under the AIX 3.2 operating system. The program execution time is measured using the UNIX profiling tool “gprof”. The following table lists the hash function generation time for keyword sets with various sizes. The time is obtained by averaging the results of 20 test runs.

Number of keywords	Hash function generation time / seconds
5000	0.4
10000	0.8
50000	4.2
100000	8.4
250000	22.8
500000	46.6
750000	70.4
1000000	96.5

Table 4.1: Hash function generation time for different keyword set sizes.

Figure 4.1 on page 25 shows a plot of hash function generation time versus the keyword set size. We can observe that the generation time increases linearly with the increase in number of keywords. This verifies the theoretical time bound.

Using this minimal perfect hashing algorithm, the hash function can be generated at a speed fast enough for practical uses. For the MIT Barton Library catalog, which contains

1,140,389 keywords, it only takes 112 seconds to generate the hash function. Moreover, we only need to generate the hash function when we rebuild the index for the library catalog. The overhead cost of generating the hash function is insignificant when compared to the wordset generation time.

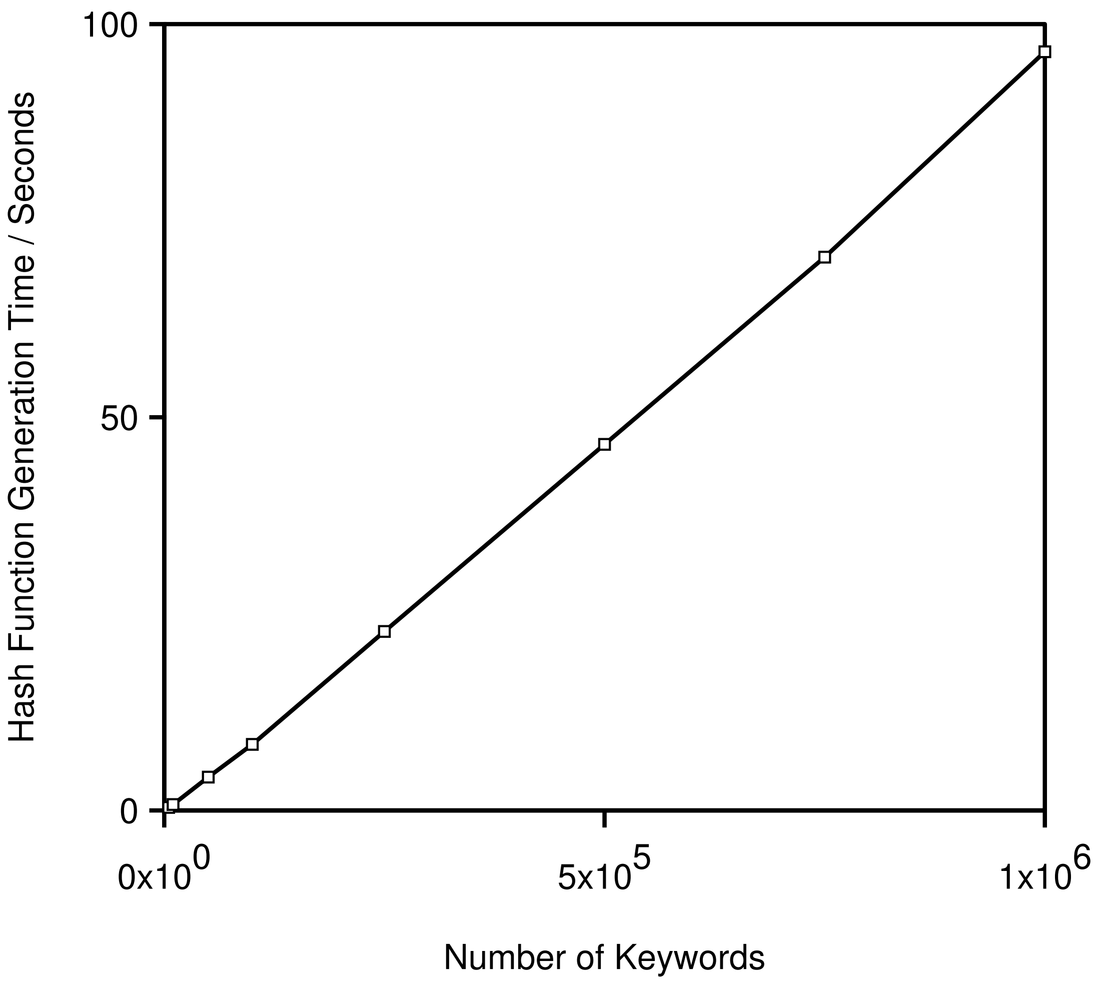

Figure 4.1: Plot of hash function generation time vs. the number of keywords.

Chapter 5

Performance Evaluation

5.1 The Performance Measurement Procedure

In this chapter, we will compare and analyze the performance of the open-addressing hashing algorithm and the minimal perfect hashing algorithm. The performance measurement is done in the following manner. We first compile a search engine for each of the hashing algorithms. Four versions of search engine are compiled for the open-addressing hashing algorithm, and they differs with each other in the loading factor. The loading factors vary from 0.38 to 0.19. Another search engine is compiled using the 3-graph implementation of the minimal perfect hashing algorithm. Since the hash function is perfect, the loading factor is 1 in this case.

In the next step, we run the search engines to search the 1,140,389 keywords of the MIT Barton Library catalog one by one. Then we use the UNIX profiling tool "gprof" to get an execution profile for each test run. From the execution profile, we can get information on the total execution time, and the total number of keyword comparisons that have been made.

The following measures are employed to ensure fairness in comparing the timing results. All the hashing code is compiled with the "-O3" flag, which instructs the compiler to use the highest level of code optimization. We also use the "-Q" flag to inline all the functions in the hashing code. This eliminates function call overhead, which can vary significantly among different hashing algorithms.

5.2 Performance Comparison

The following table summarizes the performance data for the open-addressing hashing algorithm and the minimal perfect hashing algorithm. The programs are run on an IBM

RISC/6000 530H workstation with 512 MB of main memory to ensure that there is no virtual memory activity affecting the performance. The test program uses the system call “getrusage” to monitor the number of page faults serviced that require disk I/O activity. The number of page faults is verified to be 0 for all the test runs. The execution time is obtained by averaging the result of 20 test runs.

Hashing algorithm	Hash table size	Loading factor	Total number of keyword comparisons	Average comparisons per keyword search	Total time consumed / seconds
open-adr	3000017	0.38	1417158	1.24	15.7
open-adr	4000037	0.29	1389255	1.22	15.2
open-adr	5000011	0.23	1293801	1.13	14.3
open-adr	6000011	0.19	1241504	1.09	13.8
min. perfect	1140389	1.00	1140389	1.00	17.2

Table 5.1: Hash performance data.

Figure 5.1 on page 28 shows a bar chart of the total hashing time for the various hashing algorithms.

5.3 Performance Analysis

From the experimental data, we can observe a surprising result: the performance of the minimal perfect hashing algorithm is worse than the other four configurations of the open-addressing hashing algorithm. Let us compare the minimal perfect hashing algorithm with the slowest configuration of the open-addressing hashing scheme (the one with a loading factor of 0.38). The open-addressing scheme makes 24% more keyword comparisons that the minimal perfect hashing algorithm, but it is still 9% faster. This leads us to suspect that the minimal perfect hashing function might be much more computationally expensive than the hash function used the open-addressing scheme. To solve this mystery, we need to take a closer look at the hash functions used by these hashing algorithms.

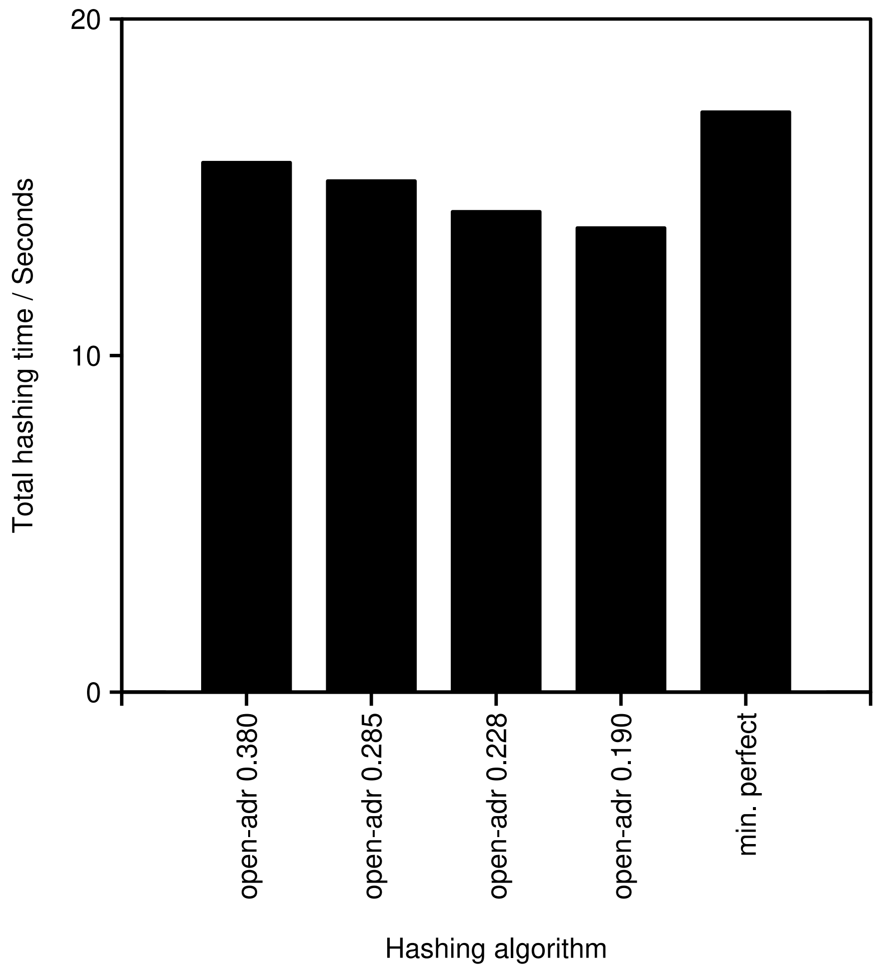

Figure 5.1: Plot of the total hashing time for different hashing algorithms.

First let us study the hash function used in the open-addressing scheme. The function uses a radix K folding scheme to convert a keyword into a base 37 number, modulo the hash table size. This function starts by setting the value of a running sum S to 1. Then beginning from the first character of the keyword, it executes the following steps for each character:

1. Convert the character into an integer in the range [0,36].

2. Multiply S by 37, then add the result from step 1.

3. Apply a modulo operation between the result in step 2 and the hash table size, and store the result in S.

4. Terminate and return S when we reach the end of the keyword.

This hash function is very simple. It consists of mostly computational operations with very few memory accesses.

Now let us study the minimal perfect hashing function. Recall that the hash function is of the form

$$h(w) = g(f_1(w)) \oplus g(f_2(w)) \oplus g(f_3(w)) \tag{5.1}$$

f_1, f_2, and f_3 are defined as:

$$f_i(w) = \left(\sum_{j=1}^{|w|} T_i(j, w[j])\right) \bmod n \tag{5.2}$$

T_i is a two dimensional array of random integers, indexed by a character and the position of the character in a keyword. For a keyword set with m keywords, $n = 1.23m$.

The hash function maps the keyword into an integer in the range [0, m]. The function first sets three running sums S_1, S_2 and S_3 to 0. Then for each character in the keyword, it carries out the following steps:

1. Lookup the three random numbers corresponding to the character from tables T_1, T_2 and T_3 respectively.

2. Add the three numbers to S_1, S_2 and S_3 respectively, and store the result modulo n into S_1, S_2 and S_3.

When we have done this for all the characters in the keyword, S_1, S_2 and S_3 will contain the value of $f_1(w)$, $f_2(w)$, and $f_3(w)$. Now we do three more table lookups to get the values of $g(S_1)$, $g(S_2)$ and $g(S_3)$ from the g-table. The value of $h(w)$ can be obtained by applying the exclusive or operation on those three numbers.

For each character in the keyword, the function executes three additions, three modulo operations, and three memory accesses. The computational cost is very low. Now we have to determine if the memory accesses are going to cause any cache misses. The IBM RISC/6000 530H workstation uses a 64KB 4-way set associative data cache. A memory access with cache hit takes 1 cycle, while each data cache miss costs a delay of approximately 8 clock cycles [8]. The total size of the random number tables T_1, T_2 and T_3 is 14.3KB. Therefore these tables will fit into the data cache. The only memory access overhead incurred is when we are accessing a table entry for the first time and thus causing a cache miss. But since the test program hashes more than one million keywords in total, the cache miss penalty would be insignificant compared to the total hash time.

Memory accesses are much more costly when we are looking up values from the g-table. The g-table contains n entries. For the MIT Barton library catalog,

$$n = 1.23m = 1.23 \times 1140389 = 1402679 \tag{5.3}$$

Each table entry contains a 4-byte integer. Thus the total table size is about 5.6MB. Therefore each table access almost guarantees a cache miss. But something worse might happen. The memory access may even cause a TLB (Translation Lookaside Buffer) miss. The TLB caches the translation of virtual page numbers into physical page numbers of the recently accessed memory pages. The workstation uses a two-way set associative TLB with 64 entries in each set [9]. Thus the TLB can cache the page number translations for 128 pages. At 4KB per page, that is equivalent to 512KB of data [8]. Thus when we are doing random access over the 5.6MB g-table, it is highly likely for TLB misses to occur. A TLB miss will incur a delay of at least 36 clock cycles [8]. Since we need to access the g-table three times for each keyword, TLB misses could add a delay of 108 cycles in the worst case.

From the above comparisons, we can conclude that minimal perfect hashing is slower than open-addressing hashing due to memory access latencies. In the following chapter, we will make some modifications to the minimal perfect hashing algorithm in order to reduce the number of memory accesses needed to hash each keyword.

Chapter 6

Performance Enhancement to the Minimal Perfect Hashing Algorithm

6.1 A 2-graph Implementation

As we have seen in Chapter 4, the hashing performance of the 3-graph implementation is worse than the open-addressing algorithm. The major culprit of the sluggish performance is the memory access latencies involved in looking up values from the *g*-table. In this chapter, we will modify the algorithm by changing the 3-graph implementation to a 2-graph implementation. The goal of this modification is to reduce the number of memory accesses needed by the hash function.

A 2-graph is a graph in which every edge consists of two vertices. A detailed description of the 2-graph implementation of the minimal perfect hashing algorithm can be found in [4]. It is very similar to the 3-graph implementation. For a keyword set **W** with m keywords, the function bijectively maps **W** into the set of integers $\{0, \ldots, m\text{-}1\}$. The hash function is of the form

$$h(w) = g(f_1(w)) \oplus g(f_2(w)) \tag{6.1}$$

where f_1 and f_2 are functions that map a keyword into an integer in the range $[0, n]$. The functions are defined as:

$$f_i(w) = \left(\sum_{j=1}^{|w|} T_i(j, w[j])\right) \bmod n \tag{6.2}$$

T_1 and T_2 are two dimensional arrays of random numbers indexed by a character and the position of the character in a keyword.

Generating the hash function involves the mapping step and the assignment step. These steps are exactly the same as those of the 3-graph implementation. Recall that the mapping step is a probabilistic process. We map the keyword set into a 2-graph using the random number tables T_1 and T_2, and repeat that process until we get an acyclic graph. The probability of getting an acyclic graph depends on the value of n. Theorem 3 in [3] states that

> "Let **G** be a random graph with n vertices and m edges obtained by choosing m random edges with repetitions. Then if $n = cm$ holds with c > 2 the probability that **G** has independent edges, for $n \to \infty$, is

$$p = e^{\frac{1}{c}} \sqrt{\frac{c-2}{c}}\text{,''} \tag{6.3}$$

The value of c is chosen to 3. For sufficiently large keyword sets, the probability value is approximately 0.81. Therefore the expected number of iterations needed to generate an acyclic graph is 1.24. The total time used by the mapping and assignment steps to generate the hash function is bounded by *O(m)* [3]. Compared with the 3-graph implementation, the hash function generation time of the 2-graph implementation is about 10% slower. For example, with the keyword set of the MIT Barton Library catalog index, the 2-graph implementation takes about 122 seconds to generate the hash function.

6.2 Performance Comparison

We use the 2-graph implementation of the minimal perfect hashing algorithm to hash the 1,140,389 keywords of the MIT Barton Library catalog index. Then we compare the timing results with the performance data enlisted in Chapter 5. The following table summarizes the hash performance data. Figure 6.1 shows a bar chart plot of the data.

Hashing Algorithm	Loading Factor	Time consumed / seconds
open-addressing	0.38	15.7
open-addressing	0.29	15.2
open-addressing	0.23	14.3
open-addressing	0.19	13.8
minimal perfect 3-graph	1.00	17.2
minimal perfect 2-graph	1.00	14.2

Table 6.1: Hash performance data.

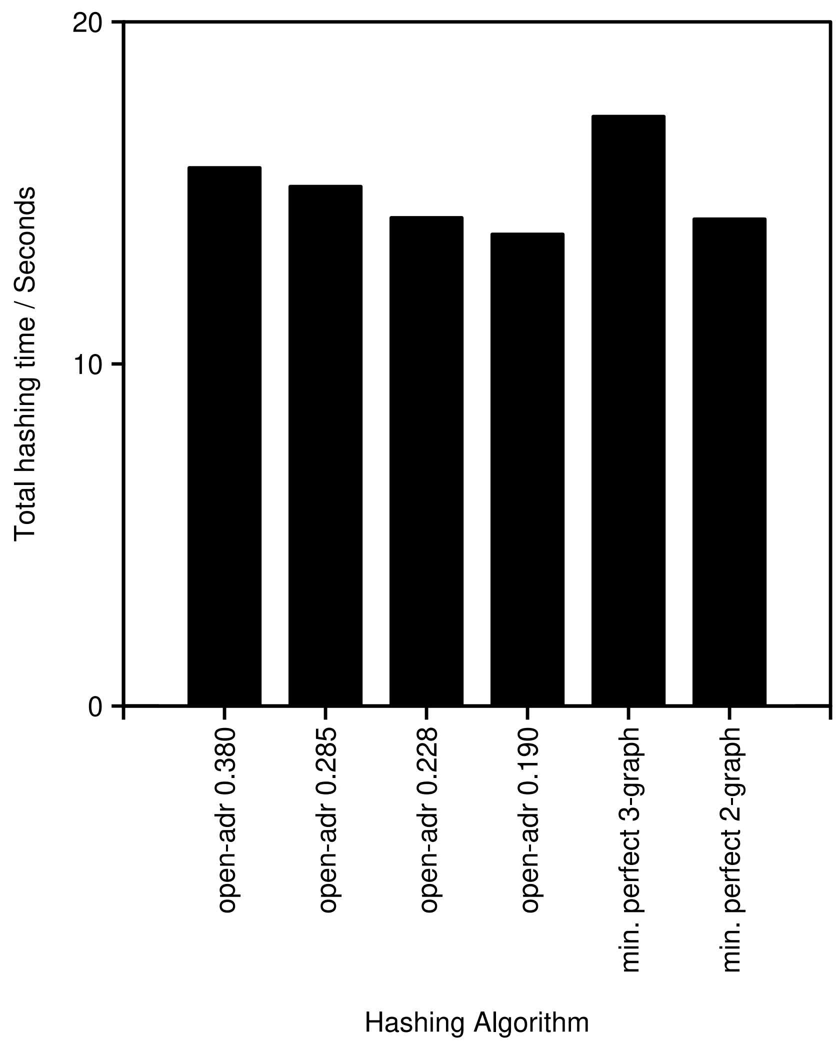

Figure 6.1: Plot of the total hashing time for different hashing algorithms.

6.3 Performance Analysis

From the hash performance data, we can observe that the 2-graph implementation of the minimal perfect hashing algorithm does offer a significant performance improvement: it is 17% faster than the 3-graph implementation. The performance of the 2-graph implementation also exceeds that of the open-addressing hashing algorithm with loading factors of 0.38, 0.29 and 0.23. But it is still slower than the open-addressing scheme with a loading factor of 0.19.

The performance improvement of the 2-graph implementation can be attributed to the increase in speed of its hash function. The 2-graph hash function saves one addition and one modulo operation per character, since for each keyword w it only needs to calculate the values of $f_1(w)$, and $f_2(w)$, while the 3-graph implementation has to calculate the values of $f_1(w)$, $f_2(w)$, and $f_3(w)$. Moreover, for each character of the keyword, the hash function only needs to do two random number table lookups (from T_1 and T_2), instead of three table lookups in the case of the 3-graph implementation. However, since the random number tables can fit into the data cache in both cases, the reduction in memory access latency will not be very significant.

On the other hand, we can get a large performance gain in the g-table lookups. For each keyword, the 2-graph hash function only needs to read two values from the g-table, while the 3-graph implementation requires three. As we have concluded in Chapter 5, memory accesses to the g-table are very likely to cause TLB misses. Therefore by using the 2-graph scheme, we manage to save one TLB miss on each keyword search.

However, the 2-graph minimal perfect hashing algorithm is still slower than the open-addressing hashing scheme with a loading factor of 0.19. The collision rate of the open-addressing scheme at such a loading factor is very low. The average number of keyword comparisons needed for each keyword search is only 1.09 (see Table 5.1 on page 27).

Therefore, although the keyword comparison overhead of the open-addressing scheme is 9% more than that of the minimal perfect hashing algorithm, it still outperforms the minimal perfect hashing scheme. This lead us to conclude that the hash function used in the 2-graph implementation is still slower than the hash function of the open-addressing scheme. Changing from the 3-graph implementation to 2-graph does make some improvement, but the improvement is not enough to make minimal perfect hashing outperform the open-addressing scheme.

At higher loading factors such as 0.38, 0.29 and 0.23, the open-addressing algorithm is slower than the minimal perfect hashing algorithm. But that is due to the high collision rate at those loading factors. The performance advantage of the open-addressing scheme's hash function is not enough to compensate for the overhead of collision resolution. But when we lower the loading factor down to 0.19, the open-addressing scheme will regain its performance advantage. This illustrates another important advantage of the open-addressing scheme: its flexibility in space-time trade-off. We can easily trade memory space for speed by increasing the hash table size.

6.4 Memory Consumption Analysis

In this section, we will compare the amount of memory space needed by the hashing algorithms. We will compare the memory consumption of the open-addressing scheme, with a loading factor of 0.23, with that of the 2-graph minimal perfect hashing algorithm. We choose these two candidates because they have about the same level of hashing performance (see Table 6.1 on page 34).

For the open-addressing algorithm with a loading factor of 0.23, the wordset hash table has 5,000,011 entries. Each entry of the wordset hash table has 4 fields, and each field takes up 4 bytes. Hence the total size of the hash table is 78.1MB.

For the minimal perfect hashing algorithm, we need to account for three data structures: the wordset hash table, the *g*-table, and the random number tables T_1 and T_2. Since the loading factor is 1 for perfect hashing, the hash table contains 1,140,389 entries, which takes up 17.8 MB. The number of entries in the *g*-table is three times the number of keywords, and each entry is a 4-byte integer. Thus the *g*-table consumes 13.4 MB. The total size of T_1 and T_2 is only 9.5 KB, which is negligible when compared to the size of the other two tables. The total memory space consumption is 31.2 MB.

Therefore for the similar level of keyword search performance, the memory space needed by the minimal perfect hashing algorithm is only 40% of that required by the open-addressing scheme.

In the next chapter, we will summarize the observations we have made from the experimental data, and draw some conclusions.

Chapter 7

Conclusion

7.1 Minimal Perfect Hashing versus Open-addressing

In this section, we will compare the minimal perfect hashing algorithm with the open-addressing scheme basing on the experimental data we have collected. Then we will draw a conclusion on which algorithm is more suitable for main memory indexing.

The comparisons will be done in three categories: performance, simplicity and memory consumption.

7.1.1 Performance

In Chapter 6, we have seen that the hashing performance of the open-addressing scheme using a loading factor of 0.23 is about the same as that of the 2-graph minimal perfect hashing algorithm. At any loading factor below about 0.2, the open-addressing scheme will outperform minimal perfect hashing. Although open-addressing hashing needs on average more than one keyword comparison per keyword search, while minimal perfect hashing needs only one keyword comparison, open-addressing hashing is still faster. This can be explained by the fact that the hash function used in the open-addressing scheme is more efficient, and that speed more than outweighs the collision resolution overhead of the scheme.

7.1.2 Simplicity

Open-addressing hashing is obviously the winner in this category. Its hash function can be coded in a few lines. The linear offset collision resolution scheme is also simple and easy to implement. Therefore it is a piece of software that is very easy to maintain.

On the other hand, the minimal perfect hashing scheme needs a very complicated graph algorithm to generate the hash function. It is also more complex in terms of data structures. In addition to the wordset hash table, it also needs the random number tables (T_1 and T_2) and the g-table. We need to dump these tables into a file after we have generated the hash function. That file has to be loaded into main memory when the search engine is in operation.

7.1.3 Memory Consumption

For a comparable level of keyword search performance, the open-addressing scheme uses 78.1 MB of memory, while the minimal perfect hashing algorithm uses 31.2 MB. Thus the minimal perfect hashing algorithm saves memory consumption by 46.9 MB. This is quite significant when we compare it to the wordset size, which is 146.1 MB.

However, such a reduction in memory consumption will become insignificant as the library system scales up. The number of catalog entries in a library will grow at a much faster rate than the size of the keyword set of the index. Therefore the size of the wordset will increase much more rapidly than the number of keywords in the wordset. The size of the wordset hash table will soon become insignificant comparing to the wordset size.

Combining the three factors enlisted above, we can reach the conclusion that the open-addressing scheme is more suitable for main memory indexing application than the minimal perfect hashing algorithm.

7.2 Future Outlook

Looking into the future, as technology advances and the size of library indexes scales up, is it possible that the minimal perfect hashing algorithm might become more preferable? The answer seems to be unlikely.

On one hand, CPU speed will increase at a faster rate than the memory access speed. The minimal perfect hashing algorithm will outperform the open-addressing scheme only if we can achieve single clock cycle memory access latency. But this is unlikely to happen due to the increasing gap between the CPU clock cycle time and the memory access latency. One might wish that a larger on-chip cache may solve the problem. But recall that even for the MIT Barton Library catalog index, the size of the g-table used by the minimal perfect hashing algorithm is 13.4 MB. This number will become much larger when we try to index the Library of Congress and when the size of the keyword set grows in the future. Therefore is it unrealistic to hope that we can fit the g-table into the data cache.

On the other hand, main memory will become more and more affordable in the future. This means that we will be able to further improve the performance of the open-addressing scheme by using a larger hash table size (hence a lower loading ratio). But the minimal perfect hashing algorithm does not have this flexibility of space-time trade-off, and will not be able to benefit from the lower main memory cost.

The minimal perfect hashing algorithm also has the disadvantage that every time we add a new keyword into the keyword set, we need to regenerate the hash function. In the current implementation, since we happen to rebuild our wordset from scratch on every catalog update, the hash table regeneration overhead is insignificant. However, as the wordset grows in size in the future, we might want to update the wordset incrementally. The open-addressing algorithm will be able to take advantage of that scheme since it allows keywords to be inserted dynamically into the hash table. Incremental additions do, of course, increase the loading factor. Therefore eventually it will be necessary to rebuild the hash table with a larger size to bring the loading factor back down. But the open-addressing still offers more flexibility in keyword set updates than the minimal perfect hashing algorithm.

References

[1] Jerome H. Saltzer. Technology, Networks, and the Library of the Year 2000. *Proceedings of the International Conference on the Occasion of the 25th Anniversary of Institut National de Recherche en Informatique et Automatique*, Paris, France, December, 1992, pages 51-67.

[2] Library 2000. *MIT Laboratory for Computer Science Progress Report*, Vol. 30, July 1992 — June 1993, pages 147-158.

[3] George Havas, Bohdan S. Majewski, Nicholas C. Wormald and Zbigniew J. Czech. Graphs, Hypergraphs and Hashing. *Proceedings of 19th International Workshop on Graph-Theoretic Concepts in Computer Science*, Utrecht, The Netherlands, June 1993.

[4] Zbigniew J. Czech, George Havas and Bohdan S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, Vol. 43, No. 5, October 1992, pages 257-264.

[5] Ted G. Lewis and Curtis R. Cook. Hashing for Dynamic and Static Internal Tables. *Computer*, Vol. 21, No.10. IEEE Computer Society, October 1988, pages 45-56.

[6] Tobin J. Lehman and Michael J. Carey. A Study of Index Structures for Main Memory Database Management Systems. *Proceedings of the 12th International Conference on Very Large Data Bases*, Kyoto, August 1986, pages 294-303.

[7] Anastasia Analyti and Sakti Pramanik. Fast Search in Main Memory Databases. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1992, pages 215-224.

[8] IBM AIX Version 3.2 for RISC System/6000. *Optimization and Tuning Guide for the XL FORTRAN and XL C Compilers*. First Edition, September 1992.

[9] IBM RISC System/6000 POWERstation and POWERserver. *Hardware Technical Reference General Information*. First Edition, 1990

[10] Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, McGraw-Hill, 1990.

[11] Donald E. Knuth. *The Art of Computer Programming, Volume 3 / Sorting and Searching*. Addison-Wesley Publishing Company, 1973.