PERFECT HASHING AND RELATED PROBLEMS

by

Ramana Rao Juvvadi

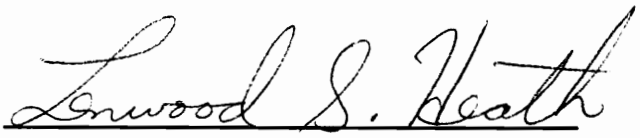
Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

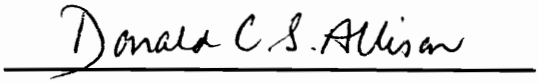
in

Computer Science

APPROVED:

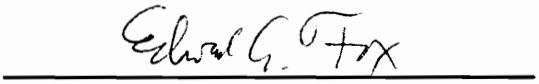


Dr. Lenwood S. Heath, Chairman

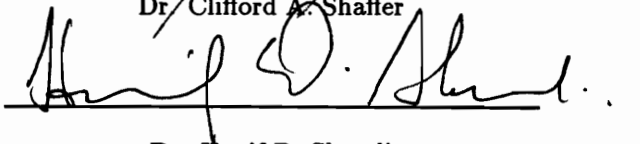


Dr. Donald C. S. Allison

Dr. Clifford A. Shaffer



Dr. Edward A. Fox



Dr. Hanif D. Sherali

June, 1993

Blacksburg, Virginia

PERFECT HASHING AND RELATED PROBLEMS

by

Ramana Rao Juvvadi

Committee Chairman: Dr. Lenwood S. Heath

Computer Science

(ABSTRACT)

One of the most common tasks in many computer applications is the maintenance of a dictionary of words. The three most basic operations on the dictionary are *find, insert*, and *delete*. An important data structure that supports these operations is a *hash table*. On a hash table, a basic operation takes $O(1)$ time in the average case and $O(n)$ time in the worst case, where n is the number of words in the dictionary. While an ordinary hash function maps the words in a dictionary to a hash table with collisions, a *perfect hash function* maps the words in a dictionary to a hash table with no collisions. Thus, perfect hashing is a special case of hashing, in which a find operation takes $O(1)$ time in the worst case, and an insert or a delete operation takes $O(1)$ time in the average case and $O(n)$ time in the worst case.

This thesis addresses the following issues.

- *Mapping, ordering and searching (MOS)* is a successful algorithmic approach to finding perfect hash functions for static dictionaries. Previously, no analysis has been given for the running time of the MOS algorithm. In this thesis, a lower bound is proved on the tradeoff between the time required to find a perfect hash function and the space required to represent the perfect hash function.

- A new algorithm for static dictionaries called the *musical chairs(MC)* algorithm is developed that is based on ordering the hyperedges of a graph. It is found experimentally that the MC algorithm runs faster than the MOS algorithm in all cases for which the MC algorithm is capable of finding a perfect hash function.

- A new perfect hashing algorithm is developed for dynamic dictionaries. In this algorithm, an insert or a delete operation takes $O(1)$ time in the average case, and a find operation takes $O(1)$ time in the worst case. The algorithm is modeled using results from queueing theory.

- An ordering problem from graph theory, motivated by the hypergraph ordering problem in the MC algorithm, is proved to be NP-complete.

ACKNOWLEDGEMENTS

I dedicate this thesis to my parents who taught me at a very young age that learning can be fun too. But for my wife Sridevi's support, I would never have been able to finish writing this dissertation. It is very difficult to imagine how I could have sustained my work all these years without encouragement from my friends Ramana Idury and Kumar Vadaparthy.

I would like to thank my adviser Lenwood Heath from whom I learnt that to be a scholar one needs not only clarity in thinking but also clarity in writing. I would like to thank all my committee members, Dr Fox, Dr Allison, Dr Shaffer, and Dr Sherali, for their helpful comments.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

One of the most common tasks in many computer applications is the maintenance of a *dictionary* of words. The three most important operations on the dictionary are *find*, *insert*, and *delete*. The literature contains several data structures that support these operations for maintaining a dictionary. These data structures can be broadly divided into three categories: *Search Trees*, *Tries*, and *Hash Tables* [10]. To analyze the relative advantages and disadvantages of these three data structures carefully, it is essential to model the representation of the words correctly. In this dissertation, we divide dictionaries into two categories—those of fixed-length words and those of variable-length words. When the words are of fixed-length, the cost of comparing one word to another is always the same. When the words are of variable-length, the cost of comparing two words depends on their lengths.

We refer to *find*, *insert*, and *delete* operations as *basic operations*. Suppose there are n words in a dictionary, and a word x in the dictionary is l characters long. If the dictionary is represented as a balanced search tree, a basic operation requires, in the worst-case, a traversal of $\Theta(\log n)$ links and a comparison of x with $\Theta(\log n)$ words. If the dictionary is represented as a trie, a basic operation, in the worst case, requires a traversal of l links and comparison of the characters of x with l other characters. If it is represented as a hash table with either separate chaining or coalesced chaining, a basic operation should require, in the average case, a traversal of $O(1)$ links and comparison of x with $O(1)$ words. In the worst case, it could require traversal of $O(n)$ links and a comparison of x with $O(n)$ words. When the size of the dictionary is small enough to fit in main memory, the cost of comparing words is usually of the same order as the cost of traversing links. However, when

the dictionary is too large to fit in main memory, it is stored on a disk and traversing the links becomes far more expensive than comparing words.

Recently, a method called perfect hashing has received considerable attention [1, 6, 9, 12, 16, 18, 20, 22, 23, 24, 30, 32, 33, 35]. As opposed to ordinary hashing, perfect hashing requires a traversal of only $O(1)$ links and a comparison of x with only one word in the worst case for a *find* operation. However, the *insert* and *delete* operations still require $O(1)$ time in the average case and $O(n)$ time in the worst case. Perfect hashing is very attractive for a large dictionary stored on a disk, especially if it is static (no insertions or deletions) or nearly so.

When perfect hashing is used for maintaining a dictionary, there are two factors that influence the total space requirement: the space occupied by the words and any space used for facilitating the basic operations. Depending on the *perfect hashing scheme* (a precise definition of perfect hashing scheme will be given later) used, it may or may not be possible to separate the data structures into two distinct parts: the ones for representing the words and the ones for facilitating the basic operations. The space occupied by the words depends on the representation of the words. In practice, when the words are of variable-length, they can be stored in an array of characters as null-terminated strings. When the words are of fixed-length, they can be stored in an array of words. While the space used for representing the words does not depend on the perfect hashing algorithm being used, the space used for basic operations depends on the algorithm being used. A good perfect hashing algorithm should minimize this space. In the case of a dynamic dictionary, another criterion for judging the merit of a perfect hashing algorithm is the time required for each basic operation. In the case of a static dictionary, it is important to consider the time required for a find operation and the time required for preprocessing.

The remainder of this chapter is organized as follows. Section 1.1 defines perfect hashing precisely and develops a framework for the comparison of different algorithms. A major portion of the space occupied by many perfect hashing algorithms consists of pointers to the words in the dictionary. So it is extremely important to use the correct type of pointer

to minimize the space required by a perfect hashing function. Section 1.2 discusses two types of pointers, and situations where they are applicable. Section 1.3 discusses previous work on perfect hashing. Section 1.4 outlines the results of this dissertation.

1.1 Definitions, Terminology, and Notation

We adopt the following notation for the rest of this dissertation.

$i^{[j]}$	$i \cdot (i-1) \cdots (i-j+1)$, the descending factorial
[ ]	Cells of an array,
	$B[i]$ refers to the contents of the ith cell of the array B
	When C is an array of characters, $C[i \cdot \cdot j]$ refers to the string obtained
	by concatenating the characters $C[i] \cdot C[i+1] \ldots C[j]$
I	Set of all non-negative integers $\{0, 1, 2, 3, \ldots\}$
I^+	Set of all positive integers $\{1, 2, 3, \ldots\}$
I_i	Set of integers between 0 and $i-1$, $\{0, 1, 2, \ldots, i-1\}$
I_i^+	Set of integers between 1 and i, $\{1, 2, \ldots, i\}$

We use the terminology of formal language theory to describe words [25]. Suppose Ψ is a finite alphabet of size greater than one, and $U \subseteq \Psi^*$ is a set of strings called the *universe.* We refer to an element of U as a *word.*

Definition 1.1.1 *When the length of the words in U is uniform, i.e., $U \subseteq \Psi^l$, for some $l > 0$, U is called a* universe of fixed-length words.

We denote a universe of fixed-length words by U_f, and a subset of U_f, $S \subset U_f$, by S_f.

Definition 1.1.2 *When the length of words in U is not uniform, U is called a* universe of variable-length words.

We denote a universe of variable-length words by U_v, and a subset of U_v, $S \subset U_v$, by S_v.

Definition 1.1.3 *For $x \in U$, $|x|$ is the length of x, and $x[i]$, where $i \leq |x|$, is the ith character of x.*

The space required to store the words in S depends on the representation of the words. Normally, we represent S_v by storing the words of S_v sequentially in an array of characters C. Each word is terminated by a special character Λ, where $\Lambda \notin \Psi$. We represent S_f by storing the words of S_f in an array B. It is not necessary to terminate each word by Λ, because the word length is uniform.

Henceforth, S_f always denotes a subset of U_f, and the words of S_f are always indexed $\{x_i : 1 \leq i \leq n\}$. Similarly, S_v always denotes a subset of U_v and the words of S_v are always indexed $\{x_i : 1 \leq i \leq n\}$. Moreover, m always equals the total number of characters required to represent all of S_v. i.e.

$$m = \sum_{i=1}^{n}(|x_i| + 1).$$

We denote S_f or S_v by S whenever it is clear from the context or whenever it is immaterial whether we are referring to S_f or S_v.

Definition 1.1.4 *b_f is the number of bits required to represent an arbitrary word in U_f.*

b_f is at least $\lceil \log_2 |U| \rceil$.

Definition 1.1.5 *b_v is the number of bits required to represent a character in $\Psi \cup \{\Lambda\}$.*

b_v is at least $\lceil \log_2(|\Psi| + 1) \rceil$. Each word $x \in U_v$ occupies $b_v(|x| + 1)$ bits of space—$b_v|x|$ bits for characters of x and b_v bits for Λ.

Definition 1.1.6 *SPACE(S_f) is the total number of bits required to store all the words in S_f:*

$$\text{SPACE}(S_f) = nb_f.$$

Definition 1.1.7 *SPACE(S_v) is the total number of bits required to store all the words in S_v:*

$$\text{SPACE}(S_v) = mb_v.$$

Definition 1.1.8 *An* address vector *for S_f in an array B of $\tilde{n}$ cells, $\tilde{n} > n$, is an n-tuple $\langle l_1, l_2, \ldots, l_n \rangle$ such that $l_j \neq l_k$ if $j \neq k$ and $1 \leq l_j \leq \tilde{n}$ for $1 \leq j, k \leq n$.*

An address vector describes a feasible way to store words of S_f in the array B, at most one word in each cell. We say that the address vector of the set S_f in B is $L = \langle l_1, l_2, \ldots, l_n \rangle$ if each word x_i is stored in $B[l_i]$.

As for S_f, an address vector can also be defined for S_v. However, an address vector for S_f describes the address of the first character of each word instead of the address of the entire word since the word occupies more than one cell.

Definition 1.1.9 *Suppose the characters of each word in S_v are stored in consecutive cells of an array C of length $\tilde{m} \geq m$. Each word is terminated by a special character Λ, and $\Lambda \notin \Psi$. An* address vector *for S_v in an array C is an n-tuple $\langle l_1, l_2, \ldots, l_n \rangle$ which satisfies the following conditions:*

- $1 \leq l_j < l_j + |x_j| \leq \tilde{m}$ *for* $1 \leq j \leq n$;
- *There exist no j and k such that $1 \leq j, k \leq n$, $j \neq k$ and $l_j \leq l_k \leq l_j + |x_j|$.*

The first condition states that the addresses of all the words should lie between 1 and $\tilde{m}$. The second condition states that the memory occupied by one word cannot overlap with that of any other word. We say that the address vector of the set S in C is $L = \langle l_1, l_2, \ldots, l_n \rangle$, if for $1 \leq i \leq n$, the word x_i is stored in $C[l_i..(l_i + |x_i| - 1)]$, $C[(l_i + |x_i|)] = \Lambda$, and none of the words overlap.

Definition 1.1.10 *A function $f : U \rightarrow I$ is a* perfect hash function *(PHF) for S, if the n-tuple $\langle f(x_1), f(x_2), \ldots, f(x_n) \rangle$ is an address vector for S.*

Note that the above definition applies to both U_f and U_v. A PHF for S can be used to answer the following question.

Given some $x \in U$, does x belong to S?

In the case of fixed-length words, the words in S_f are stored in an array B. To check whether $x \in S_f$, evaluate $f(x)$, retrieve $B[f(x)]$, and compare $B[f(x)]$ with x. In the case of variable-length words, we store the characters of each word of S_v sequentially as a Λ-terminated string in an array of characters C. To check whether $x \in S_v$, evaluate $f(x)$, retrieve $C[f(x)..f(x)+|x|-1]$, and compare with x.

Definition 1.1.11 *f is a* minimal perfect hash function *(MPHF) for S_f if*

$$\max_{1 \le i \le n} f(x_i) = n.$$

f is a minimal perfect hash function *(MPHF) for S_v if*

$$\max_{1 \le i \le n} (f(x_i) + |x_i| + 1) \quad = \quad m.$$

As mentioned above, a PHF f for a dictionary S_f can be used to answer a membership query for S_f by storing the words of S_f in an array B. When f is an MPHF, we minimize the space occupied by the array B. Similarly, when f is an MPHF for S_v, we minimize the space occupied by the array C.

Definition 1.1.12 *A function $f : U \rightarrow I$ is an* order-preserving perfect hash function *(OPPHF) for S to the n-tuple $\langle l_1, l_2, \ldots, l_n \rangle$, if*

$$\langle f(x_1), f(x_2), \ldots, f(x_n) \rangle = \langle l_1, l_2, \ldots, l_n \rangle.$$

Hence, an OPPHF maps S to I in a predetermined order. We abbreviate order-preserving as OP and non-order-preserving as NOP. An OPPHF is useful when the location of the words is predetermined, and we do not have the flexibility of rearranging them. Usually an OPPHF requires more space than a NOPPHF. A NOPPHF is useful in saving space when we are permitted to rearrange the words.

Definition 1.1.13 *If $\mathcal{F}$ is a family of functions such that for every subset S of U there exists a function f in $\mathcal{F}$ that is a PHF for S, $\mathcal{F}$ is called a* perfect hashing scheme *for U.*

For a perfect hashing scheme $\mathcal{F}$ to be practical, given a word $x \in U$ and an f from $\mathcal{F}$, it is important that the evaluation time of $f(x)$ be reasonably small. In practice, functions which can be evaluated in time linear in the length of the word are sufficient [33].

Definition 1.1.14 *Given $\mathcal{F}$, and U, an algorithm for finding a PHF $f \in \mathcal{F}$ for $S \in U$ is called a* perfect hashing algorithm.

Feasible perfect hashing algorithms are almost always probabilistic, and their time complexity is for the expected case. There are three important considerations in judging the merit of a perfect hashing algorithm:

1. How much time does it take to find a PHF $f \in \mathcal{F}$ for $S \subset U$ from $\mathcal{F}$?
2. Assuming a PHF f can be found in reasonable time, how much space is required to specify f?
3. Is there a tradeoff between the time for finding f and the space required to specify f?

In addition to the space required to specify f, we need space to store all the words of S. For some perfect hashing schemes, it may be meaningful to say that a particular data structure is used for specifying f or a particular data structure is used for storing the words in S. Sometimes, it may not be clear whether a particular data structure is used for specifying f or for storing the words in S. For this reason, when comparing two algorithms it is better to compare the total space required for specifying f and for storing the words in S.

Definition 1.1.15 *A* set-function pair *is a 2-tuple $\langle S, f \rangle$, where S is a set of words and f is a PHF for S. The space, SPACE($\langle S, f \rangle$), required to represent a set-function pair $\langle S, f \rangle$, is the total space occupied by all the data structures required to represent S and f.*

Note that SPACE$\langle S, f \rangle$ includes only the data structures used for either evaluating f or representing the words in S. A perfect hashing algorithm may use some temporary data

structures for finding f which are discarded at the end of the algorithm. The space occupied by these data structures is not included in SPACE$(\langle S, f\rangle)$.

Definition 1.1.16 *The space SPACE(f) required to represent a PHF f is the difference between the space required to represent a set-function pair $\langle S, f\rangle$ and the space required to represent S.*

$$\text{SPACE}(f) = \text{SPACE}(\langle S, f\rangle) - \text{SPACE}(S).$$

We also refer to SPACE(f) as the size of f. When comparing two perfect hashing algorithms for a set S, one is interested in the following questions.

1. **Time and space tradeoff**: If an upper bound on the space requirement for the PHF to be found is given, which algorithm runs faster? If a maximum time limit is given, which algorithm is capable of finding a smaller PHF?

2. **Range of Applicability**: What is the smallest PHF each algorithm can produce? What is the minimum time each algorithm requires for producing some PHF?

Definition 1.1.17 *The size of the smallest PHF found by an algorithm is the* lower space limit *(LSL) of the algorithm.*

Definition 1.1.18 *The minimum time required by an algorithm to find some PHF is the* lower time limit *(LTL) of the algorithm.*

Definition 1.1.19 *The size of the PHF found by an algorithm in time LTL is the* upper space limit *(USL) of the algorithm.*

Definition 1.1.20 *The time required by an algorithm to find a PHF of size LSL is the* upper time limit *(UTL) of the algorithm.*

The above definitions are illustrated in Figure 1.1. Usually, we are more interested in LSL and LTL than USL and UTL.

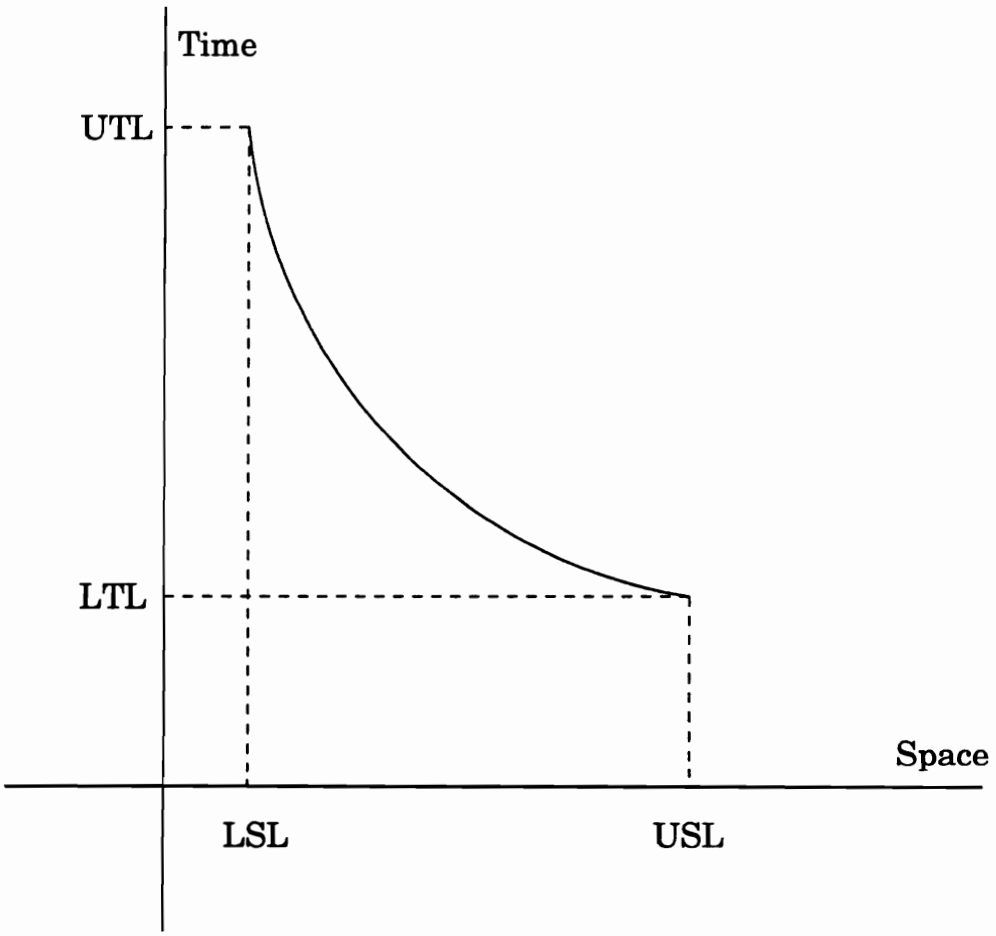


Figure 1.1: A plot of the time and space tradeoff of a typical perfect hashing algorithm.

Definition 1.1.21 *Algorithm 2 runs faster than algorithm 1 in the range* (R_1, R_2) *if*

- *Both algorithm 1 and algorithm 2 can find a PHF of size* R, *for* $R_1 \leq R \leq R_2$
- *The time required by algorithm 2 is always less than the time required by algorithm 1 within that range.*

Figure 1.2 shows a typical plot of two perfect hashing algorithms. In the range (R_1, R_2), algorithm 2 runs faster than algorithm 1.

1.1.1 Lower bound on the size of a PHF

At present, no nontrivial lower bound is known on SPACE(f) where f is a PHF for S_v. Mehlhorn [32] proves that $\Omega(n^2/\tilde{n})$ bits are required to represent a PHF f for S_f where

$$\max_{1 \leq i \leq n} f(x_i) = \tilde{n}.$$

Additionally, when f is an MPHF, i.e., $\tilde{n} = n$, he proves that at least $n/(\ln 2)$ bits are required to represent f. Fox et al. [15, 16, 17] prove that at least $n \log_2 n + O(1)$ bits are required to represent a minimal OPPHF for S_f.

Definition 1.1.22 *Suppose* f *is a PHF for* S_f. *Then the space efficiency for* f *is*

$$\eta(f) = \frac{n/\ln 2}{\text{SPACE}(f)}.$$

One can think of η as the efficiency of representing a PHF. If SPACE(f) matches the lower bound, then η is 1. and it decreases as the space occupied by the PHF increases. Ideally, η should be $\Theta(1)$. All the known algorithms for NOPPHFs, with the exception of Fox et al. [18], have an efficiency η of $O(1/\log n)$.

1.2 Representation of a Pointer

When implementing a perfect hashing algorithm, often one stores S in an array C and accesses each word by a pointer. A major fraction of the storage of many perfect hashing

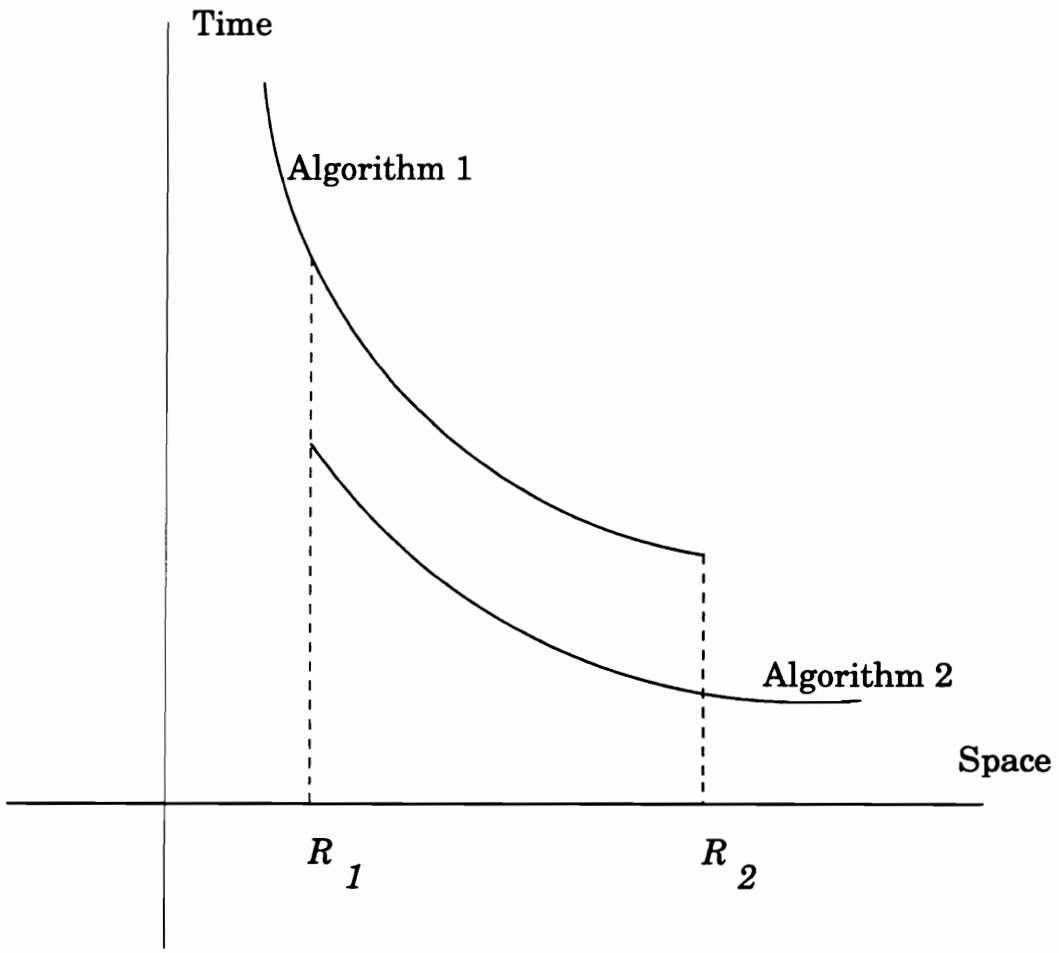


Figure 1.2: Algorithm 2 runs faster than algorithm 1 in the range (R_1, R_2).

algorithms consists of pointers. Thus, it is extremely important to use the correct type of pointer to achieve minimal space. In this section, we discuss two types of pointers and their usefulness.

Let $\mathcal{H}^{(\hat{m})}$ be a family of functions that maps a universe U to a range $I_{\hat{m}}$. A function $h^{(\hat{m})} \in \mathcal{H}$, picked randomly from $\mathcal{H}^{(\hat{m})}$, is called a *random function* if all the random variables $h^{(\hat{m})}(x)$, $x \in U$ are independent and identically distributed. If the probability that $h^{(\hat{m})}(x) = i$ is the same for all $i \in I_{\hat{m}}$, then $h^{(\hat{m})}$ is a *random function with uniform distribution.*

Let $\mathcal{HI}^{(\hat{m})}$ be a family of functions that maps a universe $U \times I$ to a range $I_{\hat{m}}$. A function $h^{(\hat{m})} \in \mathcal{H}$, picked randomly from $\mathcal{H}^{(\hat{m})}$, is called an *indexed random function* if all the random variables $h^{(\hat{m})}(x, j)$, $x \in U$ and $j \in I$ are independent and identically distributed. If the probability that $h^{(\hat{m})}(x, j) = i$ is the same for all $i \in I_{\hat{m}}$, then $h^{(\hat{m})}$ is an *indexed random function with uniform distribution.*

Let $\mathcal{HI}^{(\hat{m})}$ be a family of functions that maps a universe $U \times I$ to a range $I_{\hat{m}}$, and for every $x \in U$ and $i \in I_{\hat{m}}$ there exists a $j \in I_{\hat{m}}$ such that $h(x, j) = i$. A function $h^{(\hat{m})} \in \mathcal{H}$, picked randomly from $\mathcal{H}^{(\hat{m})}$, is called an *invertible indexed random function* if all the random variables $h^{(\hat{m})}(x, j)$, $x \in U$ and $j \in I$ are independent and identically distributed. If the probability that $h^{(\hat{m})}(x, j) = i$ is the same for all $i \in I_{\hat{m}}$, then $h^{(\hat{m})}$ is an *invertible indexed random function with uniform distribution.*

Normally, we denote the size of the range of a random function with a superscript, e.g. $h^{(\hat{m})}$. However, when the size of the range is clear from the context we drop this superscript. Also, we assume that the random functions used in this abstract have uniform distribution.

1.2.1 Simple pointer

A simple pointer is the index of a word stored in an array, represented in binary. If a word x is stored in $C[l..l + |x|]$, the pointer to x is the integer l, and it occupies $\lceil \log_2 m \rceil$ bits. This representation is most useful for OP perfect hashing. However, when there is flexibility of rearranging the words, this representation is too expensive.

1.2.2 Hash pointer

A hash pointer is a representation that takes advantage of the nature of random functions to conserve space. This representation is most useful for NOP perfect hashing, where there is flexibility of rearranging the words. It consists of two parts, an approximate location given by a random function h_0, and an offset from the approximate location given by an integer *Offset*. A random function h_0 is used for the approximate location. The formula *Address*(x) for calculating the location of x would be

$$Address(x) \quad = \quad h_0(x) + Offset(x).$$

The number of bits required for representing a hash pointer depends on the range of *Offset*(x).

Let ξ be a permutation of I_n^+ such that if $i < j$, $h_0(x_{\xi(i)}) \leq h_0(x_{\xi(j)})$. The words of S are arranged in the array C according to the permutation ξ. The address of $x_{\xi(i)}$ in C is given by

$$Address(x_{\xi(i)}) \quad = \quad \sum_{j=1}^{i-1} |x_{\xi(j)}| + 1.$$

Example 1.2.1 Let U_v be a set of all the words in the English language. Let S_v be a set of the following five words:

$$S_v \quad = \{\text{“Physics”, “Chemistry”, “Mathematics”, “Biology”, “Economics”}\}$$

The total number of characters in all the words is 43. The characters of each word are stored in the array C sequentially, and each word is terminated by Λ. Hence, $m = 48$. For addressing each word directly with a simple pointer, we need $\lceil \log_2 48 \rceil = 6$ bits per pointer.

Instead, suppose we use hash pointers with a random function h_0. Suppose h_0 maps these words as given in Table 1.1. These five words are placed in C according to the partial order induced by h_0. Of the five words, two words, “Physics” and “Biology”, are mapped to the same cell 32. Arbitrarily, we decide to put “Biology” before “Physics”. The last two columns in Table 1.1 lists the *Address* and *Offset* of each variable. The range of *Offset* is

x	$\|x\|+1$	$h_0(x)$	$Address(x)$	$Offset(x)$
Physics	8	32	30	-2
Chemistry	10	21	12	-9
Mathematics	12	8	0	-8
Biology	8	32	22	-10
Economics	10	41	38	-3

Table 1.1: An example illustrating a hash pointer.

from -2 to -10. So, totally $\lceil \log_2 9 \rceil = 4$ are required for a hash pointer, a savings over the 6 bits required for a simple pointer. □

The number of bits required to represent $Offset(x)$ is given by

$$\left\lceil log_2 \left[\max_{1 \le i \le n} \left\{ h_0(x_{\xi(i)}) - Address(x_{\xi(i)}) \right\} - \min_{1 \le i \le n} \left\{ h_0(x_{\xi(i)}) - Address(x_{\xi(i)}) \right\} \right] \right\rceil .$$

Clearly, this depends on the distribution of the lengths of the words. We assume that the words of S_v are chosen randomly from U_v and the statistics of their lengths are described by a random variable L. In Lemma 1.2.2, we prove that, under certain reasonable assumptions on the mean and variance of L, $Address(x_{\xi(i)})$ is $(i-1)m/n + O(\sqrt{n \ln n})$ with high probability. In Lemma 1.2.3 we prove that $h_0(x_{\xi(i)})$ is $(i + O(\sqrt{n \ln n}))m/n$ with high probability. Together, Lemmas 1.2.2 and 1.2.3 prove that the maximum value of *Offset* is $O(\sqrt{n \ln n})m/n$. Assuming m/n, the average length of a word in U, is $O(\ln n)$, a hash pointer requires approximately $\lceil \log_2 n \rceil / 2$ bits. To prove Lemmas 1.2.2 and 1.2.3, we need Kolmogorov's inequality[38] and Lemma 1.2.1.

Theorem 1.2.1 *(Kolmogorov's Inequality) For $1 \le i \le n$, let Y_i be a sequence of mutually independent random variables with $E[Y_i] = 0$ and $E[Y_i^2] < \infty$. Let*

$$\tilde{Y}_i = \sum_{r=1}^{i} Y_r .$$

Then, for every $\epsilon > 0$,

$$\Pr\left[\max_{1 \le r \le i} \left| \tilde{Y}_r \right| \ge \epsilon \right] \quad < \quad \epsilon^{-2} E\left[\tilde{Y}_i^2 \right] .$$

Lemma 1.2.1 *If a random variable X is binomially distributed with a mean of μ and a variance of σ^2, then for every $\epsilon > 0$*

$$\Pr[|X - \mu| \geq \epsilon\sigma] \quad < \quad \epsilon^{-1}e^{-\epsilon^2/2},$$

where ϵ is a constant satisfying the inequality $\epsilon < \min\{\sigma/10, \mu^{2/3}/2\}$.

Proof: Refer to Bollobas [2], page 11. □

Lemma 1.2.2 *Let L be a random variable which denotes the length of a word chosen randomly from U. If L has mean m/n and variance $\sigma^2 = o(\ln n)$, then the probability that*

$$Address(x_{\xi(i)}) \quad = \quad \frac{(i-1)m}{n} + O(\sqrt{n \ln n})$$

is $1 - o(1)$.

Proof: For $1 \leq i \leq n$, let Y_i be a random variable that equals $|x_{\xi(i)}| - m/n$. The expectation of Y_i is 0, and the variance of Y_i is σ^2. For $1 \leq i \leq n$, let

$$\tilde{Y}_i = \sum_{r=1}^{i} Y_r.$$

$Address(x_{\xi(i)})$ can be expressed in terms of $\tilde{Y}_i$ as,

$$Address(x_{\xi(i)}) \quad = \quad 1 + \frac{(i-1)m}{n} + \tilde{Y}_{i-1}.$$

According to Kolmogorov's inequality,

$$\Pr\left[\max_{1 \leq r \leq i-1} \left|\tilde{Y}_r\right| \geq \epsilon\right] \quad < \quad \epsilon^{-2} E\left[\tilde{Y}_{i-1}^2\right].$$

Since all the Y_r are mutually independent, the expectation of $\tilde{Y}_{i-1}^2$ is

$$\begin{aligned} E\left[\tilde{Y}_{i-1}^2\right] &= \sum_{r=1}^{i-1} E[Y_r^2] \\ &= (i-1)\sigma^2. \end{aligned}$$

Hence,

$$\Pr\left[\max_{1\le r\le i-1}\left|\tilde{Y}_r\right| \ge \epsilon\right] \;<\; \epsilon^{-2}(i-1)\sigma^2.$$

Substituting $\epsilon = \sqrt{n\ln n}$, and noting that σ^2 is $o(\ln n)$, we obtain

$$\begin{aligned}\Pr\left[\max_{1\le r\le i-1}\left|\tilde{Y}_r\right| \ge \sqrt{n\ln n}\right] &< \frac{(i-1)o(\ln n)}{n\ln n}\\ &= o(1).\end{aligned}$$

This implies that with a probability $1-o(1)$,

$$Address(x_{\xi(i)}) \;=\; \frac{(i-1)m}{n} + O(\sqrt{n\ln n}).$$

□

Lemma 1.2.3 *For* $1 \le i \le n$,

$$h_0(x_{\xi(i)}) \;=\; \frac{m}{n}\left(i + O(\sqrt{n\ln n})\right).$$

Proof: For $1 \le l \le m$, let X_l be a random variable that equals the number of words for which $h_0(x) \le l$.

$$\Pr[X_l = k] \;=\; \binom{n}{k}\left(\frac{l}{m}\right)^k\left(1-\frac{l}{m}\right)^{n-k}.$$

The random variable X_l is binomially distributed with mean $\mu_l = nl/m$ and variance $\sigma_l^2 = nl(m-l)/m^2$. According to Lemma 1.2.1,

$$\Pr[|X_l - \mu_l| \ge \epsilon\sigma_l] \;<\; \epsilon^{-1}e^{-\epsilon^2/2}.$$

The maximum value of σ_l^2 is $n/4$ for $l = m/2$. This implies that

$$\Pr\left[|X_l - \mu_l| \ge \frac{\epsilon\sqrt{n}}{2}\right] \;<\; \epsilon^{-1}e^{-\epsilon^2/2}.$$

Substituting $im/n - m\epsilon\sqrt{n}/2n$ for l,

$$\begin{aligned}\Pr\left[\left|X_{im/n-m\epsilon\sqrt{n}/2n} - \mu_{im/n-m\epsilon\sqrt{n}/2n}\right| \ge \frac{\epsilon\sqrt{n}}{2}\right] &< \epsilon^{-1}e^{-\epsilon^2/2}\\ \Pr\left[\left|X_{im/n-m\epsilon\sqrt{n}/2n} - \left(i - \frac{\epsilon\sqrt{n}}{2}\right)\right| \ge \frac{\epsilon\sqrt{n}}{2}\right] &< \epsilon^{-1}e^{-\epsilon^2/2}\\ \Pr\left[X_{im/n-m\epsilon\sqrt{n}/2n} \ge i\right] &< \epsilon^{-1}e^{-\epsilon^2/2}.\end{aligned}$$

This implies that

$$\Pr\left[h_0(x_{\xi(i)}) < \frac{im}{n} - \frac{m\epsilon\sqrt{n}}{2n}\right] < \epsilon^{-1}e^{-\epsilon^2/2}.$$

Similarly we can prove that

$$\Pr\left[h_0(x_{\xi(i)}) > \frac{im}{n} + \frac{m\epsilon\sqrt{n}}{2n}\right] < \epsilon^{-1}e^{-\epsilon^2/2}.$$

Combining both inequalities,

$$\Pr\left[|h_0(x_{\xi(i)}) - im/n| > \frac{m\epsilon\sqrt{n}}{2n}\right] < 2\epsilon^{-1}e^{-\epsilon^2/2}$$

For $1 \leq i \leq n$, the probability that the maximum of $|h_0(x_{\xi(i)}) - im/n|$ is greater than $m\epsilon\sqrt{n}/2n$ can be bounded be multiplying the right hand side by n,

$$\Pr\left[\max_{1\leq i\leq n} |h_0(x_{\xi(i)}) - im/n| > \frac{m\epsilon\sqrt{n}}{2n}\right] < 2n\epsilon^{-1}e^{-\epsilon^2/2}.$$

Substituting $\epsilon = \sqrt{2\ln n}$,

$$\begin{aligned}\Pr\left[\max_{1\leq i\leq n}\left|h_0(x_{\xi(i)}) - \frac{im}{n}\right| > \frac{m}{n}\sqrt{\frac{n\ln n}{2}}\right] &< \frac{2n}{\sqrt{2\ln n}}e^{-2\ln n/2} \\ &= \sqrt{\frac{2}{\ln n}} \\ &= o(1).\end{aligned}$$

Hence, the lemma follows. □

Theorem 1.2.2 *Let L be a random variable that denotes the length of a word selected randomly from U. Let the mean of L be m/n. If both the mean and the variance of L are $o(\ln n)$, then the expected size of a hash pointer is $\lceil \log_2 n \rceil/2 + O(\log\log n)$ bits.*

Proof: From Lemma 1.2.2, for $1 \leq i \leq n$,

$$Address(x_{\xi(i)}) = \frac{(i-1)m}{n} + O(\sqrt{n\ln n}).$$

From Lemma 1.2.3, for $1 \leq i \leq n$,

$$h_0(x_{\xi(i)}) = \frac{m}{n}(i + O(\sqrt{n \ln n})).$$

So, for $1 \leq i \leq n$,

$$\begin{aligned} \max_{1 \leq i \leq n} \left| Address(x_{\xi(i)}) - h_0(x_{\xi(i)}) \right| &= \frac{m}{n} O(\sqrt{n \ln n}) \\ \log_2 \max_{1 \leq i \leq n} \left| Address(x_{\xi(i)}) - h_0(x_{\xi(i)}) \right| &= \log_2 \left| \frac{m}{n} O(\sqrt{n \ln n}) \right|. \end{aligned}$$

Since m/n is $o(\ln n)$,

$$\begin{aligned} \log_2 \max_{1 \leq i \leq n} \left| Address(x_{\xi(i)}) - h_0(x_{\xi(i)}) \right| &= \log_2 \left| \frac{m}{n} O(\sqrt{n \ln n}) \right| \\ &= \log_2 n + O(\log \log n). \end{aligned}$$

So, a hash pointer requires approximately $\lceil \log_2 n \rceil / 2 + O(\log \log n)$ bits.

□

To further verify the conclusion of Theorem 1.2.2, an experiment was performed for NOP perfect hashing on sets of variable-length words. The words of the dictionary are generated randomly with the average length of the word as 6. The results of the experiment are tabulated in Table 1.2. The size of the hash pointer is consistently half the size of a simple pointer as expected.

1.3 Previous Work on Perfect Hashing

We divide the previous work on perfect hashing into three categories—*early approaches, static perfect hashing algorithms,* and *dynamic perfect hashing algorithms.* Section 1.3.1 discusses the early approaches. These algorithms work only when the dictionary is small. Section 1.3.2 discusses the perfect hashing algorithms which work for large but static dictionaries. Section 1.3.3 discusses algorithms which work for dynamic dictionaries.

n	Minimum Offset	Maximum Offset	Range of Offsets	Hash Pointer Size	Simple Pointer Size
10000	29	-128	157	8	14
20000	100	-98	198	8	15
30000	61	-69	130	8	15
40000	238	-62	300	9	16
50000	82	-159	241	8	16
60000	256	-72	328	9	16
70000	127	-261	388	9	17
80000	175	-68	243	8	17
90000	195	-109	304	9	17
100000	54	-284	338	9	17
200000	437	-199	636	10	18
300000	416	-159	575	10	19
400000	686	-215	901	10	19
500000	501	-53	554	10	19
600000	875	-320	1195	11	20
700000	803	-327	1130	11	20
800000	736	-518	1254	11	20
900000	174	-722	896	10	20
1000000	487	-418	905	10	20

Table 1.2: Comparison of the space requirements of simple and hash pointers.

1.3.1 Early approaches

In this section, the early approaches are categorized by author and are presented in a chronological order.

Sprugnoli

Sprugnoli [39] proposes one of the first approaches for finding PHFs. He assumes that the words of U are all integers, i.e., U is a universe of fixed-length words. He gives two methods for finding PHFs, the *quotient reduction* method and the *remainder reduction* method. In the quotient reduction method a PHF f of the following form is sought:

$$f(x) = \left\lfloor \frac{x + D}{E} \right\rfloor .$$

D and E are integers whose values depend on S. In the remainder reduction method, a PHF f of the following form is sought:

$$f(x) = \left\lfloor \frac{(Dx + E) \bmod F}{G} \right\rfloor .$$

D, E, F, and G are integers whose values depend on S. Both algorithms work in time exponential in n, and neither is guaranteed to produce an MPHF.

Jaeschke

Jaeschke [26] proposes a scheme called *reciprocal hashing.* It is similar to Sprugnoli's quotient reduction method, but is guaranteed to find a PHF. He seeks a PHF of the form:

$$f(x) = \left\lfloor \frac{D}{Ex + F} \right\rfloor .$$

He proves that for any given set S, some D, E and F can always be found producing an MPHF f. However, it takes time exponential in n to find them.

Chang

Chang [4, 5] modifies Jaeschke's method and obtains an $O(n^2 \log n)$ algorithm. Unlike Sprugnoli and Jaeschke, Chang assumes that U is a universe of variable-length words. His algorithm is called the *letter oriented reciprocal hashing scheme.* He assumes that for every given set S, there exist two integers i and j such that the tuple $\langle x[i], x[j] \rangle$ is distinct for all words $x \in S$. This assumption is obviously very restrictive. He seeks a PHF of the form:

$$f(x) = D[x[i]] + \left\lfloor \frac{E[x[i]]}{F[x[j]]} \right\rfloor \bmod G[x[j]],$$

where D, E, F and G are arrays of length $|\Psi|$(recall that Ψ is the alphabet used to build words). The algorithm consists of finding some arrays D, E, F and G such that f is a PHF. Like the previous methods, this one works only for small S. The author gives an example of $n = 36$ for which this method produces an MPHF.

Cichelli

Cichelli [8] gives an algorithm which has a better probability of success than either Jaeschke's or Sprugnoli's. He seeks a PHF of the form:

$$f(x) = |x| + D[x[1]] + D[x[|x|]],$$

where D is an array of length $|\Psi|$. The algorithm consists of finding an array D such that f is a PHF. This method does not always succeed and may take time exponential in n when it succeeds. It works well only for small n, approximately 40.

Cercone

Cercone [3] generalizes and improves Cichelli's algorithm. He runs this algorithm on a set of 500 words. However, he does not provide an analysis of his algorithm, and there is no indication that it works for larger sets.

1.3.2 Static perfect hashing algorithms

Recently, several algorithms have appeared which follow the MOS model. We define the MOS model precisely in Chapter 2. In this section, we discuss briefly the MOS algorithms.

Sager [35] modifies Cichelli's [8] algorithm and runs it on sets of size up to 200. For a set of n words, he seeks an MPHF of the form

$$f(x) = (h_a(x) + A[h_b(x)] + A[h_c(x)]) \bmod n \tag{1.1}$$

where

$h_a : U \rightarrow I_n$	is a random function,
$h_b, h_c : U \rightarrow I_N$	are random functions,
A	is an array of integers of size N.

Sager expresses the problem of computing the array A as a graph problem. Sager's algorithm requires a space of $O(n \log n)$ bits to represent the MPHF and runs in an expected time of $O(n^4)$. Fox et al. [20] implement an improved version of Sager's algorithm and runs it on sets with n up to 500. This algorithm requires a space of $O(n \log n)$ bits and runs in an expected time of $O(n^3)$.

Fox et al. [21] modify Sager's method and obtain an algorithm that requires a space of $O(n \log n)$ bits. They claim an expected running time of $O(n \log n)$, but they report only their experimental results and do not provide any analysis. They seek an MPHF of exactly the same form as in Equation 1.1, but use a different algorithm for computing the array A.

Fox et al. [22] further modify this algorithm. They seek an MPHF f of the form:

$$f(x) = \begin{cases} (h_b(x)A[h_a(x)] + h_c(x)(A[h_a(x)])^2) \pmod n & \text{if } Mark[h_a(x)] = 1 \\ A[h_a(x)] & \text{if } Mark[h_a(x)] = 0 \end{cases}$$

where

A	is an array of integers of size N,
$Mark$	is an array of N bits.
$h_a : U \rightarrow I_N$	is a random function,
$h_b, h_c : U \rightarrow I_n$	are random functions,

On a Sequent machine, they were able to run this algorithm on a dictionary of 3.8 million words in 9 hours.

Fox et al. [18] present another algorithm that conceptually splits the A array into two parts based on a constant $c, 0 < c < 1$. The two parts are $A[0 \ldots cN-1]$ and $A[cN \ldots N-1]$. A random function $h_1 : U \rightarrow I_N$ is selected such that it maps to an index in $[0 \ldots cN-1]$ with the probability p and to an index in $[cN \ldots N-1]$ with a probability $1-p$, where $0 < p < 1$. Hence h_1 is biased so as to skew the distribution of the number of words mapped to indices. The form of the MPHF sought is:

$$f(x) = h_2(x, A[h_1(x)]),$$

where

A	is an array of integers of size N.
$h_1 : U \rightarrow I_N$	is a random function with a probability of p to $\{0 \ldots cN-1\}$ and $1-p$ to $\{cN \ldots N-1\}$, and $0 < c < 1, 0 < p < 1$,
$h_2 : U \times I \rightarrow I_n$	is an invertible indexed random function,

On a NeXT machine, they were able to run this algorithm on a dictionary of 3.8 million words in 6 hours.

Fox et al. [6, 15, 16] present an OP perfect hashing algorithm. They seek an MPHF of the form:

$$f(x) = A[(h_a(x) + A[h_b(x)] + A[h_c(x)]) \bmod N] \bmod n$$

where

$h_a, h_b, h_c : U \rightarrow I_n$	are random functions,
A	is an array of integers of size N

Fredman et al. [23] propose the first algorithm for finding a PHF in deterministic polynomial time. We refer to this algorithm as the FKS algorithm. Their algorithm consists of two steps. The first step maps all the words of S to an array of size n. The second

step maps these words onto an array of size $3n$. In the worst case, this algorithm could take $O(n|U_f|)$ time. They give another variation of their method in which they can obtain a PHF in $O(n^3 b_f)$ worst-case time and $O(n)$ average-case time by increasing the size of the second array to $6n$. Cormack et al. [9] suggest a randomized version of the FKS algorithm to improve the expected running time to $O(n)$ and space to $O(n \log n)$ bits. Gonnet et al. [24, 30, 34] suggest a similar algorithm with emphasis on storing the data on external storage.

Table 1.3 provides a comparison of the different static perfect hashing algorithms using relevant results reported in the literature. In general, the worst-case time for any of the probabilistic algorithms is exponential. A part of this thesis consists of the analysis of the Fox et al.[18, 22] algorithms. As listed in the last two lines of Table 1.3, we prove in Theorem 2.2.7 that the expected time required by the MOS algorithm is $\Omega(n^{1+\eta})$ to find an MPHF of size $n/(\eta \ln 2)$ bits.

1.3.3 Dynamic perfect hashing algorithms

Aho and Lee [1] study the extension of perfect hashing to dynamic sets. However, their algorithm places a bound on n, and it fails when n exceeds the bound. In contrast, the dynamic perfect hashing algorithm that we present in Chapter 5 degrades gracefully. Dietzfelbinger et al. [12] give a dynamic perfect hashing algorithm based on the FKS algorithm. Their algorithm places no bounds on n, and it always uses $O(nb_f)$ space when there are n words in the dictionary. However, their algorithm works only for fixed-length words, and there is no straightforward way to extend it to variable-length words. Enbody and Du [13] give a survey of currently available dynamic hashing techniques. In recent work, Daoud [11] gives a dynamic perfect hashing algorithm based on the *extendible hashing* [13] paradigm that is empirically at least as successful as any other algorithm.

Author	Expected Time	Worst Case Time	Space in bits	Guaranteed to Work?
Sprugnoli [39]	$O(2^n)$			no
Jaeschke [26]	$O(2^n)$	$O(2^n)$		yes
Chang [4, 5]	$O(n^2 \log n)$			no
Cichelli [8]	$O(2^n)$			no
Cercone [3]	No Analysis			no
Sager [35]	$O(n^4)$		$O(n \log n)$	yes
Fox et al. [19]	$O(n^3)$		$O(n \log n)$	yes
Cormack, [9] Horsepool, Kaiserworth	$O(n)$		$O(n \log n)$	yes
Fredman, [23] Komlos, Szemeredi,	$O(n)$	$O(n^3 \log \lvert U \rvert)$	$O(n \log n)$	yes
Fox et al. [21]	$O(n \log n)$		$O(n \log n)$	yes
Fox et al. [16]	No Analysis		$O(n \log n)$	yes
Fox et al. [22]	$O(n^{1+\eta})$		$n/(\eta \ln 2)$	yes
Fox et al. [18]	$O(n^{1+\eta})$		$n/(\eta \ln 2)$	yes

Table 1.3: Comparison of different perfect hashing algorithms.

1.4 Survey of Results

The remainder of this dissertation consists of 5 chapters. Recently several *mapping, ordering, and searching* (MOS) algorithms have been proposed [35, 18, 19, 20, 21, 22]. Chapter 2 defines a model called the MOS model which unifies all these algorithms and provides an analysis and experimental results of these algorithms. Chapter 3 discusses another perfect hashing algorithm for static sets called the *musical chairs (MC)* algorithm. The MC algorithm uses the concept of ordering the edges of a hypergraph. Chapter 4 investigates the computational complexity of a graph ordering problem motivated by the hypergraph ordering problem in the MC algorithm. Chapter 5 extends the MOS model to dynamic perfect hashing. Chapter 6 summarizes the results of this dissertation and discusses directions for future research.

Given an S, one can find a PHF f for it in several ways. Fox et al. [18, 21, 22] follow the MOS model, and they assume that the PHFs they produce occupy $\Theta(n)$ bits. They measure their results experimentally and report that as the number of bits per word is decreased, the running time increases. In this dissertation, we analyze the performance of the MOS model and derive the precise relationship between running time and the space required by the function. We show that, to find an MPHF occupying $n/(\eta \ln 2)$ bits under the MOS model, the expected time taken by the algorithm is at least $\Omega(n^{1+\eta})$. This implies that an algorithm following the MOS model cannot find MPHFs occupying $O(n)$ bits in $O(n)$ expected time. Chapter 2 is organized as follows. Section 2.1 defines a model called the MOS model which unifies all the MOS algorithms. Section 2.2 analyzes the running time and space requirements of perfect hashing algorithms under the MOS model. Section 2.3 discusses some experimental results and shows how our analysis can help predict the time and space requirements for perfect hashing under the MOS model.

In Chapter 3, we propose an algorithm called the *musical chairs* (MC) algorithm. We provide experimental results of the MC algorithm and compare it with the MOS algorithm. Both the MOS and MC algorithms can be applied to either S_f (fixed-length words) or S_v

(variable-length words). For both S_f and S_v, either in the OP case or NOP case, and in its entire range, the MC algorithm runs faster than the MOS algorithm. The LSL of the MOS algorithm is lower than the LSL of the MC algorithm in all the cases. However, as we prove in Chapter 2, running the MOS algorithm at its LSL (Definition 1.1.17) is highly impractical. In practice, the space required by the smallest MPHF found by either algorithm is very close in all the cases.

Chapter 3 is organized as follows. Section 3.1 gives a description of the MC algorithm. Section 3.2 develops a basis for comparison of the space requirements of the MOS and MC algorithms in various cases. Section 3.3 gives experimental results of the MC algorithm and compares them the experimental results of the MOS algorithm. Section 3.4 discusses possible avenues for further research on the MC algorithm.

In the MC algorithm, we map each word x of S to a unique triple $\langle h_1(x), h_2(x), h_3(x)\rangle$ using random functions h_1, h_2, and h_3. These unique triples can be considered as the edges of a hypergraph (the generalization of a graph in which an edge may contain more than two vertices). An important step in the algorithm is ordering these edges. Chapter 4 examines a similar problem of ordering the vertices of an ordinary graph. The problem can be stated as follows. Given a graph G of n vertices, E edges, and two integers r and s, can the vertices of G be ordered such that at most r edges go backward and s edges go forward? We prove that this problem is NP-complete. However, it has a polynomial-time solution when either r or s is one. Chapter 4 is organized as follows. Section 4.1 defines notation and terminology. Section 4.2 gives a polynomial-time solution for the problem when r or s is equal to 1. Section 4.3 reduces the graph problem to NOT ALL EQUAL 3-SAT [36] and proves that it is NP-complete.

In Chapter 5, we propose a perfect hashing algorithm called the *dynamic perfect hashing* (DPH) algorithm that applies to dynamic dictionaries. While a static dictionary supports only the *find* operation, a dynamic dictionary supports *find, insert,* and *delete* operations. One of the main issues in dynamic perfect hashing is whether the space for the perfect hashing should be allocated statically or dynamically. Static allocation is appropriate when

the expected size of the dictionary is known, and variance in dictionary size is small. Dynamic allocation is applicable to a much wider variety of dynamic dictionaries than static allocation. However, dynamic allocation of space requires remapping the words whenever space for the PHF is reallocated.

Chapter 5 is organized as follows. Section 5.1 describes the dynamic perfect hashing algorithm. In Section 5.2, a general formula for the cost of an insertion or a deletion is derived. In Section 5.3, we model the dictionary as a birth and death process [27]. We assume a static allocation of memory and consider the performance of two kinds of queueing models, $M/M/1$ and $M/M/\infty$. We prove that if we allocate an appropriate amount of space for the PHF, both insertions and deletions require $O(1)$ expected time. In Section 5.4, we analyze the performance of the DPH algorithm when space for the PHF is allocated dynamically. Since remapping of the dictionary after every insertion and deletion would be very expensive, we discuss a strategy for remapping the dictionary after several insertions and deletions, amortizing the cost of remapping. With this amortization strategy we show that the expected costs of insertion and deletion for the DPH algorithm with dynamic allocation of space is $O(1)$.

In Chapter 6, we conclude with open problems and suggestions for further research.

Chapter 2

ANALYSIS OF MOS PERFECT HASHING

Given a dictionary S, one can find a PHF f for it in several ways. Recently several algorithms called *mapping, ordering, and searching (MOS)* algorithms have been proposed [18, 19, 20, 21, 22, 35]. In this chapter, we define the MOS model and unify all these algorithms. The MOS algorithms follow this model very closely. Fox et al. [18, 21, 22] assume that the PHFs they produce occupy $\Theta(n)$ bits. They measure their results experimentally and report that as the number of bits per word is decreased, the running time increases. In this chapter, we analyze the performance of the MOS model and determine the precise tradeoff between the running time and the space required by the function. We show that, to find an MPHF under the MOS model occupying $n/(\eta \ln 2)$ bits, the expected time taken by the algorithm is at least $\Omega(n^{1+\eta})$. This implies that, under the MOS model, we cannot find MPHFs occupying $O(n)$ bits in $O(n)$ expected time.

In this chapter, we assume that U denotes a universe of fixed-length words, and S denotes a set of fixed-length words, unless explicitly mentioned otherwise. Extension of the MOS algorithm to a set of variable-length words is straightforward. This chapter is organized as follows. Section 2.1 defines a model called the MOS model which unifies all the MOS algorithms. Section 2.2 analyzes the running time and space requirements of perfect hashing algorithms under the MOS model. Section 2.3 discusses some experimental results and shows how our analysis can help predict time and space requirements for perfect hashing under the MOS model.

2.1 The MOS Model

In this section, we describe the *mapping, ordering,* and *searching (MOS)* model. S is a set of n words from a universe U. A is an array of N non-negative integers. As defined in Section 1.1, B is an array of $\tilde{n}$ words.

Definition 2.1.1 *α is the ratio of n to N:*

$$\alpha = \frac{n}{N}.$$

α is typically $o(\log n)$. B is empty initially and is filled with words from S by the MOS algorithm.

Definition 2.1.2 *β is the fraction of free cells in B after all words of S have been assigned to cells of B:*

$$\beta = \frac{\tilde{n}}{n} - 1.$$

β is typically $O(1)$. For minimum perfect hashing $\beta = 0$. The PHF sought is of the form

$$f(x) = h_2(x, A[h_1(x)]) \tag{2.1}$$

where

$h_1 : U \rightarrow I_N$	is a random function,
$h_2 : U \times I \rightarrow I_{\tilde{n}}$	is an invertible indexed random function,
A	is an array of positive integers of size N.

The definitions of random function and invertible indexed function are given in Section 1.2. The values of $A[0], A[1], \ldots, A[N-1]$ are not known at the beginning of the algorithm. The perfect hashing algorithm consists of finding values for $A[0], A[1], \ldots, A[N-1]$ such that f is a PHF. The perfect hashing algorithm may be viewed as a process of obtaining a solution to Equation 2.1. Note that there can be multiple solutions satisfying Equation 2.1. An MOS algorithm gives one approach to obtaining a solution.

The form of the PHF in Equation 2.1 encapsulates the PHF used in Equation 1.2 by Fox et al. in [22]. The form of the PHF used by Fox et al. in [18] is similar to Equation 2.1, except

that the function h_1 used in [18] is not uniformly random. However, as our experimental results show in Section 2.3.2, our analysis predicts the performance of this algorithm also with a very high accuracy. The form of the PHF used in [21] is different from Equation 2.1. However, the MOS algorithm described in [21] resembles the MOS algorithm that we are about to describe. (See Figure 2.1). It is reasonable to hypothesize that our analysis predicts the performance of the algorithm in [21] also with a high accuracy.

Definition 2.1.3 *For* $1 \leq i \leq N$,

$$Q_i \quad = \quad \{x : x \in S \text{ and } h_1(x) = i\},$$

Q_i is the set of words mapped by h_1 to cell i of A. Intuitively, we associate a bin with each Q_i, and think of h_1 as a function which tosses the words of S randomly into these bins. So we refer to Q_i as *bin* i also. In the actual implementation, a bin is usually represented as a linked list. After f is constructed, these linked lists are discarded, and only the array A is kept. Initially, all the cells of B are empty. After the MOS algorithm is run, the cells of B are filled with words in S.

An MOS algorithm can be broadly divided into the three steps of mapping, ordering, and searching.

Mapping: Note that for the form of f in Equation 2.1, $f(x)$ depends on only one cell of A, that of $A[h_1(x)]$. We can view this fact as each x being mapped to bin $h_1(x)$. So we call this step the *mapping* step. The random function h_1 maps a word $x \in S$ to the bin $h_1(x)$. Since h_1 is a random function, it maps x to any bin with equal probability. The average number of words per bin is clearly $n/N = \alpha$.

Ordering: Even though $f(x)$ for a given $x \in S$ depends only on one cell of A, the converse is not true. h_1 maps the words of S randomly into the bins, but it does not ensure that all bins receive the same number of words. We define the *size* of a bin i as the cardinality of Q_i. The ordering step consists of sorting the bins in descending order by size. The sorting step produces a permutation ξ of I_N^+. After sorting, the sizes of the bins $\xi(1), \xi(2), \ldots, \xi(N)$ are in descending order.

Searching: After the ordering step, the algorithm maps the words in the bins to B according to the procedure *Search* in Figure 2.1. The procedure *Search* has two cases, a non-minimal case (β is $\Omega(1)$) and a minimal case (β is $o(1)$). In the non-minimal case, the procedure *Process* is called for each bin irrespective of its size. In the minimal case, the procedure *Process* is called only for bins of size greater than one. The bins of size one are processed according to the second **for** loop in the procedure *Search*.

The following physical analogy is helpful in making the algorithm more intuitive. Think of all the words of S as marbles. Imagine two arrays A and B. Toss the marbles randomly into the array A. Sort the bins of A by the number of marbles each bin contains. Then, for each bin of A, in the sorted order, do the following. Toss all the marbles in the bin randomly into B. If all of the marbles land in empty cells, we are done. Otherwise, take back all the marbles just tossed, and toss them again into B. Keep doing this until all the marbles land in empty cells. Continue this process until all the bins of A have been emptied into B. Note that this algorithm never fails to produce a PHF, but it may require a very large time in the worst case. The following example illustrates the MOS algorithm.

Example 2.1.1 Let the set S consist of the following 9 words: "bear", "dear", "fear", "gear", "hear", "near", "pear", "rear", "tear". The values of h_1 and h_2 on these are listed in Table 2.1. h_1 is random function which maps to an integer between 0 and 6 with equal probability. h_2 is an indexed invertible random function which maps to an integer between 0 and 8 with equal probability. Section 1.2 defines random functions and indexed invertible random functions precisely. See Pearson [33] for an example of constructing h_1 and h_2 functions.

The mapping step consists of applying the function h_1 to all the 9 words. Column 2 of Table 2.1 lists the values of h_1 on all these words.

The ordering step consists of sorting the bins by size. Bin 3 receives three words, so it is placed on top of the sorted list. Bins 0 and and 2 have two words each, so they are placed next in the list. Bins 6 and 1 receive one word each, so they are placed next in the list. As bins 4 and 5 are empty they go last in the list. The complete sorted order of bins is

```
Search()

{
    if (ñ = n) {

        /* In the minimal case, process all the bins of size greater than 1 */

        for (i = 1; |Q_ξ(i)| ≥ 2; i = i + 1)
                Process(Q_ξ(i));

        /* Process all the bins of size 1 */

        j = 1;
        for (j = 1; i ≤ N; i = i + 1) {

            /* Find an empty cell in B*/

            while (B[j] ≠Empty )
                j = j + 1;

            /* Assign it to the word in Q_ξ(i) */

            x = Word(Q_ξ(i));
            B[j] = x;
            A[ξ(i)] = h_2^{-1}(x, j);
        }

    else

        /* In the non-minimal case, process each bin irrespective of its size */

        for (i = 1; i ≤ N; i = i + 1)
                Process(Q_ξ(i));
    }
}
```

Figure 2.1: The *Search* program for MOS.

```
Process(Q_k)

{
        /* Vary i until all the words in Q_k land in empty cells */

        i = 1;
        while (CheckBin(Q_k, i) == FALSE)
                i = i + 1;

        /* Insert the words in Q_k in array B */

        for (all x ∈ Q_k)
                B[h_2(x, i)] = x;

        /* Record the value of i in A */

        A[k] = i;
}
```

Figure 2.2: The procedure *Process*.

```
CheckBin(Q_k, i)

{
        /* For each word x in Q_k check if it lands in an empty cell */

        S = {};
        for (all x ∈ Q_k) {
                if (B[h_2(x, i)] != Empty or h_2(x, i) ∈ S)
                        return (FALSE);
                S = S ∪ h_2(x, i);
        }
        return(TRUE);
}
```

Figure 2.3: The procedure *CheckBin*.

x	$h_1(x)$	$h_2(x,0)$	$h_2(x,1)$	$h_2(x,2)$	$h_2(x,3)$
"bear"	6	7	0	3	6
"dear"	3	4	6	0	5
"fear"	0	5	7	4	1
"gear"	3	4	0	5	2
"hear"	1	2	3	0	4
"near"	0	3	1	8	7
"pear"	2	3	8	6	7
"rear"	3	1	2	6	7
"tear"	2	8	7	4	5

Table 2.1: The values of h_1 and h_2 on the words in S.

i	0	1	2	3	4	5	6
$A[i]$	0	h_2^{-1}("hear",1)	1	1	-	-	h_2^{-1}("bear",4)

Table 2.2: The table A.

i	0	1	2	3	4	5	6	7	8
$B[i]$	"gear"	"hear"	"rear"	"near"	"bear"	"fear"	"dear"	"tear"	"pear"

Table 2.3: The table B.

$$\{Q_3, Q_0, Q_2, Q_6, Q_1, Q_4, Q_5\}$$

The searching step consists of applying $h_2(x, i)$ to all words in each bin in the sorted order. The bin 3 has 3 words "dear", "gear", and "rear". We start with $i = 0$. Applying $h_2(x, 0)$ to all three words, we get 4, 4, and 1. Since both "dear and "gear" map to 4, we discard this choice of i and consider $i = 1$. Applying $h_2(x, 1)$ to all these words, we get 6, 0, and 2. So we store "dear" in $B[6]$, "gear" in $B[0]$, and "rear" in $B[2]$. The value $A[3]$ is set to 1 because the trial succeeded for $i = 1$. Bin 0 is next in the sorted order which has words "fear" and "near". Applying $h_2(x, 0)$ to these words, we get 5 and 3. Since both $B[5]$ and $B[3]$ are empty, we store "fear" in $B[5]$, and "near" in $B[3]$, and set $A[0]$ to 0. Bin 2 is next in the sorted order. Applying $h_2(x, 0)$ to these words, we get 3 and 8. But $B[3]$ is already occupied by "near". So, we try $i = 1$, and get 8 and 7. Since both $B[8]$ and $B[7]$ are empty, we store "pear" in $B[8]$ and "tear" in $B[7]$, and set $A[2]$ to 1. The rest of the bins are of size 1. We store "hear" and "bear" in $B[1]$ and $B[4]$, and set $A[1]$ and $A[6]$ to h_2^{-1}("hear",1) and h_2^{-1}("bear",4) respectively. □

What is the need for handling the bins of size one separately? If we depend on random tossing to find empty cells for them, as we prove in Lemma 2.1.1, searching requires $\Theta(n \ln n)$ time on the average. So, bins of size 1 are processed separately to keep the $\Theta(n)$ time bound.

Lemma 2.1.1 *The expected cost of random tossing for bins of size one is* $\Theta(n(\ln n - \alpha))$.

Proof: Suppose there are k bins of size one in A. The expected value of k is $ne^{-\alpha}/\alpha$ [29]. The expected time taken for finding empty cells for these bins is

$$\begin{aligned} \frac{n}{1} + \frac{n}{2} + \cdots + \frac{n}{k} &= n \ln k + O(n) \\ &= n(\ln n - \alpha - \ln \alpha) + O(n) \\ &= \Theta(n(\ln n - \alpha)). \end{aligned}$$

□

The description of the MOS model is now complete. It remains to analyze its performance in the next section.

2.2 Analysis of the Running Time

An MOS algorithm employs two arrays A and B. A is an array of N integers and B is an array of $\tilde{n}$ words. Compared to A and B the sizes of other data structures are small. So we assume that the dominant operation is accessing these tables We measure the time by the number of accesses to A and B. MOS is a sequential algorithm consisting of three steps, namely mapping, ordering, and searching. The times taken by these steps are denoted as T_m, T_o, and T_s respectively. The total time T is the sum of these individual steps. The analysis of the mapping and ordering steps is very simple. The analysis of the searching step is the crucial one, and constitutes a major contribution of this thesis. We analyze the mapping and ordering steps in Section 2.2.1, and the searching step in Sections 2.2.2 and 2.2.3.

2.2.1 Analysis of the mapping and ordering steps

Theorem 2.2.1 *The mapping step requires* $\Theta(n)$ *time.*

Proof: In the mapping step, $h_1(x)$ is computed for all $x \in S$, and the words are stored in the appropriate bins. A linked list is maintained for each bin. A word is always inserted in the front of the linked list. Each operation takes $\Theta(1)$ time and the entire mapping phase takes $\Theta(n)$ time. □

Theorem 2.2.2 *The ordering step requires* $\Theta(n)$ *time.*

Proof: The ordering step consists of sorting the bins. Since the size of a bin is bounded by n in the worst case, we can sort all the bins in descending order in $\Theta(n)$ time using bin sort. □

2.2.2 Analysis of the searching step

The searching step involves processing the bins of A sequentially in the order obtained in the ordering step. Processing a bin Q_i consists of a number of *trials*. Each trial consists of a random toss by h_2 of the words in the bin Q_i into the partially-filled array B. A trial is *successful* if all the words in the bin fall into empty cells in B, otherwise it is a *failure*. Note that by our definition, a trial can result in a failure either when a word falls in a cell that was filled before the trial or even when two words of a bin both fall in the same cell that was empty before the trial. If a trial is not successful, we take back all the words into the bin and repeat until there is a successful trial.

The *cost* of a trial for bin Q_i is defined to be size $|Q_i|$ of the bin. This definition is justified by the fact that the number of times we access A and B during a trial is proportional to the number of words in the bin. The cost of the searching step is defined as the sum of the costs of the individual trials. For minimal perfect hashing, we have $\beta = 0$, and the cost of searching depends on n and α. We denote the cost of searching for minimal perfect hashing by MPHFCOST(n, α) and that of non-minimal perfect hashing by PHFCOST(n, α, β). Since we are tossing the words randomly into B, MPHFCOST is a function of n and α, and PHFCOST is a function of n, α, and β. Clearly, the expected value of MPHFCOST or PHFCOST is proportional to the average running time of the searching step, T_s.

We denote the total number of trials in the minimal case by TRIALS(n, α). In Section 2.2.3, we prove that TRIALS(n, α) has a lower bound of $\Omega(ne^{\alpha})$.

Definition 2.2.1 *For $j \geq 0$, N_j is a random variable that equals the number of bins of size j:*

$$N_j = |\{i : |Q_i| = j\}|.$$

If the N_j were independent random variables, the analysis of the searching step would have been far simpler. The fact that the N_j are dependent random variables greatly complicates the analysis.

Definition 2.2.2 *μ_j is a random variable equal to the total number of words in bins of size j or less, that is,*

$$\mu_j = \sum_{i=0}^{j} iN_i.$$

We need to analyze the random vector $\langle N_0, N_1, \ldots, N_n \rangle$ We start by considering a feasible distribution $\langle i_0, i_1, \ldots, i_n \rangle$, that is, a fixed vector that with some nonzero probability, equals $\langle N_0, N_1, \ldots, N_n \rangle$. For $\langle i_0, i_1, \ldots, i_n \rangle$ to be feasible, we must have the total number of bins equal to N:

$$\sum_{r=0}^{n} i_r = N,$$

and the total number of the words in the bins equal to n:

$$\sum_{r=0}^{n} r i_r = n.$$

Now, the probability $p_{i_0,i_1,\ldots,i_n}$ that $\langle N_0, N_1, \ldots, N_n \rangle = \langle i_0, i_1, \ldots, i_n \rangle$ is given by

$$\begin{aligned} p_{i_0,i_1,\ldots,i_n} &= \frac{n!}{0!^{i_0} 1!^{i_1} \cdots n!^{i_n}} \cdot \frac{N!}{N^n} \\ &= \frac{N!}{0!^{i_0} 1!^{i_1} \cdots n!^{i_n}} \cdot \frac{n!}{N^n}. \end{aligned} \tag{2.2}$$

Consider the generating function Φ_1 with an infinite sequence of parameters:

$$\Phi_1(z, s_0, s_1, \ldots) = \sum_{\substack{n \\ i_0+i_1+\cdots+i_n=N \\ 0i_0+1i_1+\cdots+ni_n=n}} p_{i_0,i_1,\ldots,i_n} s_0^{i_0} s_1^{i_1} \cdots s_n^{i_n} \frac{N^n z^n}{n!}.$$

Substituting the value of $p_{i_0,i_1,\ldots,i_n}$ from Equation 2.2, we get

$$\begin{aligned} \Phi_1(z, s_0, s_1, \ldots) &= \sum_{\substack{n \\ i_0+i_1+\cdots+i_n=N \\ 0i_0+1i_1+\cdots+ni_n=n}} \frac{N!}{0!^{i_0} 1!^{i_1} \cdots n!^{i_n}} s_0^{i_0} s_1^{i_1} \cdots s_n^{i_n} z^n \\ &= \left(\sum_{i=0}^{\infty} \frac{s_i z^i}{i!} \right)^N \end{aligned} \tag{2.3}$$

The probability $p_{i_0,i_1,\ldots,i_n}$ can be calculated by taking the coefficient of ${s_0}^{i_0} {s_1}^{i_1} \cdots {s_n}^{i_n} z^n$ and multiplying it by $n!/N^n$.

The generating function Φ_1 encapsulates the probability distribution for the number of bins of each size. From Φ_1, we can get a generating function $\Phi_2(z, s_0, s_1, \ldots)$ for the number of words in the bins of various sizes by substituting s_i^i for s_i in Equation 2.3.

$$\begin{aligned}\Phi_2(z, s_0, s_1, \ldots) &= \Phi_1(z, s_0^0, s_1^1, \ldots) \\ &= \sum_{\substack{i_0+i_1+\cdots+i_n=N \\ 0i_0+1i_1+\cdots+ni_n=n}} p_{i_0,i_1,\ldots,i_n} s_0^{0i_0} s_1^{1i_1} \cdots s_n^{ni_n} \frac{N^n z^n}{n!} \qquad (2.4) \\ &= \left(\sum_{i=0}^{\infty} \frac{s_i^i z^i}{i!}\right)^N \qquad (2.5)\end{aligned}$$

The probability $p_{i_0,i_1,\ldots,i_n}$ can be calculated by taking the coefficient of ${s_0}^{0i_0} {s_1}^{1i_1} \cdots {s_n}^{ni_n} z^n$ and multiplying it by $n!/N^n$.

A compound generating function $\Phi_3(z, s, j, s_j)$ for μ_{j-1} and N_j can be realized by substituting s for $s_0, s_1, \ldots, s_{j-1}$ and 1 for $s_{j+1}, s_{j+2}, \ldots, s_\infty$ in Equation 2.5.

$$\Phi_3(z, s, j, s_j) = \left(\sum_{i=0}^{j-1} \frac{s^i z^i}{i!} + \frac{s_j^j z^j}{j!} + \sum_{i=j+1}^{\infty} \frac{z^i}{i!}\right)^N \qquad (2.6)$$

The probability that $\mu_{j-1} = \tau$ and $N_j = k$ can be calculated by taking the coefficient of $s^\tau s_j^{kj} z^n$ and multiplying it by $n!/N^n$.

A generating function $\Phi_4(z, s)$ for μ_j can be realized by substituting s for $s_0, s_1, \ldots, s_j$ and 1 for $s_{j+1}, s_{j+2}, \ldots, s_\infty$ in Equation 2.5.

$$\Phi_4(z, s, j) = \left(\sum_{i=0}^{j} \frac{s^i z^i}{i!} + \sum_{i=j+1}^{\infty} \frac{z^i}{i!}\right)^N \qquad (2.7)$$

The probability that $\mu_j = \tau$ can be calculated by taking the coefficient of $s^\tau z^n$ and multiplying it by $n!/N^n$.

Let $\delta \geq 0$ be a constant. A compound generating function $\Phi_5(z, s, j, s_j)$ for $\mu_{j-1} + \delta$ and N_j is given by

$$\Phi_5(z, s, j, s_j) = s^\delta \left(\sum_{i=0}^{j-1} \frac{s^i z^i}{i!} + \frac{s_j^j z^j}{j!} + \sum_{i=j+1}^{\infty} \frac{z^i}{i!}\right)^N \qquad (2.8)$$

The probability that $\mu_{j-1} + \delta = \tau$ and $N_j = k$ can be calculated by taking the coefficient of $s^\tau s_j^{kj} z^n$ and multiplying it by $n!/N^n$.

Again, let $\delta \geq 0$ be a constant. A generating function $\Phi_6(z, s, j)$ for $\mu_j + \delta$ is given by

$$\Phi_6(z, s, j) = s^\delta \left(\sum_{i=0}^{j} \frac{s^i z^i}{i!} + \sum_{i=j+1}^{\infty} \frac{z^i}{i!} \right)^N \tag{2.9}$$

The probability that $\mu_j + \delta = \tau$ can be calculated by taking the coefficient of $s^\tau z^n$ and multiplying it by $n!/N^n$.

We use the function ϕ_k, where $1 \leq k \leq 6$, to denote the coefficient of $N^n z^n/n!$ in the generating function Φ_k. For example, $\phi_5(s, j, s_j)$ denotes the coefficient of $N^n z^n/n!$ in $\Phi_5(z, s, j, s_j)$. The coefficient of z^n in Φ_k is given by $N^n \phi_k/n!$. ϕ_k can be calculated with the help of Maclaurin's theorem [7].

Theorem 2.2.3 *(Maclaurin's Theorem) Let Γ_0 be a circle in the complex plane, with its center at the origin. If Φ is analytic everywhere in the circle Γ_0, then at each point z in the circle, $\Phi(z)$ is represented by the series*

$$\Phi(z) = \sum_{n=0}^{\infty} a_n z^n.$$

The coefficients a_n can be evaluated by the contour integral around the origin inside the circular region Γ_0:

$$a_n = \frac{1}{2\pi\sqrt{-1}} \oint \frac{\Phi(z)}{z^{n+1}} dz.$$

Applying Maclaurin's theorem to Φ_k, $1 \leq k \leq 6$,

$$\frac{N^n}{n!}\phi_k = \frac{1}{2\pi\sqrt{-1}} \oint \frac{\Phi_k(z)}{z^{n+1}} dz$$

It is difficult to evaluate this integral exactly. However, we can evaluate its asymptotic value as $n \to \infty$ with the help of Hayman's theorem [40]. Hayman's theorem is applicable to only *Hayman admissible functions.* The original set of conditions to test whether a function is Hayman admissible or not are complicated. However, Hayman gives a set of recursive conditions to test the Hayman admissibility of a given function in terms of already known Hayman admissible functions. We describe these conditions below.

1. e^z is Hayman admissible.

2. If $\chi_1(z)$ and $\chi_2(z)$ are Hayman admissible, then $e^{\chi_1(z)}$ and $\chi_1(z)\chi_2(z)$ are Hayman admissible.

3. If $\chi(z)$ is Hayman admissible and $P(z)$ is a polynomial, then $\chi(z) + P(z)$ is Hayman admissible. If the leading coefficient of $P(z)$ is positive, then $\chi(z)P(z)$ and $P(\chi(z))$ are Hayman admissible.

In Lemma 2.2.1, we prove that $\Phi_4(z, s, j)$ is Hayman admissible.

Theorem 2.2.4 *(Hayman's Theorem) Let $z = re^{i\theta}$ be a complex variable and let $\Phi(z) = \sum_{n=0}^{\infty} a_n z^n$ be a Hayman admissible function. Let $a(r)$ and $b(r)$ be defined as*

$$\begin{aligned} a(r) &= r\frac{\Phi'(r)}{\Phi(r)}, \\ b(r) &= ra'(r) \end{aligned}$$

where $\Phi'(r)$ and $a'(r)$ are derivatives of $\Phi(r)$ and $a(r)$ with respect to r respectively. Then

$$a_n = \frac{\Phi(r)}{r^n\sqrt{2\pi b(r)}}\left[\exp\left\{-\frac{[a(r)-n]^2}{2b(r)}\right\} + o(1)\right].$$

The following definitions will be used in further analysis.

Definition 2.2.3

$$\begin{aligned} t_j &= e^{-\alpha}\sum_{i=0}^{j}\frac{\alpha^i}{i!} \\ u_j &= \frac{\beta+1}{\beta+t_j} \end{aligned}$$

Definition 2.2.4

$$g(z,s,j) = \sum_{i=0}^{j}\frac{s^i z^i}{i!} + \sum_{i=j+1}^{\infty}\frac{z^i}{i!}.$$

Observe that $\Phi_4(z, s, j) = g(z, s, j)^N$.

Definition 2.2.5

$$\rho_1(\alpha,s,j) = \sqrt{\frac{\alpha}{\left(\alpha\frac{g'(\alpha,s,j)}{g(\alpha,s,j)} - \alpha^2\frac{g'(\alpha,s,j)^2}{g(\alpha,s,j)^2} + \alpha^2\frac{g''(\alpha,s,j)}{g(\alpha,s,j)}\right)}}.$$

where $g'(z,s,j)$ is the derivative of $g(z,s,j)$ with respect to z.

Definition 2.2.6

$$\rho_2(\alpha,s,j) = \alpha^2\frac{\left(\frac{g'(\alpha,s,j)}{g(\alpha,s,j)} - 1\right)^2}{2\left(\alpha\frac{g'(\alpha,s,j)}{g(\alpha,s,j)} - \alpha^2\frac{g'(\alpha,s,j)^2}{g(\alpha,s,j)^2} + \alpha^2\frac{g''(\alpha,s,j)}{g(\alpha,s,j)}\right)}.$$

Lemma 2.2.1 *$\Phi_4(z,s,j)$ is Hayman admissible.*

Proof: According to Definition 2.2.4,

$$\begin{aligned} g(z,s,j) &= \sum_{i=0}^{j}\frac{s^i z^i}{i!} + \sum_{i=j+1}^{\infty}\frac{z^i}{i!} \\ &= e^z + \sum_{i=0}^{j}\frac{(s^i-1)z^i}{i!}. \end{aligned}$$

According to condition 3 of Hayman admissibility, $g(z,s,j)$ is Hayman admissible since it is the sum of a Hayman admissible function, e^z, and a polynomial in z. According to condition 2, $g(z,s,j)^N$ is a Hayman admissible function since it is a product of Hayman admissible functions. Hence $\Phi_4(z,s,j)$ is a Hayman admissible function. □

We make use of the fact that $\Phi_4(Z,s,j)$ is Hayman admissible and apply Hayman's theorem in Lemma 2.2.2.

Lemma 2.2.2 *If α is $O(1)$, as $n, N \to \infty$, a generating function for the asymptotic value of $\phi_4(s,j)$ is given by*

$$\phi_4(s,j) \sim \rho_1(\alpha,s,j)\left(e^{-\alpha-\rho_2(\alpha,s,j)}\left(\sum_{i=0}^{j}\frac{s^i\alpha^i}{i!} + \sum_{i=j+1}^{\infty}\frac{\alpha^i}{i!}\right)\right)^N$$

Proof: According to Maclaurin's theorem,

$$\begin{aligned}\frac{N^n}{n!}\phi_4(s,j) &= \frac{1}{2\pi\sqrt{-1}}\oint\frac{\Phi_4(z,s,j)}{z^{n+1}}dz \\ &= \frac{1}{2\pi\sqrt{-1}}\oint\frac{1}{z^{n+1}}\left(\sum_{i=0}^{j}\frac{s^i z^i}{i!}+\sum_{i=j+1}^{\infty}\frac{z^i}{i!}\right)^N dz\end{aligned}$$

The asymptotic value of the integral on the right can be evaluated with the help of Hayman's formula. Substituting $g(z,s,j)^N$ for $\Phi_4(z,s,j)$, and applying Hayman's formula at $z=\alpha$, we get

$$\begin{aligned}a &= \alpha\left.\frac{(g^N)'}{g^N}\right|_{z=\alpha} \\ &= \alpha\frac{Ng'(\alpha,s,j)g(\alpha,s,j)^{N-1}}{g(\alpha,s,j)^N} \\ &= N\alpha\frac{g'(\alpha,s,j)}{g(\alpha,s,j)},\end{aligned} \tag{2.10}$$

and

$$b = N\left(\alpha\frac{g'(\alpha,s,j)}{g(\alpha,s,j)}-\alpha^2\frac{g'(\alpha,s,j)^2}{g(\alpha,s,j)^2}+\alpha^2\frac{g''(\alpha,s,j)}{g(\alpha,s,j)}\right). \tag{2.11}$$

Hence,

$$\frac{N^n}{n!}\phi_4(s,j) = \frac{g(\alpha,s,j)^N}{\alpha^n\sqrt{2\pi b}}\exp\left(-\frac{(a-n)^2}{2b}+o(1)\right).$$

Substituting $N\alpha$ for n, we get

$$\phi_4(s,j) = \frac{n!}{N^n}\frac{g(\alpha,s,j)^N}{\alpha^n\sqrt{2\pi b}}\exp\left(-\frac{(a-n)^2}{2b}+o(1)\right)$$

According to Stirling's approximation [28], we can substitute

$$n! \sim \frac{n^n}{e^n}\sqrt{2\pi n}$$

Hence,

$$\phi_4(s,j) \sim \frac{n^n\sqrt{2\pi n}}{e^n N^n}\frac{g(\alpha,s,j)^N}{\alpha^n\sqrt{2\pi b}}\exp\left(-\frac{(a-n)^2}{2b}+o(1)\right)$$

$$= \sqrt{\frac{n}{b}}\frac{g(\alpha,s,j)^N}{e^n}\exp\left(-\frac{(a-n)^2}{2b}+o(1)\right)$$
$$= \sqrt{\frac{n}{b}}\left(e^{-\alpha}g(\alpha,s,j)\right)^N\exp\left(-\frac{(a-n)^2}{2b}+o(1)\right). \quad (2.12)$$

Substituting the values of a and b from Equations 2.10 and 2.11, we have

$$\frac{(a-n)^2}{2b} = \frac{\left(N\alpha\frac{g'(\alpha,s,j)}{g(\alpha,s,j)}-N\alpha\right)^2}{2N\left(\alpha\frac{g'(\alpha,s,j)}{g(\alpha,s,j)}-\alpha^2\frac{g'(\alpha,s,j)^2}{g(\alpha,s,j)^2}+\alpha^2\frac{g''(\alpha,s,j)}{g(\alpha,s,j)}\right)}$$
$$= N\alpha^2\frac{\left(\frac{g'(\alpha,s,j)}{g(\alpha,s,j)}-1\right)^2}{2\left(\alpha\frac{g'(\alpha,s,j)}{g(\alpha,s,j)}-\alpha^2\frac{g'(\alpha,s,j)^2}{g(\alpha,s,j)^2}+\alpha^2\frac{g''(\alpha,s,j)}{g(\alpha,s,j)}\right)}$$
$$= N\rho_2(\alpha,s,j). \quad (2.13)$$

Similarly, substituting the value of b from Equation 2.10, we have

$$\frac{n}{b} = \frac{N\alpha}{N\left(\alpha\frac{g'(\alpha,s,j)}{g(\alpha,s,j)}-\alpha^2\frac{g'(\alpha,s,j)^2}{g(\alpha,s,j)^2}+\alpha^2\frac{g''(\alpha,s,j)}{g(\alpha,s,j)}\right)}$$
$$= \frac{\alpha}{\left(\alpha\frac{g'(\alpha,s,j)}{g(\alpha,s,j)}-\alpha^2\frac{g'(\alpha,s,j)^2}{g(\alpha,s,j)^2}+\alpha^2\frac{g''(\alpha,s,j)}{g(\alpha,s,j)}\right)}$$
$$= \rho_1(\alpha,s,j)^2. \quad (2.14)$$

Substituting the values of $(a-n)^2/(2b)$ and n/b from Equation 2.13 and 2.14 in Equation 2.12, we get

$$\phi_4(s,j) = \rho_1(\alpha,s,j)\left(e^{-\alpha-\rho_2(\alpha,s,j)}\left(\sum_{i=0}^{j}\frac{s^i\alpha^i}{i!}+\sum_{i=j+1}^{\infty}\frac{\alpha^i}{i!}\right)\right)^N\exp(o(1))$$
$$\sim \rho_1(\alpha,s,j)\left(e^{-\alpha-\rho_2(\alpha,s,j)}\left(\sum_{i=0}^{j}\frac{s^i\alpha^i}{i!}+\sum_{i=j+1}^{\infty}\frac{\alpha^i}{i!}\right)\right)^N.$$

□

Lemma 2.2.3 *If α is $O(1)$, the following equations hold:*

$$\rho_1(\alpha,1,j) = 1$$
$$\rho_2(\alpha,1,j) = 0$$

Let c be a constant. Then, as $s \to 1$, $N \to \infty$, and $N(1-s) \to c$, we have

$$N\rho_2(\alpha, s, j) \;\to\; 0.$$

Proof: $g(z,1,j) = e^z$ and $g(\alpha,1,j) = g'(\alpha,1,j) = g''(\alpha,1,j) = e^\alpha$ independent of j. Substituting these in Definitions 2.2.5 and 2.2.6, we obtain $\rho_1(\alpha,1) = 1$ and $\rho_2(\alpha,1) = 0$.

Substituting the value ρ_2 from Definition 2.2.6 in $N\rho_2(\alpha,s,j)$,

$$N\rho_2(\alpha,s,j) \;=\; N\alpha^2 \frac{\left(\frac{g'(\alpha,s,j)}{g(\alpha,s,j)} - 1\right)^2}{2\left(\alpha\frac{g'(\alpha,s,j)}{g(\alpha,s,j)} - \alpha^2\frac{g'(\alpha,s,j)^2}{g(\alpha,s,j)^2} + \alpha^2\frac{g''(\alpha,s,j)}{g(\alpha,s,j)}\right)}.$$

As $s \to 1$, from Definition 2.2.4, it can be easily verified that $g(\alpha,s,j)$, $g'(\alpha,s,j)$ and $g''(\alpha,s,j)$ tend to e^α. Substituting e^α for g, g', and g'' in the denominator, we see that it tends to 2α. If we prove that $N(g'(\alpha,s,j) - g(\alpha,s,j))^2 \to 0$ as $s \to 1$, it follows that $N\rho_2(\alpha,s,j) \to 0$.

$$\begin{aligned}
g'(\alpha,s,j) - g(\alpha,s,j) &= \sum_{i=0}^{j-1}\frac{s^{i+1}\alpha^i}{i!} + \sum_{i=j}^{\infty}\frac{\alpha^i}{i!} - \sum_{i=0}^{j}\frac{s^i\alpha^i}{i!} + \sum_{i=j+1}^{\infty}\frac{\alpha^i}{i!} \\
&= (s-1)\sum_{i=0}^{j-1}\frac{s^i\alpha^i}{i!} + \frac{\alpha^j}{j!} - \frac{s^j\alpha^j}{j!} \\
&= (1-s)\left\{\frac{\alpha^j}{j!}\sum_{i=0}^{j-1}s^i - \sum_{i=0}^{j-1}\frac{s^i\alpha^i}{i!}\right\} \\
&< (1-s)e^\alpha
\end{aligned}$$

So,

$$N(g'(\alpha,s,j) - g(\alpha,s,j))^2 \;<\; N(1-s)^2e^{2\alpha}$$

Substituting $s = 1 - c/N$, we get

$$\begin{aligned}
N(g'(\alpha,s,j) - g(\alpha,s,j))^2 &< \frac{c^2}{N}e^{2\alpha} \\
&\to 0
\end{aligned}$$

as $N \to \infty$. Hence $N\rho_2(\alpha,s,j) \to 0$ and the lemma follows.

□

Definition 2.2.7 *$\zeta : I \times I \rightarrow I$ is a function such that $\zeta(j,r)$ gives the index of the rth bin of size j in the sorted array of bins, that is, $\zeta(j,r) = \xi(i+r)$ if*

$$\begin{aligned} |Q_{\xi(i-1)}| &> j \\ |Q_{\xi(i)}| &= j \\ |Q_{\xi(i+1)}| &= j \\ &\vdots \\ |Q_{\xi(i+r)}| &= j \end{aligned}$$

Definition 2.2.8 *Υ_{jr} is a random variable that equals the number of empty cells in B before tossing the words of $Q_{\zeta(j,r)}$ into B.*

For convenience, we let $\Upsilon_j = \Upsilon_{j1}$, and $\Upsilon_0 = \beta n$. For $j \geq 1$, Υ_j can be expressed in terms of the N_j as

$$\begin{aligned} \Upsilon_j &= \Upsilon_0 + \sum_{i=1}^{j} iN_i \\ &= \Upsilon_0 + \mu_j. \end{aligned}$$

Similarly,

$$\Upsilon_{jr} = \Upsilon_j - (r-1)j.$$

Definition 2.2.9 *Y_{jr} is a random variable that equals the number of trials performed until a successful trial for the bin $Q_{\zeta(j,r)}$, including the trial that is successful.*

The cost of a single trial for a bin $Q_{\zeta(j,r)}$ is j, and the total cost for the bin $Q_{\zeta(j,r)}$ is jY_{jr}.

Definition 2.2.10 *The random variable X_j is the number of trials for all bins of size j. That is,*

$$X_j = \sum_{r=1}^{N_j} Y_{jr}.$$

MPHFCOST and PHFCOST can be computed by the formulas:

$$\text{MPHFCOST} = \sum_{j=1}^{n} jX_j \tag{2.15}$$

$$\text{PHFCOST} = \sum_{j=1}^{n} jX_j. \tag{2.16}$$

The total number of trials, TRIALS, can be computed by the formula:

$$\text{TRIALS} = \sum_{j=2}^{n} X_j \tag{2.17}$$

Lemma 2.2.4 *For $j > 1$, if there are i empty cells in B, the expected number of trials for $Q_{\zeta(j,r)}$ is*

$$E[Y_{jr}|\Upsilon_{jr} = i] = \frac{(1+\beta)^j n^j}{i^{[j]}}$$

Proof: If $\Upsilon_{jr} = i$, the probability p_{jri} that a trial of tossing the words in $Q_{\zeta(j,r)}$ into B succeeds is

$$p_{jri} = \frac{i}{(1+\beta)n} \cdot \frac{i-1}{(1+\beta)n} \cdots \frac{i-j+1}{(1+\beta)n}.$$

The probability that $Y_{jr} = k$ is given by the probability that the trial is a failure for the first $k-1$ times and is successful the kth time.

$$\Pr[Y_{jr} = k|\Upsilon_{jr} = i] = (1 - p_{jri})^{k-1} p_{jri}$$

For this distribution of Y_{jr}, we can easily prove that [14] $E[Y_{jr}] = 1/p_{jri}$.

$$\begin{aligned} E[Y_{jr}|\Upsilon_{jr} = i] &= \frac{1}{p_{jri}} \\ &= \frac{(1+\beta)n}{i} \cdot \frac{(1+\beta)n}{i-1} \cdots \frac{(1+\beta)n}{i-(j-1)} \\ &= \frac{(1+\beta)^j n^j}{i^{[j]}}. \end{aligned}$$

□

Lemma 2.2.5

$$E[X_j|\Upsilon_{j-1} = i, N_j = k] = \sum_{r=0}^{k-1} \frac{(1+\beta)^j n^j}{(i+(r+1)j)^{[j]}}.$$

Proof: From Definition 2.2.10,

$$X_j = \sum_{r=1}^{N_j} Y_{jr}.$$

Hence,

$$E[X_j|\Upsilon_{j-1} = i, N_j = k] = \sum_{r=1}^{k} E[Y_{jr}|\Upsilon_{j-1} = i, N_j = k]$$

From Definition 2.2.8, $\Upsilon_{jr} = \Upsilon_j - (r-1)j$. Also, the conditions $\Upsilon_{j-1} = i$ and $N_j = k$ imply that $\Upsilon_{jr} = i + (k-r+1)j$. Substituting this value on the right hand side, we have

$$E[X_j|\Upsilon_{j-1} = i, N_j = k] = \sum_{r=1}^{k} E[Y_{jr}|\Upsilon_{jr} = i + (k-r+1)j].$$

Applying Lemma 2.2.4, we obtain

$$\begin{aligned} E[X_j|\Upsilon_{j-1} = i, N_j = k] &= \sum_{r=1}^{k} \frac{(1+\beta)^j n^j}{(i+(k-r+1)j)^{[j]}} \\ &= \sum_{r=1}^{k} \frac{(1+\beta)^j n^j}{(i+rj)^{[j]}}. \end{aligned}$$

□

Lemma 2.2.6 *A compound generating function for Υ_j and N_j is given by*

$$\Phi_5(z,s,j,s_j) = s^{\beta n}\left(\sum_{i=0}^{j-1}\frac{s^i z^i}{i!} + \frac{s_j^j z^j}{j!} + \sum_{i=j+1}^{\infty}\frac{z^i}{i!}\right)^N$$

Proof: Since $\Upsilon_j = \Upsilon_0 + \mu_j$, and Υ_0 is a constant βn, we can get a compound generating function for Υ_j and N_j by substituting $\delta = \beta n$ in Equation 2.8. □

Lemma 2.2.7 *A generating function for Υ_j is given by*

$$\Phi_6(z,s,j) = s^{\beta n}\left(\sum_{i=0}^{j}\frac{s^i z^i}{i!} + \sum_{i=j+1}^{\infty}\frac{z^i}{i!}\right)^N$$

Proof: Since $\Upsilon_j = \Upsilon_0 + \mu_j$, and Υ_0 is a constant βn we can get a generating function for Υ_j by substituting $\delta = \beta n$ in Equation 2.9. □

Lemma 2.2.8

$$E[X_j] = (1+\beta)^j n^j \int_0^1 \frac{(1-s)^{j-1}}{(j-1)!} \frac{s^{\beta n}}{1-s^j} \{\phi_4(s, j-1) - \phi_4(s, j)\} ds \tag{2.18}$$

Proof: Define $X'_j = X_j(1+\beta)^{-j} n^{-j}$. We calculate $E[X'_j]$ first and then multiply it by $(1+\beta)^j n^j$. We can calculate $E[X'_j]$ by calculating a generating function $\Phi(z, s, j)$ for it first and then applying Maclaurin's theorem to $\Phi(z, s, j)$. We denote the expectation of X'_j by $E_{\hat{n}}[X'_j]$ when the number of words is $\hat{n}$. $E[X'_j]$ is identical to $E_n[X'_j]$.

$$\begin{aligned} \Phi(z, s, j) &= \sum_{\hat{n}=0}^{\infty} \frac{N^{\hat{n}} z^{\hat{n}}}{\hat{n}!} E_{\hat{n}}[X'_j] \\ &= \sum_{\hat{n}=0}^{\infty} \frac{N^{\hat{n}} z^{\hat{n}}}{\hat{n}!} \sum_{k,i} E[X'_j | \Upsilon_{j-1} = i, N_j = k] \Pr[\Upsilon_{j-1} = i, N_j = k]. \end{aligned} \tag{2.19}$$

From Lemma 2.2.5,

$$E[X_j | \Upsilon_{j-1} = i, N_j = k] = \sum_{r=0}^{k-1} \frac{(1+\beta)^j n^j}{(i+(r+1)j)^{[j]}}.$$

So,

$$\begin{aligned} E[X'_j | \Upsilon_{j-1} = i, N_j = k] &= \sum_{r=0}^{k-1} \frac{1}{(i+(r+1)j)^{[j]}} \\ &= \sum_{r=1}^{k} \frac{1}{(i+rj)^{[j]}}. \end{aligned}$$

Substituting this value in Equation 2.19,

$$\Phi(z, s, j) = \sum_{\hat{n}=0}^{\infty} \frac{N^{\hat{n}} z^{\hat{n}}}{\hat{n}!} \sum_{k,i} \sum_{r=1}^{k} \frac{1}{(i+rj)^{[j]}} \Pr[\Upsilon_{j-1} = i, N_j = k]$$

Using integration by parts, it is easy to prove that

$$\int_0^1 \frac{(1-s)^{j-1}}{(j-1)!} s^l ds = \frac{1}{(l+j)^{[j]}}.$$

Substituting this integral in the above expression,

$$\begin{aligned}
\Phi(z,s,j) &= \sum_{\hat{n}=0}^{\infty} \frac{N^{\hat{n}} z^{\hat{n}}}{\hat{n}!} \sum_{k,i} \sum_{r=1}^{k} \frac{1}{(i+rj)^{[j]}} \Pr[\Upsilon_{j-1} = i, N_j = k] \\
&= \sum_{\hat{n}=0}^{\infty} \frac{N^{\hat{n}} z^{\hat{n}}}{\hat{n}!} \sum_{k,i} \sum_{r=1}^{k} \int_0^1 \frac{(1-s)^{j-1}}{(j-1)!} s^{i+(r-1)j} ds \Pr[\Upsilon_{j-1} = i, N_j = k] \\
&= \int_0^1 \frac{(1-s)^{j-1}}{(j-1)!} \sum_{\hat{n}=0}^{\infty} \frac{N^{\hat{n}} z^{\hat{n}}}{\hat{n}!} \sum_{k,i} \sum_{r=0}^{k-1} s^{i+rj} \Pr[\Upsilon_{j-1} = i, N_j = k] ds \\
&= \int_0^1 \frac{(1-s)^{j-1}}{(j-1)!} \sum_{\hat{n}=0}^{\infty} \frac{N^{\hat{n}} z^{\hat{n}}}{\hat{n}!} \sum_{k,i} \frac{s^i - s^{i+kj}}{1-s^j} \Pr[\Upsilon_{j-1} = i, N_j = k] ds
\end{aligned}$$

From Lemma 2.2.7 we see that the summations can be written in terms of Φ_6 as

$$\frac{\Phi_6(z,s,j-1) - \Phi_6(z,s,j)}{1-s^j}$$

Replacing the summations by the above expression

$$\Phi(z,s,j) = \int_0^1 \frac{(1-s)^{j-1}}{(j-1)!} \left\{ \frac{\Phi_6(z,s,j-1) - \Phi_6(z,s,j)}{1-s^j} \right\} ds$$

Since $\Phi_6(z,s,j-1) = s^{\beta n}\Phi_4(z,s,j-1)$ and $\Phi_6(z,s,j) = s^{\beta n}\Phi_4(z,s,j)$ we can substitute them in the above expression.

$$\begin{aligned}
\Phi(z,s,j) &= \int_0^1 \frac{(1-s)^{j-1}}{(j-1)!} \left\{ \frac{s^{\beta n}\Phi_4(z,s,j-1) - s^{\beta n}\Phi_4(z,s,j)}{1-s^j} \right\} ds \\
&= \int_0^1 \frac{(1-s)^{j-1}}{(j-1)!} \frac{s^{\beta n}}{1-s^j} \{\Phi_4(z,s,j-1) - \Phi_4(z,s,j)\} ds
\end{aligned}$$

Applying Maclaurin's theorem to $\Phi(z,s,j)$ we get,

$$\begin{aligned}
E[X'_j] &= \oint \frac{\Phi(z,s,j)}{z^{n+1}} dz \\
&= \int_0^1 \frac{(1-s)^{j-1}}{(j-1)!} \frac{s^{\beta n}}{1-s^j} \{\phi_4(s,j-1) - \phi_4(s,j)\} ds
\end{aligned}$$

Substituting $X_j = (1+\beta)^j n^j$,

$$E[X_j] = (1+\beta)^j n^j \int_0^1 \frac{(1-s)^{j-1}}{(j-1)!} \frac{s^{\beta n}}{1-s^j} \{\phi_4(s,j-1) - \phi_4(s,j)\} ds,$$

as required. □

Lemma 2.2.9 *For $j > 1$, the expected number of trials for Q_j is*

$$E[X_j] \sim \frac{(1+\beta)n}{j}\left\{\frac{u_{j-2}^{j-1} - u_{j-1}^{j-1}}{j-1},\right\}$$

and for $j = 1$

$$E[X_1] \sim (1+\beta)n \ln\left(\frac{\beta + e^{-\alpha}}{\beta}\right)$$

Proof:

Case 1: $j > 1$

Consider the value of $\phi_4(s,j)$ from Lemma 2.2.2.

$$\phi_4(s,j) \sim \rho_1(\alpha,s,j)\left(e^{-\alpha-\rho_2(\alpha,s,j)}\left(\sum_{i=0}^{j}\frac{s^i\alpha^i}{i!} + \sum_{i=j+1}^{\infty}\frac{\alpha^i}{i!}\right)\right)^N$$

Clearly, as $N \to \infty$, $\phi_4(s,j) \to 0$ everywhere except in the neighborhood of $s = 1$. So, as $N \to \infty$, the integral in Equation 2.18 tends to zero everywhere except in the neighborhood of $s = 1$. We can evaluate the integral by setting $s = 1 - c/N$. We vary c from 0 to $N^{1-\epsilon}$, so that $c/N \to 0$ as $N \to \infty$.

$$\begin{aligned} E[X_j] &= (1+\beta)^j n^j \int_0^1 \frac{(1-s)^{j-1}}{(j-1)!}\frac{s^{\beta n}}{1-s^j}\{\phi_4(s,j-1) - \phi_4(s,j)\}\,ds \\ &= (1+\beta)^j n^j \int_0^1 \frac{(\frac{c}{N})^{j-1}}{(j-1)!}\frac{(1-c/N)^{\beta n}}{1-(1-\frac{c}{N})^j}\{\phi_4(1-c/N,j-1) - \phi_4(1-c/N,j)\}\,ds \end{aligned}$$

$$\begin{aligned} ds &= -\frac{dc}{N} \\ \left(1-\frac{c}{N}\right)^{\beta n} &\approx e^{-\beta cn/N} \\ &= e^{-\beta c\alpha} \\ 1-\left(1-\frac{c}{N}\right)^j &= 1-\left(1-\frac{jc}{N}\right) \\ &= \frac{jc}{N} \end{aligned}$$

Substituting these values in the integral,

$$\begin{aligned} E[X_j] &= -(1+\beta)^j n^j \int_{N^{1-\epsilon}}^{0} \frac{(\frac{c}{N})^{j-1}}{(j-1)!} \frac{e^{-\beta c\alpha}}{\frac{jc}{N}} \{\phi_4(1-c/N, j-1) - \phi_4(1-c/N, j)\} \frac{dc}{N} \\ &= \frac{(1+\beta)^j n}{j} \int_0^{N^{1-\epsilon}} \frac{n^{j-1}c^{j-2}}{N^{j-1}(j-1)!} e^{-\beta c\alpha} \{\phi_4(1-c/N, j-1) - \phi_4(1-c/N, j)\} dc \\ &= \frac{(1+\beta)^j n}{j(j-1)} \int_0^{\infty} \frac{\alpha^{j-1}c^{j-2}}{(j-2)!} e^{-\beta c\alpha} \{\phi_4(1-c/N, j-1) - \phi_4(1-c/N, j)\} dc \quad (2.20) \end{aligned}$$

According to Lemma 2.2.3 as $N \to \infty, N(1-s) \to c, s \to 1$,

$$\begin{aligned} \rho_1(\alpha, s, j) &\to 1 \\ N\rho_2(\alpha, s, j) &\to 0 \end{aligned}$$

Substituting these values in Lemma 2.2.2, we obtain

$$\phi_4(s, j) \to \left(e^{-\alpha} \left(\sum_{i=0}^{j} \frac{s^i \alpha^i}{i!} + \sum_{i=j+1}^{\infty} \frac{\alpha^i}{i!} \right) \right)^N$$

Substituting $s = 1 - c/N$,

$$\begin{aligned} \phi_4(s, j) &\sim \left(e^{-\alpha} \left(\sum_{i=0}^{j} \frac{(1-\frac{c}{N})^i \alpha^i}{i!} + \sum_{i=j+1}^{\infty} \frac{\alpha^i}{i!} \right) \right)^N \\ &= \left(e^{-\alpha} \left(\sum_{i=0}^{j} \frac{\left(1-\frac{ic}{N}\right) \alpha^i}{i!} + \sum_{i=j+1}^{\infty} \frac{\alpha^i}{i!} \right) \right)^N \\ &= \left(e^{-\alpha} \left(\sum_{i=0}^{j} \frac{\alpha^i}{i!} + \sum_{i=j+1}^{\infty} \frac{\alpha^i}{i!} \right) - e^{-\alpha} \left(\sum_{i=0}^{j} \frac{\frac{c}{N}\alpha^i}{(i-1)!} \right) \right)^N \\ &= \left(1 - \frac{c\alpha}{N} t_{j-1}\right)^N \\ &\sim e^{-c\alpha t_{j-1}} \quad (2.21) \end{aligned}$$

Similarly $\phi_4(s, j-1) \sim e^{-c\alpha t_{j-2}}$. Substituting the values of $\phi_4(s, j)$ and $\phi_4(s, j-1)$ in the Equation 2.20,

$$E[X_j] = \frac{(1+\beta)^j n}{j(j-1)} \int_0^{\infty} \frac{\alpha^{j-1}c^{j-2}}{(j-2)!} \left\{ e^{-c\alpha(\beta + t_{j-2})} - e^{-c\alpha(\beta + t_{j-1})} \right\} dc$$

$$
\begin{aligned}
&= \frac{(1+\beta)^j n}{j(j-1)} \left\{ \int_0^\infty \frac{\alpha^{j-1}c^{j-2}}{(j-2)!} e^{-c\alpha(\beta+t_{j-2})} dc - \int_0^\infty \frac{\alpha^{j-1}c^{j-2}}{(j-2)!} e^{-c\alpha(\beta+t_{j-1})} dc \right\} \\
&= \frac{(1+\beta)^j n}{j(j-1)} \left\{ \frac{1}{(\beta+t_{j-2})^{j-1}} \int_0^\infty \frac{u^{j-2}}{(j-2)!} e^{-u} du - \right. \\
&\qquad \left. \frac{1}{(\beta+t_{j-1})^{j-1}} \int_0^\infty \frac{u^{j-2}}{(j-2)!} e^{-u} du \right\} \\
&= \frac{(1+\beta)^j n}{j(j-1)} \left\{ \frac{1}{(\beta+t_{j-2})^{j-1}} \frac{\Gamma(j-1)}{(j-2)!} - \frac{1}{(\beta+t_{j-1})^{j-1}} \frac{\Gamma(j-1)}{(j-2)!} \right\} \\
&= \frac{(1+\beta)n}{j(j-1)} \left\{ (1+\beta)^{j-1} \left\{ \frac{1}{(\beta+t_{j-2})^{j-1}} - \frac{1}{(\beta+t_{j-1})^{j-1}} \right\} \right\} \\
&= \frac{(1+\beta)n}{j(j-1)} \left\{ u_{j-2}^{j-1} - u_{j-1}^{j-1} \right\}.
\end{aligned}
$$

Case 2: $j = 1$

Substituting $j = 1$ in the integral in Equation 2.20,we get

$$
\begin{aligned}
E[X_1] &\sim -(1+\beta)^1 n^1 \int_{N^{1-\epsilon}}^0 \frac{(\frac{c}{N})^{1-1}}{(1-1)!} \frac{e^{-\beta c\alpha}}{\frac{1c}{N}} \{\phi_4(1-c/N, 1-1) - \phi_4(1-c/N, 1)\} \frac{dc}{N} \\
&\sim (1+\beta)n \left\{ \int_0^\infty e^{-\beta c\alpha} \frac{\phi_4(1-c/N, 0) - \phi_4(1-c/N, 1)}{c} dc \right\}
\end{aligned}
$$

From Equation 2.21, $\phi_4(1-c/N, 1) \sim \exp(-c\alpha t_0) = \exp(-c\alpha e^{-\alpha})$. Also $\phi_4(s, 0) = 1$. Substituting it in the above expression

$$
\begin{aligned}
E[X_1] &\sim (1+\beta)n \left\{ \int_0^\infty \frac{\exp(-c\alpha\beta) - \exp(-c\alpha(\beta + e^{-\alpha}))}{c} dc \right\} \\
&= (1+\beta)n \ln\left(\frac{\beta + e^{-\alpha}}{\beta}\right)
\end{aligned}
$$

□

Theorem 2.2.5 *The expected cost of the searching step* $E[PHFCOST(n, \alpha, \beta)]$ *tends to* $nZ_{non\text{-}minimal}(\alpha, \beta)$ *for non-minimal perfect hashing, as* $n \to \infty$ *where*

$$
Z_{\text{non-minimal}}(\alpha, \beta) = (1+\beta) \left[\ln\left(\frac{\beta + e^{-\alpha}}{\beta}\right) + \sum_{j=1}^{\infty} \left\{ \frac{u_{j-1}^j - u_j^j}{j} \right\} \right]
$$

Proof:

$$\begin{aligned} \text{PHFCOST}(n,\alpha,\beta) &= \sum_{j=1}^{\infty} jX_j \\ E[\text{PHFCOST}(n,\alpha,\beta)] &= \sum_{j=1}^{\infty} jE[X_j] \\ &= E[X_1] + \sum_{j=2}^{\infty} jE[X_j] \end{aligned}$$

Substituting the values $E[X_1]$ and $E[X_j]$ from Lemma 2.2.9

$$\begin{aligned} E[\text{PHFCOST}(n,\alpha,\beta)] &\sim (1+\beta)n\ln\left(\frac{\beta+e^{-\alpha}}{\beta}\right) + (1+\beta)n\sum_{j=2}^{\infty}\left\{\frac{u_{j-2}^{j-1}-u_{j-1}^{j-1}}{j-1}\right\} \\ &= (1+\beta)n\left[\ln\left(\frac{\beta+e^{-\alpha}}{\beta}\right) + \sum_{j=1}^{\infty}\left\{\frac{u_{j-1}^{j}-u_{j}^{j}}{j}\right\}\right] \\ &= nZ_{\textit{non-minimal}}(\alpha,\beta). \end{aligned}$$

□

Lemma 2.2.10 *For minimal perfect hashing,* $E[X_1]$ *is* $\Theta(n(1+1/\alpha))$.

Proof: Searching for MPHF is performed by the second **for** loop in the procedure *Search*. The statements inside the **while** loop inside are executed exactly n times. The **for** loop is executed exactly n/α times. So $E[X_1]$ is $\Theta(n(1+1/\alpha))$. □

Theorem 2.2.6 *E[MPHFCOST*(n,α)*] for minimal perfect hashing is* $\Theta(n(1+1/\alpha)) + nZ(\alpha))$, *where*

$$Z(\alpha) = \sum_{j=1}^{\infty}\left\{\frac{t_{j-1}^{-j}-t_{j}^{-j}}{j}\right\},$$

and $E[TRIALS(n,\alpha)]$ *is* $n\aleph(\alpha)$ *where*

$$\aleph(\alpha) = \sum_{j=1}^{\infty}\left\{\frac{t_{j-1}^{-j}-t_{j}^{-j}}{j(j+1)}\right\}.$$

Proof:

The cost for MPHF can be computed by the Equation 2.15.

$$\begin{aligned} \text{MPHFCOST} &= \sum_{j=1}^{\infty} jX_j \\ E[\text{MPHFCOST}(n,\alpha)] &= \sum_{j=1}^{\infty} jE[X_j] \\ &= E[X_1] + \sum_{j=2}^{\infty} jE[X_j] \end{aligned}$$

Substituting the values of $E[X_1]$ and $E[X_j]$ from Lemmas 2.2.9, 2.2.10,

$$E[\text{MPHFCOST}(n,\alpha)] = \Theta(n(1+1/\alpha)) + (1_\beta)n\sum_{j=1}^{\infty}\left\{\frac{u_{j-1}^{j} - u_j^j}{j}\right\}$$

For minimal perfect hashing the value of β is 0. When β is 0, u_j becomes t_j^{-1} according to the definition of u_j in Lemma 2.2.9. Substituting 0 for β and t_j^{-1} for u_j, we get

$$\begin{aligned} E[\text{MPHFCOST}(n,\alpha)] &= \Theta(n(1+1/\alpha)) + n\sum_{j=1}^{\infty}\left\{\frac{t_{j-1}^{-j} - t_j^{-j}}{j}\right\} \\ &= \Theta(n(1+1/\alpha)) + nZ(\alpha) \end{aligned}$$

Similarly, the expected number of trials, TRIALS, can be computed by the Equation:

$$\text{TRIALS}(n,\alpha)] = \sum_{j=2}^{\infty} E[X_j]$$

Substituting the values of $E[X_j]$ from and substituting $\beta = 0$, Lemmas 2.2.9,

$$\begin{aligned} E[\text{TRIALS}(n,\alpha)] &= n\sum_{j=1}^{\infty}\left\{\frac{t_{j-1}^{-j} - t_j^{-j}}{j(j+1)}\right\} \\ &= n\aleph(\alpha) \end{aligned}$$

□

Some computed values of $Z(\alpha)$ are listed in Table 2.4 and plotted in Figure 2.4. We prove later in Lemma 2.2.19 that $Z(\alpha)$ has a lower bound of $\Omega(e^{\alpha})$.

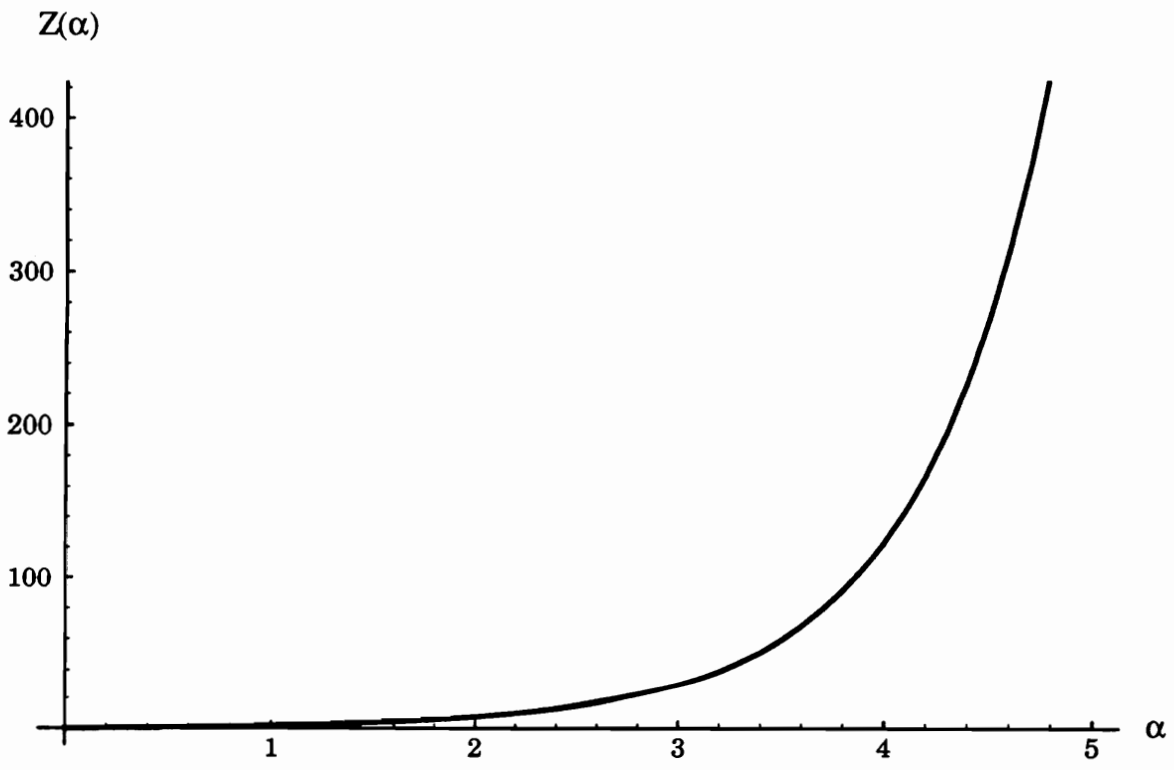


Figure 2.4: A plot of $Z(\alpha)$ vs. α.

α	Increments to α				
	+0.0	+0.1	+0.2	+0.3	+0.4
0.0	0.0	0.11	0.22	0.35	0.49
0.5	0.65	0.83	1.03	1.25	1.51
1.0	1.79	2.10	2.46	2.86	3.32
1.5	3.83	4.41	5.07	5.82	6.67
2.0	7.63	8.73	9.98	11.40	13.03
2.5	14.89	17.03	19.48	22.29	25.53
3.0	29.27	33.58	38.57	44.34	51.05
3.5	58.84	67.93	78.52	90.92	105.44
4.0	122.48	142.53	166.17	194.08	227.12
4.5	266.29	312.86	368.31	434.49	513.64

Table 2.4: Listing of the function $Z(\alpha)$.

2.2.3 A lower bound on MPHFCOST

In Section 2.2.2, we proved that, under the MOS model, the cost of finding an MPHF is $n(\Theta(1 + 1/\alpha) + Z(\alpha))$. In this section, we prove that $Z(\alpha)$ has a lower bound of $\Omega(e^{\alpha})$. This implies that if we attempt to decrease the space by increasing α up to $\ln n$, the running time increases to $O(n^2)$.

Let us define two terms v_j and w_j, which will be used in further analysis.

Definition 2.2.11 *For $j \geq 1$,*

$$v_j = e^{-\alpha}\frac{\alpha^j}{j!} - \frac{2}{(j+1)(j+2)}$$

Definition 2.2.12 *For $j \geq 0$,*

$$w_j = e^{-\alpha} + \sum_{i=1}^{j} v_i$$

Note that the above definition implies $w_0 = e^{-\alpha}$.

Lemma 2.2.11 *For $j \geq 0$, $w_j = t_j - j/(j+2)$.*

Proof: A calculation shows

$$\begin{aligned} w_j &= e^{-\alpha} + \sum_{i=1}^{j} v_i \\ &= e^{-\alpha} + \sum_{i=1}^{j} e^{-\alpha}\frac{\alpha^i}{i!} - \frac{2}{(i+1)(i+2)} \\ &= e^{-\alpha} + \sum_{i=1}^{j} e^{-\alpha}\frac{\alpha^i}{i!} - \sum_{i=1}^{j}\frac{2}{(i+1)(i+2)} \\ &= \sum_{i=0}^{j} e^{-\alpha}\frac{\alpha^i}{i!} - 2\sum_{i=1}^{j}\frac{1}{i+1} - \frac{1}{i+2} \\ &= t_j - 2\left(\frac{1}{2} - \frac{1}{j+2}\right) \\ &= t_j - \frac{j}{j+2}, \end{aligned}$$

as required by the lemma. □

Lemma 2.2.12 *If $\alpha \geq 2$, then $v_1 < 0$.*

Proof:

$$\begin{aligned} v_1 &= e^{-\alpha}\frac{\alpha^1}{1!} - \frac{2}{(1+1)(1+2)} \\ &= \frac{\alpha}{e^\alpha} - \frac{1}{3} \\ &= \frac{3\alpha - e^\alpha}{3e^\alpha}. \end{aligned}$$

Since $\alpha \geq 2$, we have $v_1 < 0$ as required by the lemma. □

Lemma 2.2.13 *There exists a $j \geq \alpha$ such that $v_i < 0$ for all $i \geq j$.*

Proof: Consider the value of $(j+1)(j+2)v_j$ as $j \to \infty$.

$$\begin{aligned} (j+1)(j+2)v_j &= \frac{e^{-\alpha}\alpha^j(j+1)(j+2)}{j!} - 2 \\ &= \left\{\frac{e^{-\alpha}\alpha^{j-2}}{(j-2)!}\right\}\alpha^2\left(1+\frac{2}{j-1}\right)\left(1+\frac{2}{j}\right) - 2 \\ &\to -2, \end{aligned}$$

as $j \to \infty$. Hence, the lemma follows. □

Lemma 2.2.14 *If* $1 \leq j \leq \alpha$ *and* $v_j < 0$, *then* $v_k < 0$ *for all* k *such that* $1 \leq k \leq j$.

Proof: The proof is by induction on k.

Base case: If $k = j$, then $v_j < 0$ by hypothesis.

Step: Suppose $v_i < 0$ for some $i \leq j$. We want to show that $v_{i-1} < 0$. Since $v_i < 0$,

$$e^{-\alpha}\frac{\alpha^i}{i!} < \frac{2}{(i+1)(i+2)}.$$

Multiplying by i/α on both sides, we obtain

$$\begin{aligned} e^{-\alpha}\frac{\alpha^{i-1}}{(i-1)!} &< \frac{2i}{\alpha(i+1)(i+2)} \\ &< \frac{2}{(i+1)(i+2)} \\ &< \frac{2}{(i-1)i} \end{aligned}$$

Hence, $v_{i-1} < 0$, as desired. □

Lemma 2.2.15 *If* $\alpha \leq j - 1$ *and* $v_j < 0$, *then* $v_k < 0$ *for all* $k \geq j$.

Proof: The proof is by induction on k.

Base Case: If $k = j$, then $v_j < 0$ by hypothesis.

Step: Suppose $v_i < 0$ for some $i \geq j$. We want to show that $v_{i+1} < 0$. $v_i < 0$ implies

$$e^{-\alpha}\frac{\alpha^i}{i!} < \frac{2}{(i+1)(i+2)}$$

Multiplying by $\alpha/(i+1)$ on both sides

$$e^{-\alpha}\frac{\alpha^{i+1}}{(i+1)!} < \frac{2}{\alpha}(i+1)^2(i+2).$$

Since $\alpha \leq i - 1$, this implies

$$\begin{aligned} e^{-\alpha}\frac{\alpha^{i+1}}{(i+1)!} &< \frac{i-1}{(i+1)^2(i+2)} \\ &< \frac{1}{i+2}\cdot\frac{i-1}{(i+1)^2} \\ &< \frac{1}{i+2}\cdot\frac{1}{i+3}. \end{aligned}$$

Hence, $v_{i+1} < 0$ as desired.

□

Lemma 2.2.16 *If $\alpha \geq 2$, $v_j \geq 0$ for $j = \lfloor \alpha \rfloor$.*

Proof:

$$\begin{aligned} v_j &= e^{-\alpha}\frac{\alpha^j}{j!} \\ &= e^{-\alpha}\frac{j^j}{j!}\left(1+\frac{\alpha-j}{j}\right)^j \\ &\geq e^{-j}\frac{j^j}{j!}e^{j-\alpha}(1+\alpha-j). \end{aligned}$$

Since $\alpha - j$ is at most 1, $e^{j-\alpha}(1+\alpha-j)$ is at least $2/e$. Hence

$$v_j \geq e^{-j}\frac{j^j}{j!}\frac{2}{e}$$

A calculation shows that for $j \geq 2$,

$$e^{-j}\frac{j^j}{j!}\frac{2}{e} > \frac{2}{(j+1)(j+2)}$$

Hence the lemma follows. □

Lemma 2.2.17 *If $\alpha \geq 2$, there exist two integers k_1 and k_2 such that $k_1 \leq \alpha \leq k_2$ and*

1. *$v_i < 0$ for $1 \leq i \leq k_1$,*
2. *$v_i > 0$ for $k_1 + 1 \leq i \leq k_2 - 1$,*
3. *$v_i < 0$ for $i \geq k_2$.*

Proof: According to Lemma 2.2.12, v_1 is negative. If v_i is negative for $1 \leq i \leq k_1 < \alpha$, and v_{k_1+1} is positive, then v_i is positive for $k_1 + 1 \leq i \leq \alpha$. Let k_2 be a number such that v_i is positive for $\alpha - 1 \leq i \leq k_2 - 1$ and v_{k_2} is negative. The existence of k_2 is guaranteed by Lemma 2.2.13. According to Lemma 2.2.15, v_i is negative for all $i \geq k_2$. □

Lemma 2.2.18 *For $\alpha \geq 2$, there exists a $k \geq \alpha$ such that w_i is negative for $1 \leq i < k$ and positive for $i \geq k$.*

Proof: For $\alpha \geq 2$

$$\begin{aligned} w_1 &= e^{-\alpha}(1+\alpha) - \frac{2}{3} \\ &< 0. \end{aligned}$$

Since

$$\begin{aligned} w_j &= e^{-\alpha} + \sum_{i=1}^{j} v_i \\ &= w_1 + \sum_{i=1}^{j} v_i, \end{aligned}$$

and $w_j \to 0$ as $j \to \infty$, at least some v_i are positive for $i > 1$. According to Lemma 2.2.17, there exist two integers k_1 and k_2 such that v_i is negative for $1 \leq i \leq k_1$, positive for $k_1 + 1 \leq i \leq k_2 - 1$, and negative for $i \geq k_2$. So w_i decreases initially, and then it increases, and then it decreases. w_i approaches 0 as $i \to \infty$. So $w_{k_2-1} > 0$. Since w_1 is negative we conclude that w_i is negative initially, i.e. for $1 \leq i \leq k$ for some k and positive $i \geq k$. □

Lemma 2.2.19 *$\aleph(\alpha)$ has a lower bound of $\Omega(e^{\alpha})$.*

Proof:

$$\aleph(\alpha) = \sum_{j=1}^{\infty} \left\{ \frac{t_{j-1}^{-j} - t_j^{-j}}{j(j+1)} \right\}$$

According to Lemma 2.2.18 there exists a k such that w_j is negative for $1 \leq j < k$, and positive for all $j \geq k$. Divide the series into three parts, terms with $j < k, j = k$, and $j > k$.

$$\begin{aligned} \aleph(\alpha) &= \sum_{j=1}^{k-1} \left\{ \frac{t_{j-1}^{-j} - t_j^{-j}}{j(j+1)} \right\} + \left\{ \frac{t_{k-1}^{-k} - t_k^{-k}}{k(k+1)} \right\} + \sum_{j=k+1}^{\infty} \left\{ \frac{t_{j-1}^{-j} - t_j^{-j}}{j(j+1)} \right\} \\ &= \frac{t_0^{-1}}{2} + \sum_{j=1}^{k-1} \left\{ \frac{t_j^{-(j+1)}}{(j+1)(j+2)} - \frac{t_j^{-j}}{j(j+1)} \right\} - \frac{t_k^{-k}}{k(k+1)} + \sum_{j=k+1}^{\infty} \left\{ \frac{t_{j-1}^{-j} - t_j^{-j}}{j(j+1)} \right\} \end{aligned}$$

Since $t_j < j/(j+2)$ for the terms in the first sum, they are all positive. Since $t_{j-1} < t_j$, all the terms in the second sum are positive. So both the first and the second sums are positive. This yields

$$\aleph(\alpha) > \frac{t_0^{-1}}{2} - \frac{t_k^{-k}}{k(k+1)}$$

Note that $t_k > k/(k+2)$. So

$$\begin{aligned} t_k^{-1} &< \frac{k+2}{k} \\ &= 1 + \frac{2}{k} \end{aligned}$$

Hence,

$$\begin{aligned} t_k^{-k} &< \left(1 + \frac{1}{k}\right)^k \\ &< e^2, \end{aligned}$$

and

$$\frac{t_k^{-k}}{k(k+1)} < \frac{e^2}{k(k+1)}.$$

Since $k \geq \alpha$,

$$\begin{aligned} \aleph(\alpha) &> \frac{t_0^{-1}}{2} - \frac{t_k^{-k}}{k(k+1)} \\ &> e^\alpha - e^2/\alpha \end{aligned}$$

and

$$\Omega(e^\alpha)$$

□

Theorem 2.2.7 *An MOS algorithm producing an MPHF, which occupies a space of* $n/(\eta \ln 2)$ *bits, requires at least* $\Omega(n^{1+\eta})$ *time.*

Proof: The array A requires $\log_2 n$ bits per bin. So the total number of bits required by A is $n \log_2 n/\alpha$. The lower bound on the number of bits is $n/\ln 2$ [32]. According to Definition 1.1.22, the efficiency of representing the MPHF f is

$$\begin{aligned}\eta &= \frac{n/\ln 2}{\text{Number of bits required by } A} \\ &= \frac{n/\ln 2}{n \log_2 n/\alpha} \\ &= \frac{\alpha}{\ln n}.\end{aligned}$$

Hence the expected execution time of the MOS algorithm T is

$$\begin{aligned}T &= T_m + T_o + T_s \\ &= \Theta(n) + \Theta(n) + E[\text{MPHFCOST}(\alpha)] \\ &> \Theta(n) + \Theta(n) + \Theta(n(1 + 1/\alpha) + n\aleph(\alpha)) \\ &= \Omega(ne^{\alpha}) \\ &= \Omega(n^{1+\eta})\end{aligned}$$

□

This equation gives the tradeoff between the running time of the algorithm and the space required to store it. If we try to represent the MPHF in the most compact manner, i.e., $n/\ln 2$ bits, the MOS algorithm requires $\Omega(n^2)$ time. The time complexity decreases as we allot more and more space to the PHF and approaches $\Omega(n)$ as the number of bits allotted to the PHF approaches $O(n \log n)$.

2.3 Experimental Results

A formula was derived for the searching time for minimal perfect hashing in Theorem 2.2.6. In this section, we apply the analysis of Section 2.2.2 to two implementations. The first implementation follows the code in Figures 2.1, 2.2, and 2.3. The second implementation [18] adds an optimization to the algorithms given in Figures 2.1, 2.2, and 2.3 by

maintaining a list of empty cells. We call this the *Optimized MOS (OMOS)* implementation. We modify our formula so that it takes care of this optimization and compare it with the performance of the actual implementation. We discuss the unoptimized implementation in Section 2.3.1 and the optimized implementation in Section 2.3.2.

2.3.1 Unoptimized implementation

We implemented the MOS algorithm exactly as described in Figure 2.1. It was run on 9 dictionaries of random strings. The size of the dictionary was varied from 10,000 to 90,000 in steps of 10,000. On each dictionary α was varied from 0.1 to 5 in steps of 0.1. As assumed in the analysis, the cost for each trial was counted as the number of words in the bin. For each dictionary and for each α, the following expression was calculated.

$$Z_{experimental} = \frac{\sum_{j=2}^{n} jX_j}{n}.$$

$nZ_{experimental}$ is the actual cost of tossing the words in bins of size greater than 1. X_j is defined in Definition 2.2.10. $Z_{experimental}$ is tabulated against the theoretical $Z(\alpha)$ in Tables 2.5 and 2.6. As can be seen from the tables, $Z(\alpha)$ predicts $Z_{experimental}$ very closely. Since X_1 is completely deterministic, it is not included in the comparison.

2.3.2 OMOS implementation

We apply our analysis to the MOS algorithm of Chen [18]. That algorithm has one significant optimization. In the searching step, the unoptimized MOS implementation processes the bins by tossing all the words in the bin randomly into B. The parameter i is set to 0 initially and is incremented until all the words land in empty slots. The OMOS implementation iterates i in a faster way by maintaining a list of empty slots in B. Since $h_2(x, i)$ is invertible, for any $x \in U$, it is possible to set i such that $h_2(x, i)$ points to any desired location. Suppose there are k words in a bin Q_r, and y is one of the words in the bin. i is iterated in such a way that $h_2(y, i)$ always maps to an empty slot of B. The trial is considered successful, if $h_2(x, i)$ maps to an empty slot for all the x in Q_r.

α	$Z(\alpha)$	$Z_{experimental}$ for different n and α								
		10,000	20,000	30,000	40,000	50,000	60,000	70,000	80,000	90,000
0.1	0.11	0.10	0.11	0.10	0.11	0.11	0.11	0.11	0.11	0.11
0.2	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.23	0.22	0.23
0.3	0.35	0.34	0.35	0.35	0.37	0.35	0.35	0.34	0.35	0.35
0.4	0.49	0.48	0.49	0.50	0.49	0.50	0.49	0.50	0.49	0.50
0.5	0.65	0.64	0.65	0.64	0.65	0.66	0.66	0.67	0.65	0.65
0.6	0.83	0.82	0.86	0.83	0.86	0.84	0.83	0.82	0.82	0.83
0.7	1.03	0.99	1.02	1.01	1.05	1.04	1.03	1.04	1.05	1.02
0.8	1.25	1.22	1.26	1.26	1.27	1.26	1.25	1.28	1.27	1.24
0.9	1.51	1.52	1.53	1.48	1.47	1.51	1.54	1.51	1.51	1.48
1.0	1.79	1.83	1.82	1.77	1.81	1.77	1.78	1.82	1.78	1.78
1.1	2.10	2.10	2.10	2.04	2.09	2.12	2.08	2.09	2.09	2.12
1.2	2.46	2.41	2.46	2.47	2.52	2.49	2.50	2.46	2.48	2.44
1.3	2.86	2.83	2.89	2.85	2.87	2.86	2.80	2.86	2.89	2.93
1.4	3.32	3.38	3.31	3.36	3.35	3.31	3.29	3.33	3.35	3.33
1.5	3.83	3.94	3.86	3.79	3.82	3.79	3.89	3.85	3.88	3.89
1.6	4.41	4.33	4.60	4.41	4.51	4.42	4.46	4.39	4.47	4.41

Table 2.5: Experimental verification of $Z(\alpha)$, for $0.1 \leq \alpha \leq 1.6$. n is the number of words in the dictionary.

α	$Z(\alpha)$	$Z_{experimental}$ for different n and α				
		10,000	20,000	30,000	40,000	50,000
1.7	5.07	4.99	4.99	4.94	5.00	5.05
1.8	5.82	6.07	5.94	5.97	5.75	5.78
1.9	6.67	6.83	6.88	6.79	6.57	6.67
2.0	7.63	7.97	7.54	7.57	7.85	7.70
2.1	8.73	8.70	8.79	8.52	8.63	8.89
2.2	9.98	9.99	10.03	9.79	9.95	9.77
2.3	11.40	11.07	11.50	11.33	11.34	11.44
2.4	13.03	13.77	13.17	13.06	13.05	13.02
2.5	14.89	14.05	14.27	15.21	15.01	14.78
2.6	17.03	16.27	16.42	17.52	17.30	17.02
2.7	19.48	19.97	19.99	19.44	19.17	19.10
2.8	22.29	22.09	22.39	22.36	22.27	22.19
2.9	25.53	24.47	24.52	24.97	26.34	25.40
3.0	29.27	29.26	28.99	28.61	29.01	29.41
3.1	33.58	32.04	33.95	33.16	33.90	35.38
3.2	38.57	39.42	36.78	37.64	38.75	39.17
3.3	44.34	43.05	45.14	43.85	44.02	42.29
3.4	51.05	52.92	49.74	51.24	49.48	50.29
3.5	58.84	53.49	60.46	55.18	60.45	59.80
3.6	67.93	67.02	64.84	64.71	69.02	65.63
3.7	78.52	80.54	87.32	81.05	77.29	79.64
3.8	90.92	84.92	92.39	92.72	92.30	88.61
3.9	105.44	102.37	104.07	110.90	105.87	102.44
4.0	122.48	126.82	118.36	116.15	126.46	127.56
4.1	142.53	133.67	139.82	132.76	148.04	141.27
4.2	166.17	167.49	171.55	170.19	170.38	174.01
4.3	194.08	179.09	201.02	197.22	194.86	195.54
4.4	227.12	222.27	218.39	214.60	228.50	222.38
4.5	266.29	279.42	270.39	276.87	263.50	264.71
4.6	312.86	348.58	311.78	316.96	331.98	320.83
4.7	368.31	349.64	379.09	383.98	373.86	369.12
4.8	434.49	422.70	434.60	388.98	440.77	457.02
4.9	513.64	548.18	480.30	520.88	516.27	513.63
5.0	608.49	601.55	639.44	625.68	633.04	593.18

Table 2.6: Experimental verification of $Z(\alpha)$, for $1.7 \leq \alpha \leq 5.0$. n is the number of words in the dictionary.

α	$Z(\alpha)$	$Z_{experimental}$ for different n and α			
		60,000	70,000	80,000	90,000
1.7	5.07	5.15	5.05	5.16	5.13
1.8	5.82	5.77	5.81	5.90	5.65
1.9	6.67	6.59	6.78	6.65	6.60
2.0	7.63	7.62	7.61	7.63	7.63
2.1	8.73	8.74	8.86	8.67	8.57
2.2	9.98	10.00	9.81	9.68	10.03
2.3	11.40	11.64	11.57	11.56	11.63
2.4	13.03	13.37	13.18	13.04	13.20
2.5	14.89	14.80	15.26	14.85	14.90
2.6	17.03	17.38	17.29	17.23	17.21
2.7	19.48	18.75	19.37	19.89	19.64
2.8	22.29	22.15	22.59	22.02	22.23
2.9	25.53	25.62	25.56	25.86	25.40
3.0	29.27	28.96	29.16	28.89	28.82
3.1	33.58	33.83	33.80	33.20	33.82
3.2	38.57	37.31	36.96	38.73	37.97
3.3	44.34	44.03	44.56	44.97	44.96
3.4	51.05	51.08	51.01	50.96	51.02
3.5	58.84	60.57	58.42	57.67	59.39
3.6	67.93	72.27	68.89	67.55	66.08
3.7	78.52	78.95	78.45	78.29	78.30
3.8	90.92	90.47	90.95	92.00	87.79
3.9	105.44	107.26	104.58	102.56	106.52
4.0	122.48	118.03	122.53	120.61	123.44
4.1	142.53	144.31	136.81	143.97	140.68
4.2	166.17	163.16	166.27	167.03	167.19
4.3	194.08	183.86	190.79	190.34	193.13
4.4	227.12	228.12	234.35	228.38	224.84
4.5	266.29	255.06	262.08	271.02	267.25
4.6	312.86	316.53	321.92	309.88	307.46
4.7	368.31	376.66	362.75	373.74	381.29
4.8	434.49	424.27	439.10	439.12	427.47
4.9	513.64	526.50	511.29	508.17	485.32
5.0	608.49	579.87	584.50	602.73	602.91

Table 2.7: Experimental verification of $Z(\alpha)$, for $1.7 \leq \alpha \leq 5.0$. n is the number of words in the dictionary.

For the purpose of analysis, we can consider this as a random toss of $k-1$ words into B instead of k words as in the MOS implementation, because one of the words, y, is always certain to land in an empty slot. Consequently, we need to modify our formula for calculating $E[X_j]$. When $j = 1$, there is no random tossing involved, and $E[X_1]$ is $\Theta(n)$. For $j > 1$, the new formula for calculating $E[X_j]$ is

$$E[X_j] = (1+\beta)^{j-1} n^{j-1} \int_0^1 \frac{(1-s)^{j-2}}{(j-2)!} \frac{s^{\beta n}}{1-s^j} \{\phi_4(s, j-1) - \phi_4(s, j)\}\, ds$$

Evaluating this integral in the same manner as in Lemma 2.2.9, we obtain

$$\begin{aligned} E[X_j] &\sim \frac{(1+\beta)n}{j} \left\{ \frac{u_{j-2}^{j-2} - u_{j-1}^{j-2}}{j-2} \right\} && \text{for } j > 2 \\ &\sim \frac{(1+\beta)n}{2} \ln \frac{\beta + e^{-\alpha}(1+\alpha)}{\beta + e^{-\alpha}} && \text{for } j = 2 \end{aligned}$$

Substituting this in Theorem 2.2.5, $E[\text{PHFCOST}(n, \alpha, \beta)]$ for this algorithm becomes $n(\Theta(1) + \Xi'(\alpha, \beta))$ where

$$\begin{aligned} \Xi'(\alpha, \beta) &= (1+\beta) \left\{ \ln \left(\frac{\beta + e^{-\alpha}(1+\alpha)}{\beta + e^{-\alpha}} \right) + \sum_{j=2}^{\infty} \left\{ \frac{u_{j-1}^{j-1} - u_j^{j-1}}{j-1} \right\} \right\} \\ &= (1+\beta) \left\{ \ln \left(\frac{\beta + e^{-\alpha}(1+\alpha)}{\beta + e^{-\alpha}} \right) + \sum_{j=1}^{\infty} \left\{ \frac{u_j^j - u_{j+1}^j}{j} \right\} \right\} \end{aligned}$$

For minimal perfect hashing, $\beta = 0$. Substituting it in the above expression,

$$\Xi(\alpha) = \ln(1+\alpha) + \sum_{j=1}^{\infty} \left\{ \frac{t_j^{-j} - t_{j+1}^{-j}}{j} \right\}$$

A plot of $\Xi(\alpha)$ for OMOS implementation is given in Figure 2.5.

Like the unoptimized MOS implementation, this formula for $\Xi(\alpha)$ was checked on nine dictionaries of sizes 100,000 to 900,000. On each dictionary, α was varied from 1 to 4 in steps of 0.1. For each bin, the number of trials was observed and the following expression was calculated.

$$\Xi_{experimental} = \frac{\sum_{j=2}^{n} j X_j}{n}$$

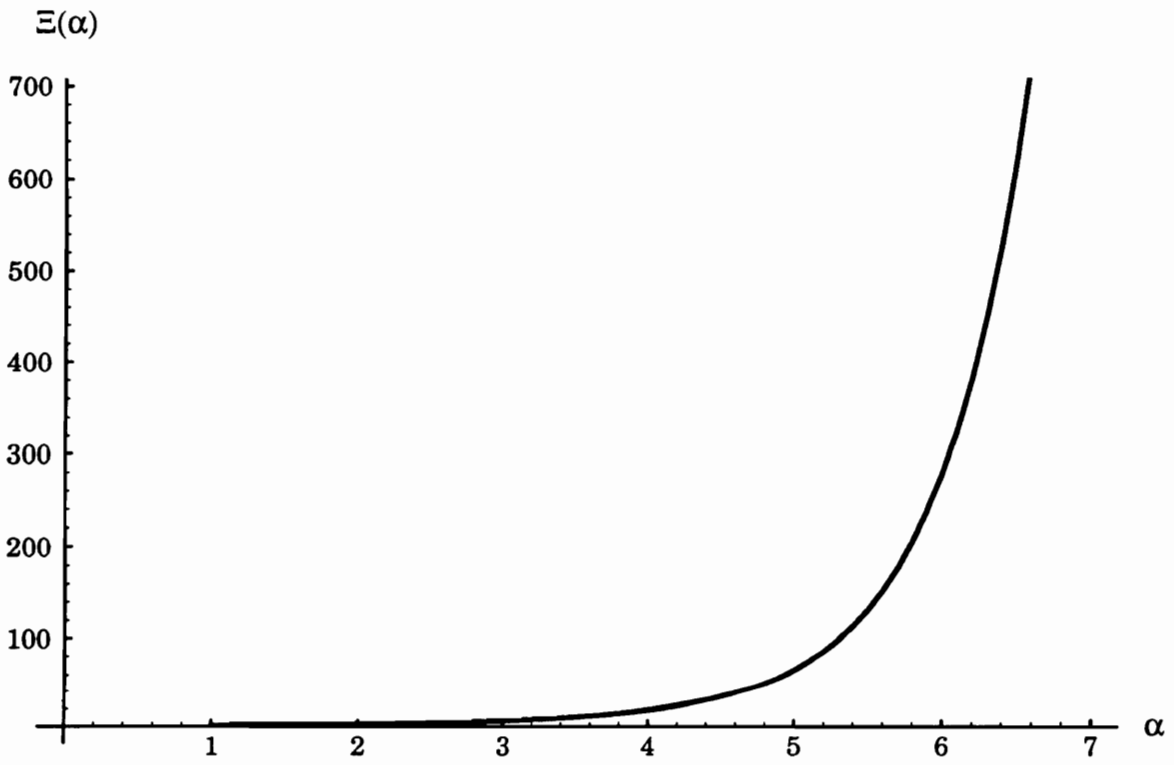


Figure 2.5: A plot of $\Xi(\alpha)$ vs. α.

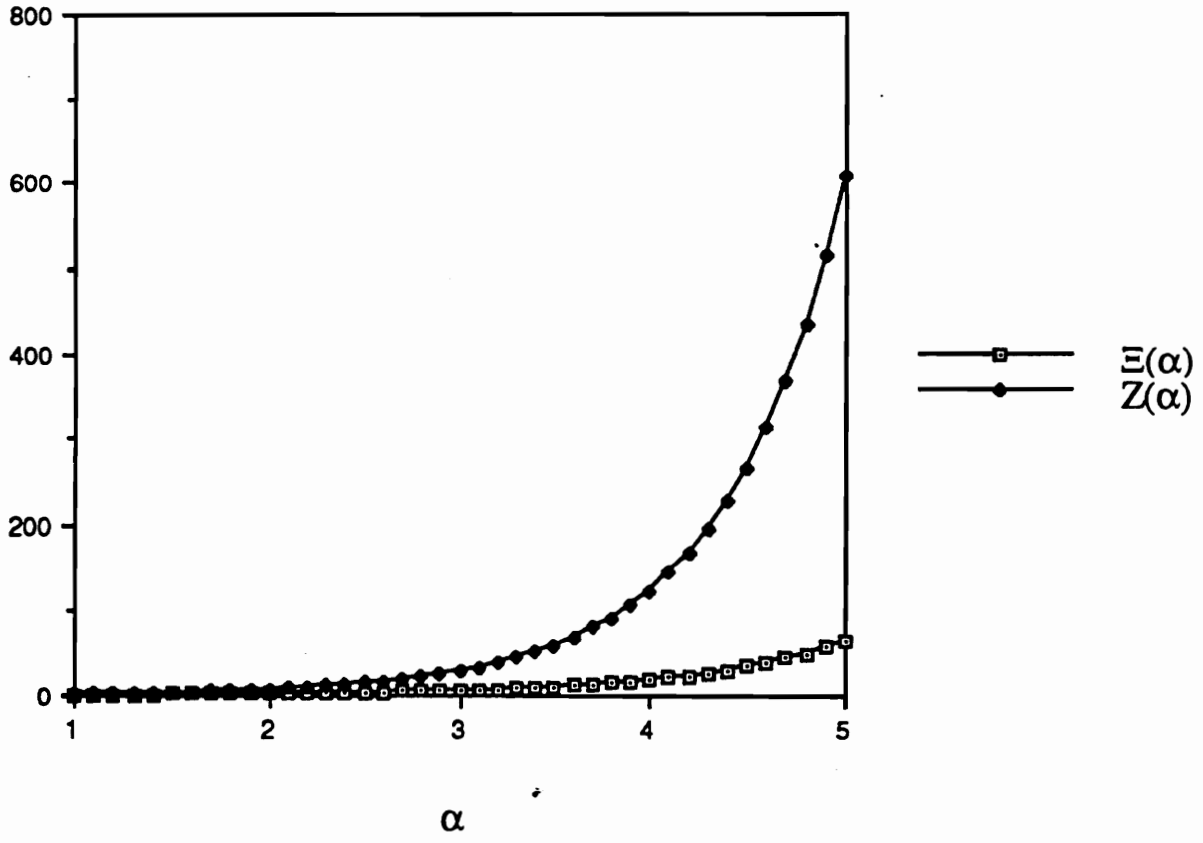


Figure 2.6: A comparison of $\Xi(\alpha)$ and $Z(\alpha)$.

α	$\Xi(\alpha)$	$Z_{experimental}$ for different n and α								
		10,000	20,000	30,000	40,000	50,000	60,000	70,000	80,000	90,000
1.00	1.06	1.10	1.08	1.06	1.04	1.06	1.06	1.07	1.06	1.06
1.10	1.18	1.31	1.22	1.22	1.21	1.24	1.23	1.21	1.22	1.24
1.20	1.31	1.50	1.36	1.38	1.38	1.37	1.40	1.37	1.39	1.39
1.30	1.44	1.67	1.53	1.53	1.55	1.55	1.54	1.53	1.56	1.56
1.40	1.59	1.84	1.70	1.70	1.74	1.73	1.72	1.70	1.71	1.72
1.50	1.74	2.00	1.92	1.88	1.87	1.86	1.91	1.87	1.88	1.87
1.60	1.90	2.15	2.07	2.04	2.03	2.05	2.05	2.03	2.06	2.06
1.70	2.08	2.28	2.25	2.20	2.17	2.19	2.22	2.21	2.21	2.23
1.80	2.27	2.49	2.39	2.38	2.34	2.39	2.35	2.35	2.37	2.36
1.90	2.47	2.64	2.56	2.47	2.60	2.53	2.51	2.53	2.54	2.53
2.00	2.69	2.91	2.74	2.73	2.70	2.70	2.71	2.68	2.71	2.68
2.10	2.93	3.09	2.95	2.93	2.94	2.97	2.97	2.97	2.96	2.94
2.20	3.20	3.42	3.18	3.27	3.19	3.28	3.30	3.23	3.27	3.28
2.30	3.49	3.79	3.52	3.55	3.54	3.59	3.58	3.58	3.56	3.54
2.40	3.81	3.96	3.96	3.88	3.92	3.97	3.98	3.95	3.86	3.91

Table 2.8: Experimental verification of $\Xi(\alpha)$, for $1.0 \leq \alpha \leq 2.4$. n is the number of words in the dictionary.

The experimentally observed values are a little higher than what $\Xi(\alpha)$ predicts. This is probably due to the fact that function h_2 used in the algorithm is not exactly random. Tables 2.8 and 2.9 lists the experimental observations.

α	$\Xi(\alpha)$	$Z_{experimental}$ for different n and α								
		10,000	20,000	30,000	40,000	50,000	60,000	70,000	80,000	90,000
2.50	4.16	4.30	4.31	4.23	4.30	4.24	4.30	4.28	4.28	4.30
2.60	4.55	4.86	4.60	4.57	4.69	4.61	4.70	4.70	4.68	4.66
2.70	4.98	5.34	4.91	5.18	5.17	5.17	5.08	5.05	5.12	5.14
2.80	5.45	5.83	5.53	5.48	5.63	5.52	5.48	5.58	5.64	5.64
2.90	5.99	6.39	6.01	6.04	5.99	6.11	6.03	5.94	6.09	6.06
3.00	6.58	6.98	6.72	6.63	6.71	6.45	6.63	6.56	6.62	6.59
3.10	7.24	7.38	7.44	7.09	7.11	7.20	7.27	7.25	7.41	7.32
3.20	7.99	8.40	7.70	8.06	7.96	8.12	7.92	7.94	7.83	8.08
3.30	8.82	9.16	8.98	9.10	8.65	8.81	8.79	8.80	8.83	8.76
3.40	9.76	9.87	9.99	9.74	9.52	9.77	9.76	9.82	9.62	9.73
3.50	10.83	11.50	10.70	10.93	10.83	11.04	10.91	10.79	10.98	10.90
3.60	12.03	11.81	12.04	11.77	12.43	12.10	11.96	12.07	12.03	11.73
3.70	13.38	13.92	13.32	13.49	13.37	13.43	13.33	13.65	13.39	13.52
3.80	14.92	15.28	14.63	15.09	14.92	15.02	14.99	15.11	14.92	14.96
3.90	16.67	16.18	17.40	16.42	16.78	16.35	16.82	16.81	16.70	16.66
4.00	18.66	18.19	18.51	19.02	18.87	18.99	18.93	18.78	18.67	18.24
4.10	20.94	21.13	21.00	20.72	20.67	20.75	21.21	20.92	20.76	20.78
4.20	23.53	23.94	23.00	22.78	24.10	23.41	22.82	23.39	23.52	23.42
4.30	26.51	24.90	29.15	26.29	25.78	26.43	26.32	26.66	25.61	27.04
4.40	29.92	28.09	30.02	28.65	28.75	29.15	29.80	29.69	29.29	30.08
4.50	33.84	30.40	32.07	34.01	31.94	33.30	32.77	34.03	34.15	33.32
4.60	38.35	36.43	36.30	36.62	37.58	38.98	38.35	36.94	37.97	38.70
4.70	43.56	41.46	44.35	44.07	42.32	42.75	43.43	43.37	42.13	42.90
4.80	49.59	45.36	48.05	49.49	48.86	50.28	48.75	49.02	49.04	48.45

Table 2.9: Experimental verification of $\Xi(\alpha)$, for $2.5 \leq \alpha \leq 4.8$. n is the number of words in the dictionary.

Chapter 3

MC ALGORITHM

In Chapter 2, we discussed the MOS algorithm for perfect hashing. In this chapter, we discuss an alternate perfect hashing algorithm called the *musical chairs algorithm (MC)* and experimentally compare its performance with the MOS algorithm.

Both the MOS and MC algorithms can be applied to sets of either fixed-length or variable length words. For both S_f and S_v, either in the OP case or NOP case, and in its entire range, the MC algorithm runs faster than the MOS algorithm. The LSL (lower space limit) of the MOS algorithm is lower than the LSL of the MC algorithm for both OP and NOP cases. However, as proved in Chapter 2, running the MOS algorithm at its LSL, i.e., at $\alpha = \ln n$, requires $O(n^2)$ time. In practice, it is too expensive to run the MOS algorithm for $\alpha > 5.0$. Table 3.1 compares the important characteristics of the MOS algorithm and MC algorithm.

The remainder of this chapter is organized as follows. Section 3.1 gives a description of the MC algorithm. Section 3.2 develops a basis for comparison of the space requirements of the MOS and MC algorithms. Section 3.3 gives experimental results of the MC algorithm and compares the results for the MOS algorithm.

3.1 A Description of the MC Algorithm

3.1.1 Form of the PHF for various cases

The MC algorithm can be applied to either S_f or S_v to produce either an OPPHF or a NOPPHF. This gives four cases, which we describe now. In every case, it maintains an array A of size N, and h_1, h_2, and h_3 denote random functions that map U to I_N.

Set	Order	Running time for a PHF of the same space	LSL of MOS	Space of MOS for $\alpha = 5$	LSL of MC
S_f	OP	MC runs faster	$n \log_2 n + n/\ln 2$	$1.2n \log_2 n$	$1.25n \log_2 n$
S_f	NOP	MC runs faster	$n/\ln 2$	$0.2n \log_2 n$	$0.25nb_f$
S_v	OP	MC runs faster	$n \log_2 m + n/\ln 2$	$n \log_2 m + 0.2n \log_2 n$	$1.25n \log_2 m$
S_v	NOP	MC runs faster	$0.5n \log_2 n + n/\ln 2$	$0.7n \log_2 n$	$0.6125n \log_2 n$

Table 3.1: Comparison of the MC algorithm with the MOS algorithm.

Case 1: OP for S_f

In this case, the MC algorithm maintains two arrays, A and $\tilde{B}$. A is an array of N integers. $\tilde{B}$ is an array of size n, and the words of S_f are stored in $\tilde{B}$. The MC algorithm seeks a PHF of the form

$$f(x) = (A[h_1(x)] + A[h_2(x)] + A[h_3(x)]) \pmod{n}. \tag{3.1}$$

Figure 3.1 lists a program for checking the membership of a given word x in S_f in the OP case.

Case 2: NOP for S_f

In this case, the MC algorithm maintains only one array A and the words of S_f are directly stored in A. To answer the question whether a given $x \in S_f$, we simply check whether it is equal to one of $A[h_1(x)]$, $A[h_2(x)]$, or $A[h_3(x)]$. This requires at most three accesses to the array A. Figure 3.2 lists a program for checking the membership of a given word x in S_f in the NOP case.

Find-S_f-OP(x)

{

/* Calculate the address of x */

$a = (h_1(x) + h_2(x) + h_3(x)) \pmod{n}$;

/* Compare the word at this address in C with x */

if ($\tilde{B}[a] == x$)
return (TRUE);
else
return (FALSE);
endif
}

Figure 3.1: The *find* program for S_f in the OP case.

Find-S_f-NOP(x)

{

/* Check if x is equal to one of $h_1(x)$, $h_2(x)$, or $h_3(x)$ */

if ($x == A[h_1(x)]$ **or** $A[h_2(x)]$ **or** $A[h_3(x)]$)
return (TRUE);
else
return (FALSE);
endif
}

Figure 3.2: The *find* program for the S_f in the NOP case.

Case 3: OP for S_v

In this case, the MC algorithm maintains two arrays, A and C. A is an array of integers, and C is an array of characters of size m. The words of S_v are stored sequentially in C. The MC algorithm seeks a PHF of the form

$$f(x) = (A[h_1(x)] + A[h_2(x)] + A[h_3(x)]) \pmod{m}. \tag{3.2}$$

Figure 3.3 lists a program for checking the membership of a given word x in S_v in the OP case.

Case 4: NOP for S_v

In this case, the MC algorithm maintains two arrays, A and C. A is an array of integers of size N and C is an array of characters of size m. The words of S_v are stored sequentially C according to the partial order produced by a random function h_0 that maps U to I_m. So, as discussed in Section 1.2, a hash pointer can be used to point to a word in C. The address of x can be calculated according to the formula

$$\mathit{Address}(x) = h_0(x) + \mathit{Offset}(x).$$

For $1 \leq i \leq n$, let the maximum value of $\mathit{Offset}(x)$ be $\hat{n}$. The MC algorithm seeks a PHF of the form:

$$f(x) = h_0(x) + (A[h_1(x)] + A[h_2(x)] + A[h_3(x)]) \pmod{\hat{n}}. \tag{3.3}$$

Theorem 1.2.2 implies that with very high probability the value of $\log_2 \hat{n}$ is $\log_2 n/2$. Hence the total space occupied by A in this case is only half of the total space occupied by A in other cases. Figure 3.4 outlines a program for checking the membership of a given word x in S_v in the NOP case.

Substituting $x_i \in S$ for x in Equation 3.1 or 3.2 or 3.3 for $1 \leq i \leq n$ yields a system of n linear equations with N variables (the values in the array A). This observation was made independently by Seider and Hirschberg [37]. They propose solving a system of linear

Find-S_v-OP(x)

{

/* Calculate the address of x */

$a = (h_1(x) + h_2(x) + h_3(x)) \pmod{m}$;

/* Compare the word at this address in C with x */

if $(C[a..a + |x|] == x)$
return (TRUE);
else
return (FALSE);
endif
}

Figure 3.3: The *find* program for S_v in the OP case.

Find-S_v-NOP(x)

{

/* Calculate the address of x */

$a = h_0(x) + (h_1(x) + h_2(x) + h_3(x)) \pmod{\hat{n}}$;

/* Compare the word at this address in C with x */

if $(C[a..a + |x|] == x)$
return (TRUE);
else
return (FALSE);
endif
}

Figure 3.4: The *find* program for S_v in the NOP case.

equations of order $n \times N$ which is an $O(n^3)$ operation. Equations 3.1, 3.2, 3.3 contain only three variables. This special structure allows us to solve these equations with the help of the MC algorithm given in Figure 3.5. The same algorithm in Figure 3.5 applies to all the above cases.

Definition 3.1.1 *A* 3-regular hypergraph $G = (V, E)$ *consists of a set of vertices* V, *and a set of* hyperedges E, *where each hyperedge is a subset of* V *of cardinality 3. Hence* $\{v_1, v_2, v_3\}$ *may be a hyperedge of* G, *if* v_1, v_2, v_3 *are all distinct, and* $v_1, v_2, v_3 \in V$.

Definition 3.1.2 *A 3-regular hypergraph* $G' = (V', E')$ *is a* subhypergraph *of* $G = (V, E)$, *if* $V' \subseteq V$ *and* $E' \subseteq E$.

Definition 3.1.3 *An edge* $e = \{v_1, v_2, v_3\}$ *is* incident *on a vertex* v, *if* $v \in e$.

Definition 3.1.4 *In a 3-regular hypergraph, the* degree *of a vertex* $v \in V$ *is the number of edges incident on* v. *That is,*

$$degree(v) \quad = \quad |\{e \in E : \ v \in e\}| \, .$$

Definition 3.1.5 *A hyperedge* $e = \{v_1, v_2, v_3\}$ *in a hypergraph is a leaf, if the degree of at least one of its vertices is one.*

3.1.2 The MC algorithm

The MC algorithm consists of three phases.

The first phase places all the words $x_i \in S$ in a circular list and constructs a 3-regular hypergraph $G = (V, E)$, where $V = I_N$ and

$$E = \{\{h_1(x_i), h_2(x_i), h_3(x_i)\} : 1 \leq i \leq n \text{ and } x_i \in S\} \, .$$

The vertices of G correspond to the cells of A and the hyperedges of G correspond to words in S. The MC algorithm is based on the following observation. Suppose a word $x \in S$ corresponds to a hyperedge $\{v_1, v_2, v_3\}$ in G and it is a leaf. Without loss of generality, let

the degree of v_1 be one. Note that the value of $A[v_1]$ affects only x. This means that if we delete x from G and assign values for rest of the vertices of G, we can assign a value to $A[v_1]$ such that $f(x)$ maps to the address of x.

The second phase orders the words by traversing the circular list while deleting leaves from G and pushing the corresponding words onto a stack. At any point, if the MC algorithm finds subhypergraph of G with no leaves, it reports FAILURE and quits. The MC algorithm succeeds if we can delete all the hyperedges from G while traversing the circular list.

The third phase pops each word x from the stack and calls the appropriate *process* procedure depending upon the case. The degree of at least one of the vertices $h_1(x)$, $h_2(x)$, and $h_3(x)$ is guaranteed to be one before popping x from the stack because $\{h_1(x), h_2(x), h_3(x)\}$ was a leaf when x was pushed onto the stack. Let i be one of the vertices $h_1(x)$, $h_2(x)$, or $h_3(x)$ such that the degree of i is one. $A[i]$ is assigned a value depending upon the case in the following manner. In *Process-S_f-NOP*, x is stored in the cell $A[i]$. In *Process-S_f-OP*, $A[i]$ is adjusted such that

$$(A[h_1(x)] + A[h_2(x)] + A[h_3(x)]) \bmod n = l$$

where l is the address of x in $\tilde{B}$. In *Process-S_v-OP*, $A[i]$ is adjusted such that

$$(A[h_1(x)] + A[h_2(x)] + A[h_3(x)]) \bmod m = l$$

where l is the address of x in C. In *Process-S_v-NOP*, $A[i]$ is adjusted such that

$$h_0(x) + (A[h_1(x)] + A[h_2(x)] + A[h_3(x)]) \bmod \hat{n} = l$$

where l is the address of x in C. The third phase always succeeds.

Example 3.1.1 In this example, the MC algorithm is illustrated for the OP case for S_v. Let S_v be a set of 8 words "USA", "Russia", "Britain", "France", "Germany", "India", "China" and "Japan". The characters of all these words are stored sequentially in an array C, and each word is terminated by a special character Λ. The size of the array C is thus 52. The values of $h_1(x)$, $h_2(x)$, and $h_3(x)$ and the address of x in C for all these words is

```
MC()

{

    /*
    Make a hyper-graph of N vertices and n 3-tuples {h1(x), h2(x), h3(x)}, and make
    circular list of all the words in S
    */

    CreateEmptyGraph(Hypergraph);
    CreateEmptyCircularList(CircularList);
    for(all x ∈ S) {
        AddEdge({h1(x), h2(x), h3(x)},Hypergraph);
        AddWord(x,CircularList);
    }

    /* Traverse the circular list and transfer all the leaves to the stack */

    while ( !Empty(CircularList)) {

        if (NoMoreLeaves(Hypergraph) )
            return(FAILURE);

        x = GetWord(CircularList);
        if (Leaf({h1(x), h2(x), h3(x)}, Hypergraph) {
            Push(x,Stack);
            DeleteEdge({h1(x), h2(x), h3(x)}, Hypergraph);
            DeleteWord(x,CircularList);
        }
    }
```

[Continued on next page]

```
        /*
        Pop the words from the stack and call the appropriate process routine, depending
        upon the case.
        */

        while ( !Empty(Stack)) {
                x = Pop(Stack);
                case  {
                        S_f,OP:     Process-S_f-OP (h_1(x), h_2(x), h_3(x), x);
                        S_f,NOP:    Process-S_f-NOP (h_1(x), h_2(x), h_3(x), x);
                        S_v,OP:     Process-S_v-OP (h_1(x), h_2(x), h_3(x), x);
                        S_v,NOP:    Process-S_v-NOP (h_1(x), h_2(x), h_3(x), x);
                }
        }
        return(SUCCESS);
}
```

Figure 3.5: The MC algorithm.

```
Process-S_f-NOP(i, j, k, x)
{
        /* Store x in one of the free cells A[i], A[j], or A[k]. */

        if (Degree(i) == 1 )
                A[i] = x;
        else if (Degree(j) == 1 )
                A[j] = x;
        else if (Degree(k) == 1 )
                A[k] = x;
}
```

Figure 3.6: The process procedure for S_f in the NOP case.

```
Process-S_f-NOP(i, j, k, x)
{
        l = Address(x, B̃);

        /*
        Change one of the variables A[i], A[j], or A[k] such that

                                (A[i] + A[j] + A[k])  (mod n) = l.
        */

        if (Degree(i) == 1 )
                A[i] = (l − A[j] − A[k])  (mod n);
        else if (Degree(j) == 1 )
                A[j] = (l − A[i] − A[k])  (mod n);
        else if (DegreeA(k) == 1 )
                A[k] = (l − A[j] − A[i])  (mod n);
}
```

Figure 3.7: The process procedure for S_f in the OP case.

Process-S_v-OP(i, j, k, x)
{

l = Address(x, C);

/*
Change one of the variables $A[i]$, $A[j]$, or $A[k]$ such that

$$(A[i] + A[j] + A[k]) \pmod{m} = l.$$

*/

if (Degree(i) == 1)
$A[i] = (l - A[j] - A[k]) \pmod{m}$;
else if (Degree(j) == 1)
$A[j] = (l - A[i] - A[k]) \pmod{m}$;
else if (Degree$A(k)$ == 1)
$A[k] = (l - A[j] - A[i]) \pmod{m}$;
}

Figure 3.8: The process procedure for S_v in the NOP case.

Process-S_v-NOP(i, j, k, x)
{

l = Address$(x, C) - h_0(x)$;

/*
Change one of the variables $A[i]$, $A[j]$, or $A[k]$ such that

$$(A[i] + A[j] + A[k]) \pmod{\hat{n}} = l.$$

*/

if (Degree(i) == 1)
$A[i] = (l - A[j] - A[k]) \pmod{\hat{n}}$;
else if (Degree(j) == 1)
$A[j] = (l - A[i] - A[k]) \pmod{\hat{n}}$;
else if (Degree$A(k)$ == 1)
$A[k] = (l - A[j] - A[i]) \pmod{\hat{n}}$;
}

Figure 3.9: The process procedure for S_v in the OP case.

x	$h_1(x)$	$h_2(x)$	$h_3(x)$	Address of x in C
"USA"	0	5	3	0
"Russia"	9	6	0	4
"Britain"	4	6	5	11
"France"	2	0	8	19
"Germany"	3	4	0	26
"India"	9	4	1	34
"China"	8	1	7	40
"Japan"	3	7	9	46

Table 3.2: An example illustrating the MC algorithm.

listed in Table 3.2. h_1, h_2, and h_3 are random functions which map to an integer between 0 and 9 with equal probability. Section 1.2 defines random functions precisely.

The following is a list of all the steps in the MC algorithm while traversing the circular list. In each step, we give all the words in the circular list and underline the current word under consideration. We also give the degrees of all 10 vertices in each step and underline the vertices of the current hyperedge.

1. Circular List = { "USA", "Russia", "Britain", "France", "Germany", "India", "China", "Japan"}.
   Hyperedge = {0, 5, 3}.
   Degrees = [4, 2, 1, 3, 3, 2, 2, 2, 2, 3].

2. Circular List = { "USA", "Russia", "Britain", "France", "Germany", "India", "China", "Japan"}.
   Hyperedge = {9, 6, 0}.
   Degrees = [4, 2, 1, 3, 3, 2, 2, 2, 2, 3].

3. Circular List = { "USA", "Russia", "Britain", "France", "Germany", "India", "China", "Japan"}.

Hyperedge = {4, 6, 5}.

Degrees = [4, 2, 1, 3, 3, 2, 2, 2, 2, 3].

4. Circular List = {"USA", "Russia", "Britain", "France", "Germany", "India", "China", "Japan"}.

   Hyperedge = {2, 0, 8}.

   Degrees = [4, 2, 1, 3, 3, 2, 2, 2, 2, 3].

   Since the degree of vertex 2 is 1, "France" is removed from the circular list.

5. Circular List = {"USA", "Russia", "Britain", "Germany", "India", "China", "Japan"}.

   Hyperedge = {3, 4, 0}.

   Degrees = [3, 2, 0, 3, 3, 2, 2, 2, 1, 3].

6. Circular List = {"USA", "Russia", "Britain", "Germany", "India", "China", "Japan"}.

   Hyperedge = {9, 4, 1}.

   Degrees = [3, 2, 0, 3, 3, 2, 2, 2, 1, 3].

7. Circular List = {"USA", "Russia", "Britain", "Germany", "India", "China", "Japan"}.

   Hyperedge = {8, 1, 7}.

   Degrees = [3, 2, 0, 3, 3, 2, 2, 2, 1, 3].

   Since the degree of vertex 8 is 1, "China" is removed from the circular list.

8. Circular List = {"USA", "Russia", "Britain", "Germany", "India","Japan"}.

   Hyperedge = {3, 7, 9}.

   Degrees = [3, 1, 0, 3, 3, 2, 2, 1, 0, 3].

   Since the degree of vertex 7 is 1, "Japan" is removed from the circular list.

9. Circular List = {"USA", "Russia", "Britain", "Germany", "India"}.

   Hyperedge = {0, 5, 3}.

   Degrees = [3, 1, 0, 2, 3, 2, 2, 0, 0, 2].

10. Circular List = {"USA", "Russia", "Britain", "Germany", "India"}.

    Hyperedge = {9, 6, 0}.

    Degrees = [3, 1, 0, 2, 3, 2, 2, 0, 0, 2].

11. Circular List = {"USA", "Russia", "Britain", "Germany", "India"}.

    Hyperedge = {4, 6, 5}.

    Degrees = [3, 1, 0, 2, 3, 2, 2, 0, 0, 2].

12. Circular List = {"USA", "Russia", "Britain", "Germany", "India"}.

    Hyperedge = {3, 4, 0}.

    Degrees = [3, 1, 0, 2, 3, 2, 2, 0, 0, 2].

13. Circular List = {"USA", "Russia", "Britain", "Germany", "India"}.

    Hyperedge = {9, 4, 1}.

    Degrees = [3, 1, 0, 2, 3, 2, 2, 0, 0, 2].

    Since the degree of vertex 1 is 1, "India" is removed from the circular list.

14. Circular List = {"USA", "Russia", "Britain", "Germany", }.

    Hyperedge = {0, 5, 3}.

    Degrees = [3, 0, 0, 2, 2, 2, 2, 0, 0, 1].

15. Circular List = {"USA", "Russia", "Britain", "Germany", }.

    Hyperedge = {9, 6, 0}.

    Degrees = [3, 0, 0, 2, 2, 2, 2, 0, 0, 1].

    Since the degree of vertex 9 is 1, "Russia" is removed from the circular list.

16. Circular List = {"USA", "Britain", "Germany", }.

    Hyperedge = {4, 6, 5}.

    Degrees = [2, 0, 0, 2, 2, 2, 1, 0, 0, 0].

    Since the degree of vertex 6 is 1, "Britain" is removed from the circular list.

17. Circular List = {"USA", "Germany", }.

    Hyperedge = {3, 4, 0}.

    Degrees = [2, 0, 0, 2, 1, 1, 0, 0, 0, 0].

    Since the degree of vertex 4 is 1, "Germany" is removed from the circular list.

18. Circular List = {"USA", }.

    Hyperedge = {0, 5, 3}.

    Degrees = [1, 0, 0, 1, 0, 1, 0, 0, 0, 0].

    Since the degree of vertex 0 is 1, "USA" is removed from the circular list.

The words in S_v are pushed onto the stack in the order "France", "China", "Japan", "India", "Russia", "Britain", "Germany", and "USA". The next phase consists of popping all the words from the stack and assigning values to the cells of A. We denote all the unassigned cells by a *, and underline the cells of the current hyperedge.

1. Word = "USA", Hyperedge = {0, 5, 3 }, Address = 0

   A = [*, *, *, *, *, *, *, *, *, *]

   We need to assign values to $A[0]$, $A[3]$, and $A[5]$ such that $(A[0]+A[3]+A[5]) \bmod 52$ is 0. Since $A[0]$, $A[3]$, and $A[5]$ are unassigned, we assign 0 to all of them.

2. Word = "Germany", Hyperedge = {3, 4, 0 }, Address = 26

   A = [0, *, *, 0, *, 0, *, *, *, *]

   We need to assign values to $A[0]$, $A[3]$, and $A[4]$ such that $(A[0]+A[3]+A[4]) \bmod 52$ is 26. $A[0]$ is 0 and $A[3]$ is 0. So we assign 26 to $A[4]$.

3. Word = "Britain", Hyperedge = {4, 6, 5 }, Address = 11

   A = [0, *, *, 0, 26, 0, *, *, *, *]

We need to assign values to $A[4]$, $A[5]$, and $A[6]$ such that $(A[4] + A[5] + A[6])$ mod 52 is 11. $A[4]$ is 26 and $A[5]$ is 0. So So we assign 37 to $A[6]$. The rest of the steps are similarly carried out.

4. Word = "Russia", Hyperedge = {9, 6, 0 }, Address = 4
   A = [0, *, *, 0, 26, 0, 37, *, *, *]

5. Word = "India", Hyperedge = {9, 4, 1 }, Address = 34
   A = [0, *, *, 0, 26, 0, 37, *, *, 19]

6. Word = "Japan", Hyperedge = {3, 7, 9 }, Address = 46
   A = [0, 41, *, 0, 26, 0, 37, *, *, 19]

7. Word = "China", Hyperedge = {8, 1, 7 }, Address = 40
   A = [0, 41, *, 0, 26, 0, 37, 27, *, 19]

8. Word = "France", Hyperedge = {2, 0, 8 }, Address = 19
   A = [0, 41, *, 0, 26, 0, 37, 27, 24, 19]

Finally $A[2]$ is set to 47. □

Definition 3.1.6 *Define* $\gamma = N/n$.

Definition 3.1.7 *μ is the ratio of the total number of links traversed on the circular list during the MC algorithm to n.*

Since traversing the circular list is the dominant operation in the MC algorithm, we may consider μ as a measure of the running time of the MC algorithm. The MC algorithm

fails if we find that there are no edges in the 3-regular hypergraph that are leaves. The MC algorithm succeeds if and only if every subgraph of G has at least one leaf. An interesting question is, for what value of N does the MC algorithm succeed with high probability? We found experimentally that the MC algorithm almost always fails when $\gamma < 1.21$ and almost always succeeds when $\gamma > 1.24$. Also, as γ varies from 1.25 to 2, μ decreases from 6.01 to 1.37 monotonically. We discuss these experiments in more detail in Section 3.3. Majewski [31] independently provides an experimental verification that when $\gamma = 1.23$, every subgraph of a 3-regular hypergraph has a leaf vertex. However, he does not provide a linear-time algorithm for getting a PHF from this 3-regular hypergraph.

3.2 The Space Requirements of the MC and MOS Algorithms

In this section, we compare the space requirements of the MC and MOS algorithms. Since the number and size of the arrays used depends on the case, the total space occupied by these arrays varies accordingly. For each case, we calculate the total space occupied by these data structures and derive a relationship between the parameters of the two algorithms such that the total space occupied by the arrays of the two algorithms is the same. Table 3.3 gives a brief summary of the arrays used by each algorithm and the total space occupied by them. We refer to the PHF found by the MOS algorithm as f_{MOS} and the PHF found by the MC algorithm as f_{MC}.

Case 1: OP for S_f

In the OP case for S_f, the MOS algorithm employs three arrays—A, B, and $\tilde{B}$. The array A has n/α cells, and each cell occupies $\lceil \log_2 n \rceil$ bits. The array B has n cells. Assuming it uses simple pointers, each cell occupies $\lceil \log_2 n \rceil$ bits. The array $\tilde{B}$ has n cells, and each cell occupies b_f bits. So,

$$\text{SPACE}(\langle S_f, f_{MOS} \rangle) = n\left(\frac{\lceil \log_2 n \rceil}{\alpha} + \lceil \log_2 n \rceil\right) + nb_f.$$

Case	Arrays used by the MOS algorithm	Space Occupied by the MOS data structures	Arrays used by the MC algorithm	Space required by the MC data structures
S_f, OP	A, $\|A\| = n/\alpha$ B, $\|B\| = n$ C, $\|C\| = n$	$n\log_2 n/\alpha+$ $n\log_2 n+$ nb_f	A, $\|A\| = \gamma n$ C, $\|C\| = n$	$\gamma n\log_2 n$ + nb_f
S_f, NOP	A, $\|A\| = n/\alpha$ B, $\|B\| = n$	$n\log_2 n/\alpha+$ nb_f	A, $\|A\| = \gamma n$	γnb_f
S_v, OP	A, $\|A\| = n/\alpha$ B, $\|B\| = n$ C, $\|C\| = m$	$n\log_2 n/\alpha+$ $n\log_2 m+$ mb_v	A, $\|A\| = \gamma n$ C, $\|C\| = m$	$\gamma n\log_2 m$ + mb_v
S_v, NOP	A, $\|A\| = n/\alpha$ B, $\|B\| = n$ C, $\|C\| = m$	$n\log_2 n/\alpha+$ $n\log_2 n/2+$ mb_v	A, $\|A\| = \gamma n$ C, $\|C\| = m$	$\gamma n\log_2 n/2+$ + mb_v

Table 3.3: Data structures used by the MOS and MC algorithms.

The MC algorithm employs two arrays, A and $\tilde{B}$. The array A has γn cells. Assuming it uses simple pointers, each cell occupies $\lceil \log_2 n \rceil$ bits. The array $\tilde{B}$ has n cells, and each cell occupies b_f bits. So,

$$\text{SPACE}(\langle S, f_{MOS} \rangle) = n\gamma\lceil \log_2 n \rceil + nb_f.$$

SPACE($\langle S, f_{MOS} \rangle$) and SPACE($\langle S, f_{MC} \rangle$) become equal when

$$\alpha = \frac{1}{\gamma - 1} \tag{3.4}$$

Case 2: NOP for S_f

In the NOP case for S_f, the MOS algorithm employs two arrays—array A and array B. The array A has n/α cells, and each cell occupies $\lceil \log_2 n \rceil$ bits. The array B has n cells, and each cell occupies b_f bits. So,

$$\text{SPACE}(\langle S, f_{MC} \rangle) = \frac{n\lceil \log_2 n \rceil}{\alpha} + nb_f.$$

The MC algorithm employs only one array A, that has γn cells, and each cell occupies b_f bits. So, SPACE($\langle S, f_{MC} \rangle$) is γnb_f. SPACE($\langle S, f_{MOS} \rangle$) and SPACE($\langle S, f_{MC} \rangle$) are equal

when

$$\alpha = \frac{\lceil \log_2 n \rceil}{b_f} \frac{1}{\gamma - 1}. \tag{3.5}$$

Case 3: OP for S_v

In the OP case for S_v, the MOS algorithm employs three arrays—array A, array B, and array C. The array A has n/α cells, and each cell occupies $\lceil \log_2 n \rceil$ bits. The array B has n cells. Assuming it uses simple pointers, each cell occupies $\lceil \log_2 m \rceil$ bits. The array C has m cells, and each cell occupies b_v bits. The space occupied by the set-function pair of the MOS algorithm is

$$\text{SPACE}(\langle S_v, f_{MOS} \rangle) = n \left(\frac{\lceil \log_2 n \rceil}{\alpha} + \lceil \log_2 m \rceil \right) + m b_v.$$

For this case, the MC algorithm employs two arrays, A and C. The array A has γn cells. Assuming it uses simple pointers, each cell occupies $\lceil \log_2 m \rceil$ bits. The array C has m cells, and each cell occupies b_v bits. The space occupied by the set-function pair of the MC algorithm is

$$\text{SPACE}(\langle S_v, f_{MC} \rangle) = n\gamma \lceil \log_2 m \rceil + m b_v.$$

$\text{SPACE}(\langle S_v, f_{MOS} \rangle)$ equals $\text{SPACE}(\langle S_v, f_{MC} \rangle)$ when

$$\alpha = \frac{\lceil \log_2 n \rceil}{\lceil \log_2 m \rceil} \frac{1}{\gamma - 1}. \tag{3.6}$$

Case 4: NOP for S_v

When OP is not required, the words can be rearranged in the array C to decrease the space required by pointers to C. In the MOS algorithm, changing the simple pointer of the array B to a hash pointer, the space occupied by each cell decreases from $\lceil \log_2 m \rceil$ bits to $\lceil \log_2 n \rceil / 2$ bits. So the space requirement decreases to

$$\text{SPACE}(\langle S, f_{MOS} \rangle) = n \left(\frac{\lceil \log_2 n \rceil}{\alpha} + \frac{\lceil \log_2 n \rceil}{2} \right) + m b_v.$$

Similarly, when the simple pointer of the array A is changed to a hash pointer, SPACE($\langle S, f_{MC} \rangle$) decreases to

$$\text{SPACE}(\langle S, f_{MC} \rangle) = \frac{\gamma n \lceil \log_2 n \rceil}{2} + mb_v.$$

SPACE($\langle S, f_{MOS} \rangle$) and SPACE($\langle S, f_{MC} \rangle$) are equal when

$$\alpha = \frac{2}{\gamma - 1}. \tag{3.7}$$

3.3 Experimental Results

In this section, we discuss the performance of the MC algorithm. In total, we conducted three experiments. The first experiment investigates the probability of success of the MC algorithm. The second experiment measures the dependence of μ (Definition 3.1.7) on γ, essentially a time versus space tradeoff. The third experiment compares the performance of the MC and MOS algorithms. Sections 3.3.1, 3.3.2, and 3.3.3 discuss the first, second, and third experiments respectively.

3.3.1 Probability of success of the MC algorithm

The MC algorithm was run on nine dictionaries of sizes varying from 10,000 to 90,000. On each dictionary, γ was varied from 1.20 to 1.25. For each γ, the probability of success was estimated by running the MC algorithm 100 times. The results are tabulated in Table 3.4. As can be seen from the table, there is a very sharp jump in the probability of success as γ varies from 1.22 to 1.23. For $\gamma \leq 1.20$ the algorithm almost always fails, and when $\gamma \geq 1.24$ the algorithm almost always succeeds.

3.3.2 Running time of the MC algorithm

On a dictionary of size 100,000, the MC algorithm was run with γ varying from 1.25 to 1.95 in steps of 0.05. Table 3.5 lists the experimental observation of μ as γ is varied. Figure 3.10 shows a plot of γ versus μ.

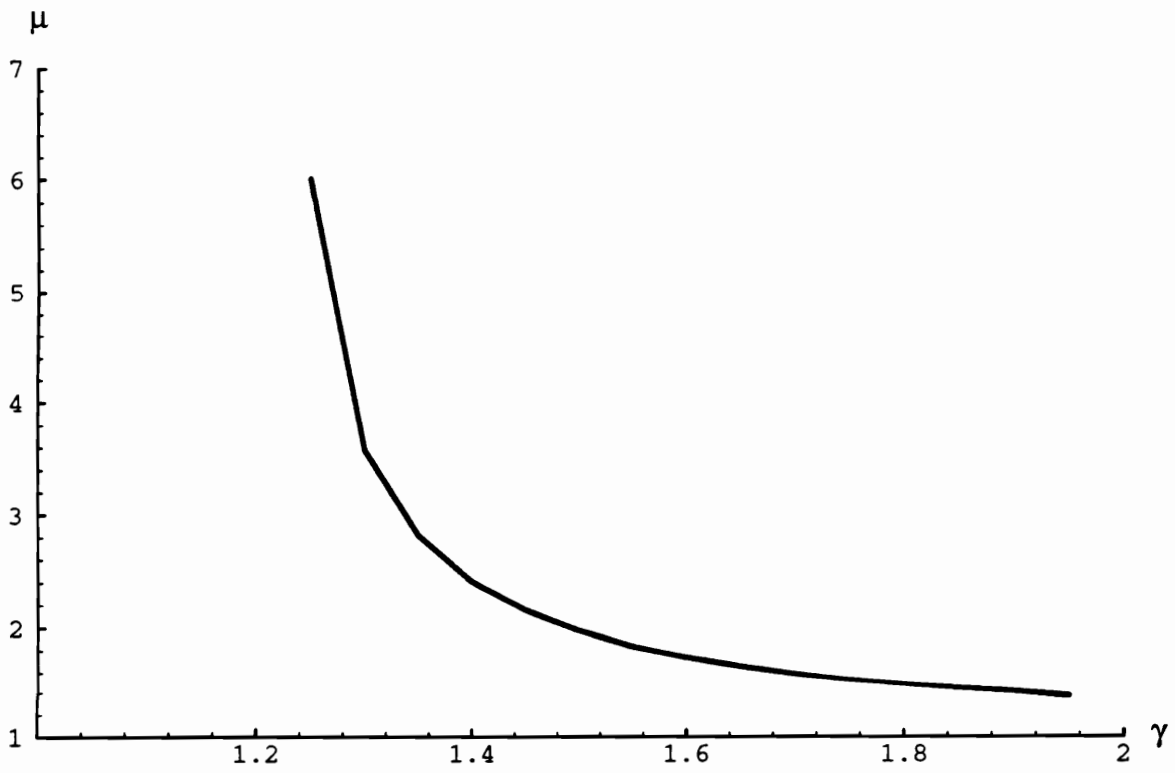


Figure 3.10: A plot of μ vs γ for $1.25 \leq \gamma \leq 1.95$.

γ	Size of the dictionary, n								
	10,000	20,000	30,000	40,000	50,000	60,000	70,000	80,000	90,000
1.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
1.21	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
1.22	0.25	0.26	0.23	0.23	0.18	0.26	0.15	0.14	0.10
1.23	0.88	0.93	0.94	0.98	0.98	1.00	1.00	1.00	1.00
1.24	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1.25	1.00	1.00	0.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 3.4: Probability of success of the MC algorithm.

γ	1.25	1.30	1.35	1.40	1.45	1.50	1.55	1.60
μ	6.01	3.58	2.82	2.41	2.16	1.98	1.84	1.74

γ	1.65	1.70	1.75	1.80	1.85	1.90	1.95
μ	1.66	1.59	1.53	1.48	1.44	1.41	1.37

Table 3.5: Variation of μ with γ for $1.25 \leq \gamma \leq 1.95$.

3.3.3 Running times of the MC and MOS algorithms

In this section, we compare the performances of the MC and MOS algorithms. For the MOS algorithm, we choose the OMOS implementation of Chen [18] in Chapter 2. Both algorithms were run for only the OP case for S_v. For NOP case, we need to rearrange the words and point to the words with the help of a hash pointer. However, the time required for rearranging the words is independent of which perfect hashing algorithm (MC or MOS) is used. So we ignore the time taken for rearranging the words for the purpose of comparing the two algorithms. Similarly, the times taken by the MC and MOS algorithms do not depend on whether the S is a set of fixed-length words or variable-length words. The timings of the MC and MOS algorithms are dependent only on the parameters γ and α. It is only space occupied by the tables (A, B, and C) that depends on the case. So, comparative timings for each case can be found with the help of Equations 3.4, 3.5, 3.6,

	Size of the dictionary	Running time of the MC algorithm in seconds
Dictionary 1	786,432	1906
Dictionary 2	1,200,502	4130

Table 3.6: Running times of the MC algorithm for two real dictionaries on a DECStation 3100.

and 3.7.

Two experiments were conducted to determine the running time of the MC algorithm. The first experiment was run on a DECStation 3100. It consists of applying the MC algorithm to two dictionaries, Dictionary1 and Dictionary2. Dictionary2 was used in [22] and Dictionary1 is a subset of Dictionary2. These times are comparable to those for random dictionaries of the same size, according to other tests on this machine. Table 3.6 shows the running times for these dictionaries.

The second experiment was run a NeXT machine. It consists of applying the MC and MOS algorithms for various values of α and γ. The MC algorithm was applied to 10 dictionaries of sizes varying from 100,000 to 1,000,000. On each dictionary, γ was varied from 1.25 to 1.34. Table 3.9 lists the running times of the MC algorithm. The MOS algorithm was applied to the first seven of the 10 dictionaries. On each dictionary, α was varied from 2.30 to 4.00. As can be seen from this table, when the MOS algorithm was run on dictionaries of size more than 400,000, there was a sharp jump in the time, presumably due to virtual memory thrashing. Table 3.8 lists the running times of the MOS algorithm. The average word length in each dictionary was 20. So m and n satisfy the relation $m = 20n$. We discuss all the cases for our experimental values. In case 1 and case 4, Equations 3.4 and 3.7 are not affected by m and n. In case 2 and case 3, we simplify Equations 3.5 and 3.6 for the purpose of our comparison. Table 3.7 lists the equivalent values of α for γ for all 4 cases.

Case 1: OP for S_f

In this case, the relation between α and γ for equal space is

$$\alpha = \frac{1}{\gamma - 1},$$

and the values of m and n do not affect the relation.

Example 3.3.1 Suppose MC algorithm is run on a dictionary of size 300,000 with $\gamma = 1.29$. Table 3.9 shows that MC algorithm took 111 seconds to run with $\gamma = 1.29$. Table 3.7 shows that if $\alpha = 3.45$ and $\gamma = 1.29$, then $\alpha = 1/\gamma - 1$. The nearest value for $\alpha = 3.45$ is 3.5. From Table 3.8, we see that the MOS algorithm took 239 seconds for a dictionary of size 300,000. For $n = 300,000$, $\log_2 n = 19$. In the MOS algorithm, each cell of the tables A and B occupy 19 bits. So the total space occupied by the table A is $300,000 * 19/3.45 = 1,653,000$ bits. The space occupied by B table is $5,700,000$ bits. The total space occupied by the two tables is $7,353,000$ bits. In the MC algorithm, $\gamma = 1.29$. The space occupied by the table A is $1.29 * 300,000 * 19 = 7,353,000$ bits. The total space occupied by the MOS and MC algorithms came out as equal because α and γ satisfy Equation 3.4, the relationship for equal space by the MOS and MC algorithms.

Case 2: NOP for S_f

In this case, the relation between α and γ for equal space is

$$\alpha = \frac{\lceil \log_2 n \rceil}{b_f} \frac{1}{\gamma - 1}.$$

Assuming b_f is $\lceil \log_2 n \rceil$,

$$\alpha = \frac{1}{\gamma - 1}.$$

This is the same as the relation between α and γ in case 1.

Example 3.3.2 *Consider Example 3.3.1 in case 1. Since α is $1/(\gamma - 1)$ for both case 1 and and this case, Example 3.3.1 applies to this case also.*

Case 3: OP for S_v

In this case, the relation between α and γ for equal space is

$$\alpha = \frac{\lceil \log_2 n \rceil}{\lceil \log_2 m \rceil} \frac{1}{\gamma - 1}.$$

The dictionary size was varied from 100,000 to 1,000,000. On these dictionaries, the minimum value of $\lceil \log_2 n \rceil / \lceil \log_2 m \rceil$ is 17/21. If we take the equivalent value of α to be $0.8/(\gamma - 1)$, SPACE($\langle S_v, f_{MOS} \rangle$) is at least as much as SPACE($\langle S_v, f_{MC} \rangle$).

Example 3.3.3 Again consider Example 3.3.1 in case 1. In this case, α is $0.8/(\gamma - 1)$. Table 3.9 shows that MC algorithm took 111 seconds to run with $\gamma = 1.29$. Table 3.7 shows that if $\alpha = 2.76$ and $\gamma = 1.29$, then $\alpha = 0.8/\gamma - 1$. The nearest value for $\alpha = 2.76$ is 2.8. From Table 3.8, we see that the MOS algorithm took 237 seconds for a dictionary of size 300,000.

Case 4: NOP for S_v

In this case, the relation between α and γ for equal space is

$$\alpha = \frac{2}{\gamma - 1},$$

and the values of m and n do not affect the relation. As γ varies from 1.25 to 1.34, α varies from 5.88 to 8.00 The MOS algorithm failed for $\alpha > 5.00$.

3.4 Discussion

3.4.1 Analytical derivation of the threshold $\gamma = 1.25$

In this chapter we have described the MOS algorithm, provided experimental results and compared these results to the MOS algorithm. As can be seen from Table 3.4, there is a very sharp jump in the probability of success at $\gamma = 1.25$. In future research, it would be very interesting if this result can be analytically predicted. However, our experience suggests an analytical proof is likely to be very tedious. We state two related results here.

γ	α for different cases		
	OP & NOP, S_f $0.8/(\gamma-1)$	OP, S_v $1/(\gamma-1)$	NOP, S_v $2/(\gamma-1)$
1.25	3.20	4.00	8.00
1.26	3.08	3.85	7.69
1.27	2.96	3.70	7.41
1.28	2.86	3.57	7.14
1.29	2.76	3.45	6.90
1.30	2.67	3.33	6.67
1.31	2.58	3.23	6.45
1.32	2.50	3.12	6.25
1.33	2.42	3.03	6.06
1.34	2.35	2.94	5.88

Table 3.7: Equivalent Values of α and γ.

α	Size of the dictionary, n						
	100,000	200,000	300,000	400,000	500,000	600,000	700,000
2.30	78	156	254	473	5270	11647	19080
2.40	77	155	242	546	5211	11285	18863
2.50	77	155	239	483	5510	11358	18645
2.60	77	157	242	472	5159	11330	18920
2.70	76	155	237	476	5167	11266	18627
2.80	77	156	237	475	5159	11299	18618
2.90	76	155	236	490	5192	11230	18549
3.00	77	155	237	476	5176	11248	18508
3.10	77	154	238	476	5193	11483	18763
3.20	77	155	237	465	5189	11191	18474
3.30	77	155	288	474	5188	11159	18523
3.40	77	156	243	472	5045	11166	18417
3.50	78	156	239	476	5175	11200	18676
3.60	78	157	239	466	5135	11181	18415
3.70	78	158	241	486	5177	11181	18439
3.80	79	160	242	464	5052	11397	18349
3.90	79	159	243	484	5074	11120	18268
4.00	80	160	245	472	5071	11124	18540

Table 3.8: Running time of MOS algorithm in seconds.

γ	Size of the dictionary, n				
	100,000	200,000	300,000	400,000	500,000
1.25	38	76	116	156	199
1.26	36	74	114	155	195
1.27	36	74	113	153	192
1.28	36	73	111	151	190
1.29	35	72	111	150	189
1.30	35	72	110	149	188
1.31	36	71	110	148	187
1.32	35	72	109	148	186
1.33	35	71	108	147	186
1.34	34	71	108	147	185

γ	600,000	700,000	800,000	900,000	1,000,000
1.25	246	307	941	1460	1655
1.26	238	300	867	1325	1459
1.27	240	297	804	1181	1340
1.28	243	294	735	1110	1258
1.29	235	302	715	1045	1184
1.30	241	300	683	998	1129
1.31	232	305	661	966	1085
1.32	237	307	640	926	1043
1.33	228	336	635	898	1016
1.34	227	317	603	870	983

Table 3.9: Running time of the MC algorithm in seconds.

The first one is on the the existence of a cycle in a random graph. The second one is on the existence of a cycle in a 3-regular hypergraph.

Theorem 3.4.1 *Let G be a random graph with N vertices and n edges. If $N/n > 2$, the probability that there exists a cycle in G is $o(1)$. If $N/n < 2$, the probability that there exists no cycle in G is $o(1)$.*

Proof: Refer to Chapter 5 in [2]. □

Theorem 3.4.2 *Let G be a 3-regular random graph with N vertices and n edges. If $N/n > 9.67$, the probability that there exists a cycle (subhypergraph with minimum degree 2) in G is $o(1)$.*

Proof: Refer to Theorem 4.9 in [31]. □

Theorem 3.4.1 provides a threshold for the existence of a cycle in random graph. The proof of Theorem 3.4.1 is very tedious and there is no straightforward way to extend it to 3-regular hypegraphs. Theorem 3.4.2 provides only an upper bound on the existence of a cycle in a 3-regular random hypergraph. There is a large gap between the experimentally observed threshold $\gamma = 1.25$ and the upper bound given by Theorem 3.4.2[31]. However, since the proof of Theorem 3.4.1 is very tedious, tightening the bound provided by Theorem 3.4.2 is likely to be very difficult too.

3.4.2 Comparison of MC and MOS algorithms

The MC algorithm succeeds whenever γ is at least 1.25. The size of PHF found by the MC algorithm depends upon the cases as described in Section 3.2. The main advantage of the MC algorithm over the MOS algorithm is that it runs faster than the MOS algorithm whenever the MC algorithm can find a PHF of the same size as the one found by the MOS algorithm. Its disadvantages are:

- For a find operation, it requires three accesses to the A table as opposed to one access by the MOS algorithm.

- The MC algorithm fails to produce a PHF before approaching Mehlhorn's lower bound, whereas the MOS algorithm takes longer and longer time as it approaches the Mehlhorn's lower bound. Consequently, the smallest PHF that can be found by the MC algorithm is larger than the smallest PHF that can be found by the MOS algorithm given a large amount of time.

Chapter 4

GRAPH ORDERING

The MC algorithm in Chapter 3 suggests a problem of ordering the vertices of a 3-regular hypergraph so as to meet certain constraints. In this chapter, we discuss the equivalent problem for an ordinary undirected graph. In the MC algorithm, if we map every word $x \in S$ to a 2-tuple $\{h_1(x), h_2(x)\}$ instead of a 3-tuple $\{h_1(x), h_2(x), h_3(x)\}$, we obtain an ordinary graph G, and the ordering problem is equivalent to the following graph problem. Given a graph G, can we lay out the vertices of G on a straight line such that from every vertex at most one edge goes left? It is easy to see that this layout is possible if and only if the graph G is acyclic, i.e., a forest.

Another motivation for this problem comes from Fox et al.'s [21] algorithm. They seek a PHF of the form in Equation 1.1. Their algorithm builds a graph from Equation 1.1, where each edge of the graph corresponds to a word in the dictionary. Then it orders the vertices of the graph and processes each vertex in that order. At each vertex it randomly tosses the words corresponding to the edges that go left from the vertex into an array of words. The random tossing of the words can be avoided if we can order the vertices of the graph such that at most one edge goes left from every vertex [16].

In this chapter, we consider two generalizations of this problem.

1. Is it possible to lay out the vertices of G on a straight line such that at most r edges go left from every vertex of G?

2. Is it possible to lay out the vertices of G on a straight line such that at most r edges go left and at most s edges go right from every vertex of G?

The first problem has a polynomial-time solution, while the second problem is NP-

complete. The remainder of this chapter is organized as follows. Section 4.1 precisely defines terminology. Section 4.2 presents a polynomial-time solution for the first problem. Section 4.3 presents a proof that the second problem is NP-complete. Section 4.4 describes an equivalence problem for digraphs.

4.1 Definitions and Terminology

Definition 4.1.1 *An* undirected graph $G = (V, E)$ *consists of a set of* vertices V *and a set of* edges E. *An edge of an undirected graph is an unordered pair* $\{u, v\}$ *where* $u, v \in V$. *Edge* $\{u, v\} \in E$ *is said to be* incident *on the vertices* u *and* v.

Definition 4.1.2 *If* $|V| = N$, *an* ordering $\xi : V \rightarrow I_N$ *is a one-to-one mapping of the vertices of* V *onto* I_N.

We also refer to the ordering of the vertices of a graph as a *layout* of the graph. Vertex u is to the *left* of v under the layout ξ if $\xi(u) < \xi(v)$, and vertex u is to the right of v if $\xi(u) > \xi(v)$.

Definition 4.1.3 *An edge* $\{u, v\} \in E$ *is* backward *with respect to* v *and* forward *with respect to* u *if* $\xi(u) > \xi(v)$.

Definition 4.1.4 *The partition of the set of incident edges incident on a vertex* v *into backward and forward edges is the* split *of the vertex* v.

Definition 4.1.5 *With respect to an ordering* ξ, *the* backward degree *of* v, $b(\xi, v)$, *is the number of backward edges incident on* v. *The* forward degree *of* v, $\phi(\xi, v)$, *is the number of forward edges incident on* v. *The* degree *of a vertex* v *is the sum of the backward and forward degrees of* v.

Definition 4.1.6 *The maximum backward degree with respect to* ξ *in the graph* G *is*

$$\Delta b(\xi, G) = \max_{1 \leq i \leq N} b(\xi, v_i).$$

The maximum forward degree with respect to ξ in the graph G is

$$\Delta\phi(\xi, G) = \max_{1 \le i \le N} \phi(\xi, v_i).$$

The maximum degree of all the vertices in G is

$$\Delta(G) = \max_{1 \le i \le N} \{\text{degree}(v_i)\}.$$

The minimum degree of all the vertices in G is

$$\delta(G) = \min_{1 \le i \le N} \{\text{degree}(v_i)\}.$$

Definition 4.1.7 *A* directed graph *$\vec{G}$ is a 2-tuple $(V, \vec{E})$, where v is a set of* vertices *and $\vec{E}$ is a set of* edges. *An edge of a directed graph is an ordered pair $\langle u, v \rangle$ where $u, v \in V$.*

The edge $\langle u, v \rangle$ is said to be directed from u to v.

Definition 4.1.8 *The* indegree *of a vertex $v \in \vec{E}$ is the number of edges $\langle u, v \rangle \in \vec{G}$. The* outdegree *of a vertex $u \in \vec{G}$ is the number of edges $\langle u, v \rangle \in \vec{E}$.*

4.2 Minimum Backward Edge Ordering

Given a graph G and an integer k, the minimum backward edge ordering problem for G and k is to find an ordering ξ such that $\Delta b(\xi, G) \le k$. Figure 4.1 presents an algorithm to solve this problem. It consists of deleting vertices of degree at most k from G until G is empty. The deletion of vertex v from G consists of removing all the edges incident on v from G also. This algorithm takes $O(|E| + |V| \log |V|)$ time in the worst case.

Theorem 4.2.1 *Let $\mathcal{G}_1$ be the set of graphs G for which the minimum degree of every subgraph is at most k. Let $\mathcal{G}_2$ be the set of graphs G that have an ordering in which every vertex of the graph has backward degree at most k. Then, $\mathcal{G}_1 = \mathcal{G}_2$.*

Proof: We first show that $\mathcal{G}_1 \subseteq \mathcal{G}_2$. Suppose $G \in \mathcal{G}_1$. Apply the procedure *OrderVertices* in Figure 4.1 to G. Since every subgraph of G has at least one vertex of degree at most k, it always returns an ordering ξ. Hence $G \in \mathcal{G}_2$, and $\mathcal{G}_1 \subseteq \mathcal{G}_2$.

```
OrderVertices(G, k)

{
        i = N;
        while (i ≥ 1) {
                if (there exists a vertex v of degree ≤ k) {
                        ξ(v) = i;
                        Delete(v, G);
                        i = i − 1;
                }
                else
                        return(FALSE);
        }
        report(ξ);
        return(TRUE);
}
```

Figure 4.1: A procedure for finding a ξ such that $\Delta b(\xi, G) \leq k$.

It remains to show that $\mathcal{G}_2 \subseteq \mathcal{G}_1$. Suppose $G \in \mathcal{G}_2$. Let ξ be an ordering such that $\Delta b(\xi, G) \leq k$. Let H be any subgraph of G. Let v be the vertex of H with greatest $\xi(v)$. Obviously, the degree of this vertex in H is at most k. So, every subgraph has at least one vertex of degree at most k. This implies $G \in \mathcal{G}_1$, and hence $\mathcal{G}_2 \subseteq \mathcal{G}_2$.

We conclude that $\mathcal{G}_1 = \mathcal{G}_2$.

□

With a slight modification of the above algorithm, we can find an ordering ξ which minimizes the maximum backward degree, $\Delta b(\xi, G)$. Instead of selecting an arbitrary vertex of degree at most k as in the *if* statement in Figure 4.1, we greedily select a vertex of minimum degree every time.

4.3 Bounded Split Problem

In this section, we consider a more general ordering problem than the minimum backward edge ordering problem. Instead of restricting attention only to backward edges, we consider the problem of ordering the vertices such that both the number of forward and backward edges is bounded. The decision problem we consider is the following.

BOUNDED SPLIT PROBLEM(BSP):

INSTANCE: A graph $G = (V, E)$ and two integers r and s.

QUESTION: Is there an ordering ξ of its vertices such that for all $v \in V$, $b(\xi, v) \leq r$ and $\phi(\xi, v) \leq s$?

This problem is solvable in polynomial time if either r or s is equal to one.

Theorem 4.3.1 *The BSP has a linear-time solution for* $r = 1$.

Proof: It is easy to see that when we are restricting either the backward degree or forward degree to one, the only class of graphs that satisfies this restriction is the class of acyclic graphs. An algorithm for the problem is as follows. Assume that $r = 1$. Use depth-first

search to check the graph G for the existence of a cycle. If G is acyclic, check $\Delta(G)$ is greater than $r+1$. If so, answer *no*, otherwise answer *yes*. □

However, BSP is NP-complete in general. We prove this for the case $r = 2$ and $s = 2$. The reduction is from another NP-complete problem NOT ALL EQUAL 3-SAT [36].

Definition 4.3.1 *A boolean variable y has two* literals *y and $\bar{y}$ associated with it.*

NOT ALL EQUAL 3-SAT(NAE3-SAT):

INSTANCE: A set of m clauses $C_1, C_2, \ldots, C_M$ each consisting of three literals chosen from N boolean variables $y_1, y_2, \ldots, y_N$.

QUESTION: Is there a truth assignment for the variables $y_1, y_2, \ldots, y_N$ such that every clause has at least one literal with a value 1 and at least one literal with a value 0.

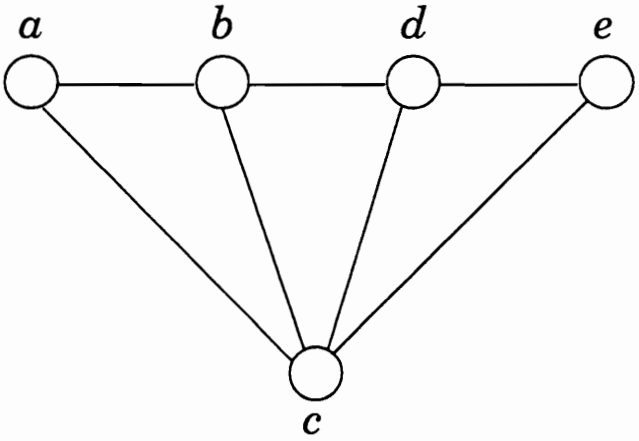
Theorem 4.3.2 *BSP is NP-complete for $r = 2$ and $s = 2$.*

Proof: We prove the theorem by constructing a graph G that satisfies the following conditions.

- An instance of NAE3-SAT can be transformed to a corresponding instance of BSP in polynomial time.
- The instance of NAE3-SAT has a solution if and only if the corresponding instance of BSP has a solution.

Construction of G

Consider the graph H in Figure 4.2. The only two possibilities for its layout that satisfy the constraints $\Delta b(\xi, H) \leq 2$ and $\Delta\phi(\xi, H) \leq 2$ are *abcde* and *edcba*. The graph G contains copies of H as subgraphs. A short form for the graph H is depicted below H in Figure 4.2. The vertex c is not included in the short form, because G contains no edges incident on c other than a, b, d or e. The vertices a, b, d, and e are always depicted on the upper left, lower left, lower right, and upper right corners of the square respectively. Now consider a



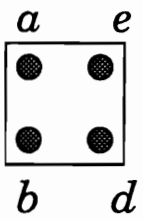


Figure 4.2: The graph H and a short form representation.

Figure 4.3: A cascade of k copies of H: $H_1, H_2, \ldots, H_k$.

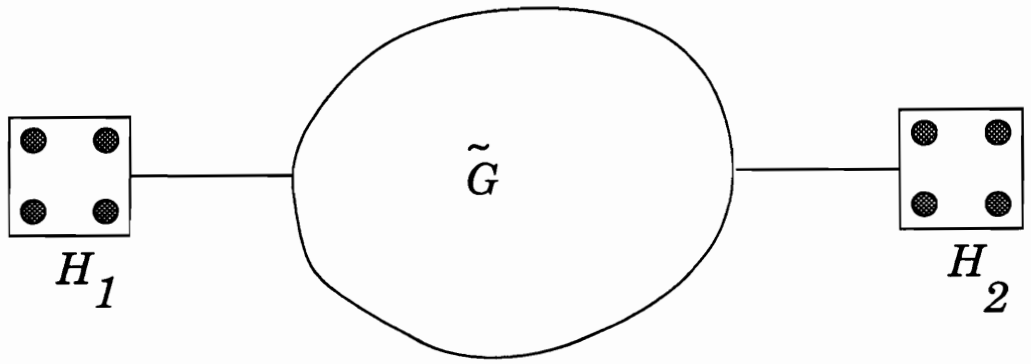


Figure 4.4: An arbitrary graph $\tilde{G}$ with an H on each side.

cascade of k copies of $H_1, H_2, \ldots, H_k$ connected as in Figure 4.3. For $1 \leq i \leq k$, name the vertices of H_i as a_i, b_i, c_i, d_i and e_i respectively. The only two possibilities for its layout are: $a_1b_1c_1d_1e_1a_2b_2c_2d_2e_2 \ldots a_kb_kc_kd_ke_k$ and $e_kd_kc_kb_ka_k \ldots e_2d_2c_2b_2a_2e_1d_1c_1b_1a_1$.

Now consider a graph as in Figure 4.4. In that figure, an arbitrary graph $\tilde{G}$ is connected to one of a_1, b_1, c_1, or d_1 by a single edge and to one of a_2, b_2, c_2, or d_2 by a single edge. The only two possible patterns for the layout of $\tilde{G}$ are

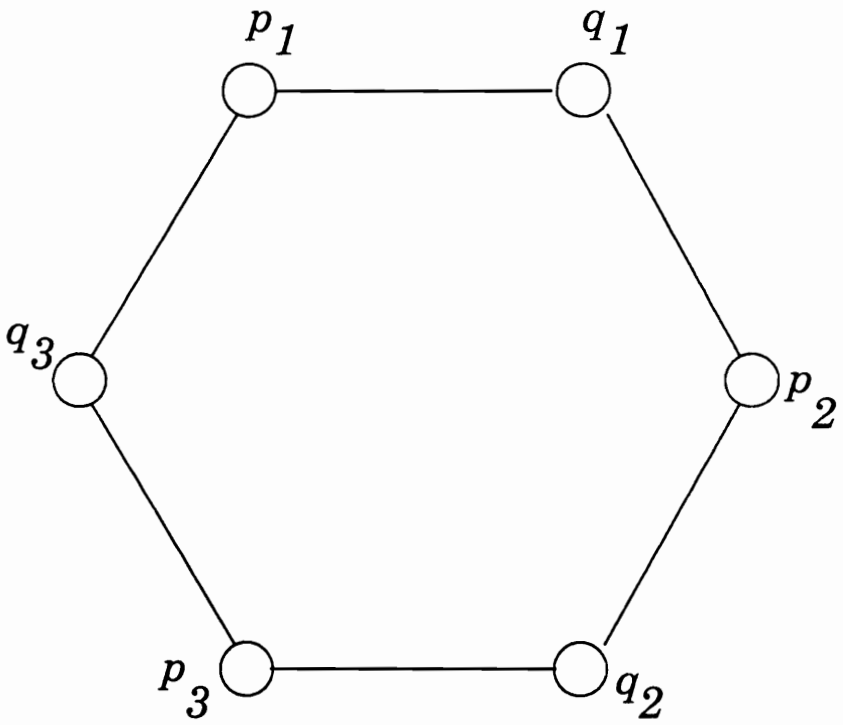
$a_1b_1c_1d_1e_1\{\text{vertices of } \tilde{G}\}a_2b_2c_2d_2e_2$ and $e_2d_2c_2b_2a_2\{\text{vertices of } \tilde{G}\}e_1d_1c_1b_1a_1$.

The main idea of the construction is this. The $\tilde{G}$ in the center is a gadget that stands for a clause. H_1 and H_2 stand for the literals y and $\bar{y}$ of a particular boolean variable. One layout corresponds to setting the truth value of the variable to 1, and the other layout corresponds to setting it to 0.

Suppose a variable y occurs in k clauses $C_{i_1}, C_{i_2}, \ldots, C_{i_k}$. We represent this variable by a cascade of $2k$ copies of H. Denote the first k copies by $H_{i_1}, H_{i_2}, \ldots, H_{i_k}$ and the last k copies by $\bar{H}_{i_k}, \bar{H}_{i_{k-1}}, \ldots, \bar{H}_{i_1}$. The layout

$$H_{i_1}, H_{i_2}, \ldots, H_{i_k}\bar{H}_{i_k}, \bar{H}_{i_{k-1}}, \ldots, \bar{H}_{i_1}$$

corresponds to a value of 1 for y and the reverse layout corresponds to a value of 0 for y.



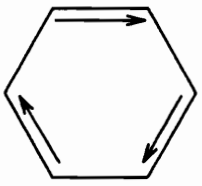


Figure 4.5: The gadget corresponding to a clause.

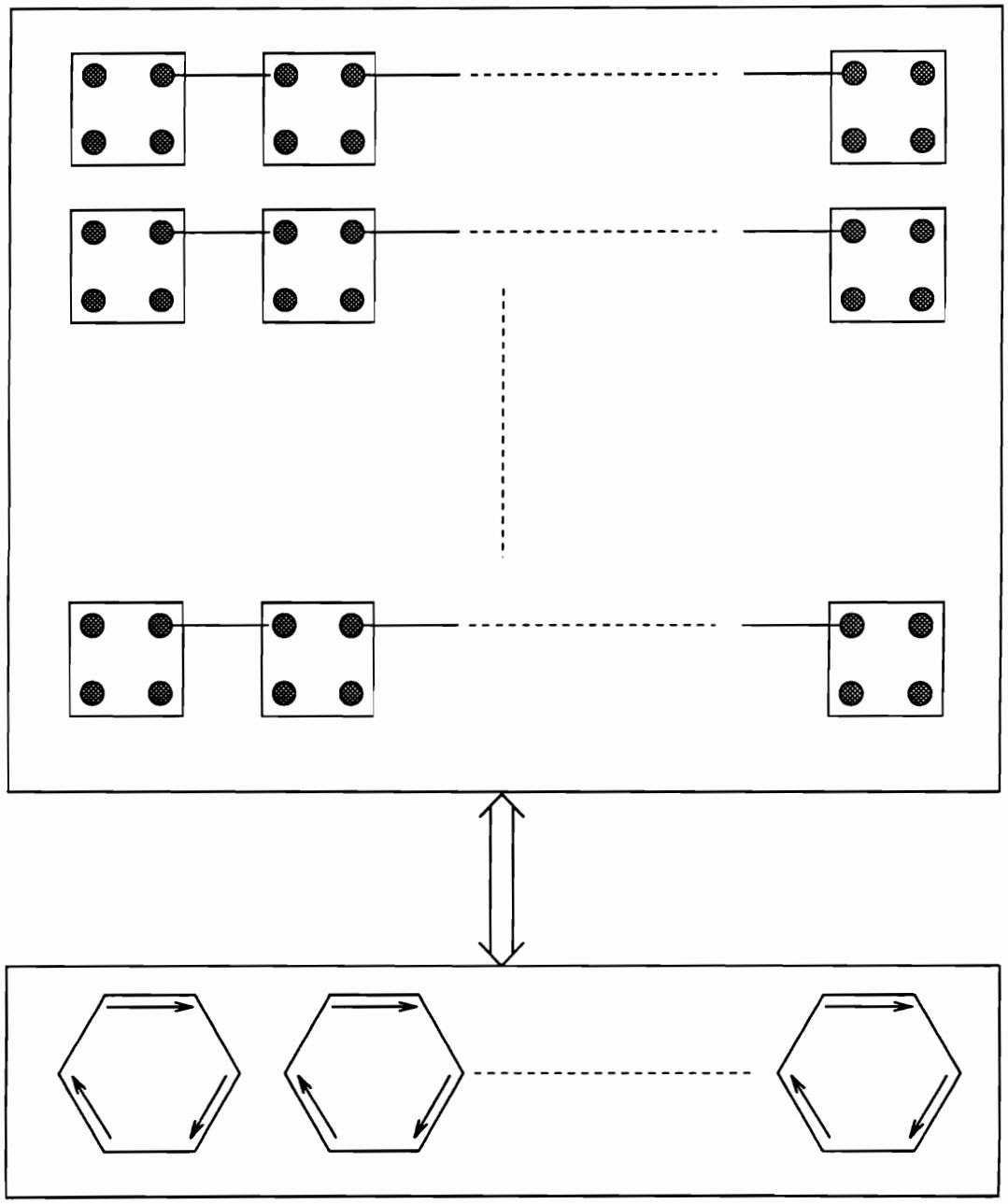


Figure 4.6: A simplified representation of the entire construction.

Figure 4.5 illustrates the gadget corresponding to a clause, a hexagon with six vertices. These vertices are viewed as three pairs—$\langle p_1, q_1 \rangle$, $\langle p_2, q_2 \rangle$, and $\langle p_3, q_3 \rangle$. Each pair corresponds to one of the three literals in the clause. A schematic of the graph G appears in Figure 4.6. Each hexagon in the graph connects to exactly three cascades. Consider a clause C_i for some i, $1 \leq i \leq M$. Suppose the pair $\langle p_1, q_1 \rangle$ corresponds to literal y. p_1 is connected to the e vertex of H_i and the a vertex of $\bar{H}_i$, and q_1 is connected to the d vertex of H_i. Instead, if the pair $\langle p_1, q_1 \rangle$ corresponds to a literal $\bar{y}$, p_1 is connected to the same e vertex of H_i and the a vertex of $\bar{H}_i$, but q_1 is connected to the b vertex of $\bar{H}_i$. Hence, each p vertex has degree 4 and each q vertex has degree 3. The hexagon corresponding to the clause C_i can be laid out if and only if the following conditions are satisfied.

- The leftmost and the rightmost vertices of the hexagon are q vertices.
- The backward degree of the leftmost q vertex is one and the forward degree of the rightmost q vertex is one.

An example illustrating all these constructions together, for the boolean expression $y_1\bar{y}_2y_3 + y_1\bar{y}_2\bar{y}_3$, is given in Figure 4.7.

If part

Suppose G has a layout ξ such that $\Delta\phi(\xi, G) \leq 2$ and $\Delta b(\xi, G) \leq 2$. As we observed in our construction, the vertices of all the H's corresponding to a cascade can appear only in one of the two orders. Assign a truth value of 1 to all the literals that fall on the left, and assign a truth value of 0 to the ones that fall on the right. By the construction, each hexagon can fall only in the middle H and $\bar{H}$ graphs it is connected to. So, the leftmost and the rightmost vertices of the hexagon cannot be p vertices and therefore must be q vertices. Since the forward degree of the leftmost q vertex cannot exceed two, its backward degree must be one. Similarly, the forward degree of the rightmost q vertex is one. So every clause has at least one literal that has a value 1 and one literal that has a value 0.

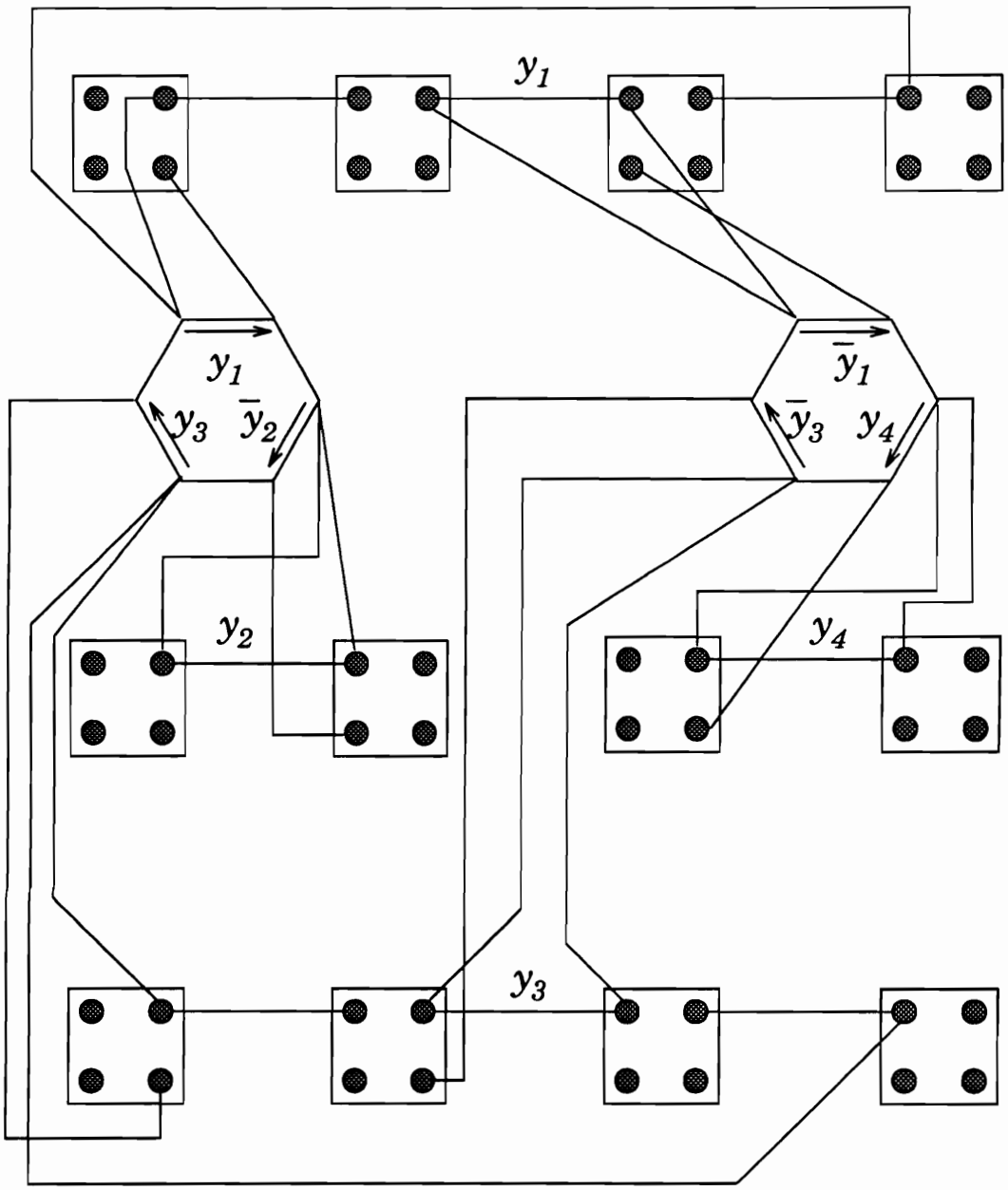


Figure 4.7: The graph for the boolean expression $y_1\bar{y}_2y_3 + y_1\bar{y}_3\bar{y}_4$.

Only if part

Suppose the instance of NAE3-SAT has a solution, i.e., a truth assignment such that every clause has a literal of value 1 and a literal of value 2. Lay out the vertices of the graph G in the following manner. Recall from the construction that each variable y corresponds to a cascade of H and $\bar{H}$ graphs. If y has a value of 1, lay out the vertices of all the H graphs on the left and the vertices of all the $\bar{H}$ graphs on the right. If y has a value of 0, lay out the vertices of all the $\bar{H}$ graphs on the left and the vertices of all the H graphs on the right. Lay out all the vertices corresponding to the clauses in the middle. Lay out the graph G corresponding to a clause in the following manner. For every clause, there is at least one literal that has a value 1 and one literal that has a value 0. Pick two such literals from every clause. Put the q vertex corresponding to the literal with the value 1 on the left and the q vertex corresponding to the literal with the value 0 on the right. Since all the hexagons are in the middle, with H and $\bar{H}$ graphs on the left and the right, the backward and forward degrees of all the vertices of H and $\bar{H}$ graphs are less than or equal to 2. Consider the vertices corresponding to a hexagon. The leftmost and the rightmost vertices of the hexagon are q vertices, and all the p vertices of the hexagons are in the middle. Since the hexagon is in the middle of an H and $\bar{H}$ graph, the forward and backward degree of each p vertex is exactly two. Since one edge of the middle q vertex goes left and one edge goes right, neither the forward degree nor the backward degree of this vertex exceeds two. According to the layout, two edges from the leftmost q go right and one edge goes left. Similarly, two edges from the rightmost q vertex go left and one edge goes right. So, neither the forward degree nor the backward degree of these vertices exceeds two.

Transformation in polynomial time

For every variable y that is present in a clause, we used 12 vertices—two vertices for p and q, and five vertices for each literal y and $\overline{y}$. Clearly, the transformation can be accomplished in polynomial-time. Also, a nondeterministic Turing machine can guess a

permutation ξ and check the $b(\xi, v)$ and $\phi(\xi, v)$ for every vertex in polynomial-time. So BSP is in NP. We conclude that BSP is NP-complete.

□

4.4 An Equivalent Digraph Problem

The *minimum backward edge problem* and the BSP can be viewed from another interesting perspective. Suppose we have an ordering ξ of an undirected graph $G = (V, E)$. We can convert it to a directed graph $\vec{G}$ in the following manner. If $(u, v) \in E$ and $\xi(u) > \xi(v)$, direct it from v to u. Obviously $\vec{G}$ will have no cycles. Also given a digraph $\vec{G}$ with no cycles, we can always order the vertices such that no edge points left. In light of this observation, the above problems can be rephrased as follows.

Bounded outdegree DAG:

INSTANCE: An undirected graph G, and an integer k.

QUESTION: Can we direct the edges of G such that the resulting digraph $\vec{G}$ is acyclic and the indegree of every vertex $v \in \vec{G}$ is less than k?

Theorem 4.4.1 *Bounded outdegree DAG question can be answered in polynomial-time.*

Proof: Similar to Theorem 4.2.1. □

Bounded split DAG:

INSTANCE: An undirected graph G, and two integers r and s.

QUESTION: Can we direct the edges of the graph G such that the resulting graph $\vec{G}$ is acyclic and the outdegree of every vertex of $\vec{G}$ is at most r and the indegree of every vertex of $\vec{G}$ is at most s?

Theorem 4.4.2 *Bounded split DAG question can be answered in polynomial-time.*

Proof: Similar to Theorem 4.3.2. □

Chapter 5

DYNAMIC PERFECT HASHING

In Chapter 2 and Chapter 3, we discussed perfect hashing algorithms for static dictionaries. In this chapter, we give an algorithm called the *dynamic perfect hashing (DPH)* algorithm that applies to dynamic dictionaries. While a static dictionary can support only the *find* operation, a dynamic dictionary can support *find, insert,* and *delete* operations.

Just like the MOS or MC algorithm, the DPH algorithm maintains an array A of size $O(n)$, where n is the number of words in the dictionary. One important factor that affects its performance is the dynamic nature of n itself. How does n vary with time? Does it oscillate about a fixed value n_0? Is there a probability distribution on n? Should the size of the array A vary as n varies? Depending on whether we vary the size of the array A or not, we divide the DPH algorithm into two cases.

In the first case, the size of A is fixed, and we model the set S as a birth and death process [27]. We analyze the performance of the DPH algorithm when the dictionary is either an $M/M/1$ queue or an $M/M/\infty$ queue. This case is appropriate when we know the nature of the dictionary. We can exploit this knowledge by keeping the size of A fixed and avoid the penalty of adjusting A. For this case, the cost of insertion or deletion depends on the kind of queue. It is $O(1)$ for both $M/M/1$ and $M/M/\infty$ queues that we analyze in this chapter. However, it is not guaranteed to be $O(1)$ for any birth and death process.

In the second case, as n varies, the size of A varies. The second case has wider applicability. However, for its generality, we pay the penalty of readjusting the array A . For the second case, the expected cost of insertion, deletion, and find is always $O(1)$.

Aho and Lee [1] present a dynamic perfect hashing algorithm with the size of A fixed. However, their algorithm has an upper bound on n. In our algorithm, the performance

degrades gracefully as the number of elements is increased. Dietzfelbinger et al. [12] give an algorithm in which the size of A varies as n varies. There is no upper bound on n, and the total space occupied by the data structures of the algorithm is always proportional to n. However, their algorithm works only for sets of fixed-length words, and there is no obvious way to extend it to sets of variable-length words, while keeping their space bounds. In this chapter, we give an algorithm which works for sets of variable-length words.

The remainder of this chapter is organized as follows. Section 5.1 gives a description of the dynamic perfect hashing algorithm. Section 5.2 gives a general formula for insertion and deletion in the DPH algorithm. Section 5.3 models S as a birth and death process[27], and analyses the performance of the DPH algorithm. Section 5.4 analyses the performance of the DPH algorithm when the array A grows or shrinks in accordance with the size of the dictionary.

5.1 DPH Algorithm

We describe the DPH algorithm only for fixed-length words. It can be extended to variable-length words by storing the words in a character array C and storing pointers to the pointers to words in place of the original words. The DPH algorithm employs the following data structures:

- An array A of size N;
- For $i > 0$, M_i memory blocks of size i.

A memory block of size i is a contiguous array of i cells. Each cell can hold exactly one word in S. We refer to the contents of a particular memory block by the operator []. For example, $b[j]$ refers to the word stored in the jth cell of memory block b. Each cell can hold exactly one word of S. At a particular instant, a block is either free or allocated. All free blocks are maintained in linked lists, with one list for each block size. We assume that we have two routines *GetBlock(j)*, which allocates a previously free memory block of size j, and *ReleaseBlock(b)*, which frees an allocated memory block b.

Each cell of the array A consists of three fields: *blockptr, param, blocksize.* The address of a word x, if it exists in the dictionary, can be computed by the following formula:

$$\begin{aligned} j &= A[h_1(x)].blocksize \\ Address(x) &= A[h_1(x)].blockptr + h_2^{(j)}(x, A[h_1(x)].param). \end{aligned}$$

where h_1 is a random function which maps U to I_N. $h_2^{(j)}(x,i)$ is a random function that maps $U \times I$ to I_j.

Figure 5.1 gives a procedure *find*, which can answer the question whether a given word x exists in the dictionary. It checks the content addressed as above. It answers TRUE if it matches x, otherwise it answers FALSE.

Figure 5.1 gives a procedure *insert* to insert a word x into the dictionary. The procedure consists of three steps. The first step maps x to the cell $A[h_1(x)]$. Let b denote the memory block pointed to by $A[h_1(x)].blockptr$, and let the size of b be j. Let $\mathcal{S}$ denote all the words in the block b. The second step consists of releasing the block b and getting another block b' of size $j+1$. The third step consists of finding an i such that $h_2^{(j+1)}(y,i)$ is a PHF for all of $\mathcal{S} \cup \{x\}$. The program for deletion is similar to insertion except that we get a block of size $j-1$ instead of $j+1$, and we find a PHF for $\mathcal{S} - \{x\}$.

5.2 Insertion and Deletion Time

When there are n words in the dictionary, let R_n and D_n be the expected costs of insertion and deletion respectively. The first step in inserting or deleting a word x is to map it into the bin $h_1(x)$ of array A. The probability that h_1 maps x to a bin of size j is given by

$$\binom{n}{j}\left(1-\frac{1}{N}\right)^{n-j}\frac{1}{N^j}.$$

The probability that a random function is a PHF for a set of size j is $j!/j^j \approx e^{-j}$. So, the expected cost to find a PHF for a bin of size j is e^j. Since the bin size increases from j

Find(x)

begin

/* Map x to the bin $h_1(x)$ */

$k = h_1(x)$
$b = A[k].blockptr$
$i = A[k].param$
$j = A[k].blocksize$

/* Compare the content of this address with x */

if $(b[h_2^{(j)}(x,i)] == x)$
return (TRUE)
else
return (FALSE)
endif
end

Figure 5.1: The *find* program for dynamic perfect hashing.

Insert(x)

begin
/* Map x to the bin $h_1(x)$ */

$k = h_1(x)$
$b = A[k].blockptr$
$j = A[k].blocksize$

[Continued on next page]

```
        /*
        Copy all the words in the memory blocks into set S.
        Note that *(b + l) stands for the content of the address b + l
        */

        S = {x}
        for  l = 0 to j − 1
                S = S ∪ {b[l]}
        endfor

        /* Release the block and get a new block of size j + 1 */

        ReleaseBlock(b)
        b = GetBlock(j + 1)
        A[k].blockptr = b
        A[k].blocksize = j + 1

        /*
        Randomly toss all the words in S into the new block till they land in empty cells.
        The routine permutation checks if its input is a permutation
        */

        i = 0
        loop
                if (permutation(h_2^(j+1)(S, i))) exit out of the loop
                i = i + 1
        forever

        /* Copy the words in S into the new memory block */

        for  all y ∈ S
                b[h_2^(j+1)(y, i)] = y
        endfor
        A[k].param = i
end
```

Figure 5.2: The *insert* program for dynamic perfect hashing.

Delete(x)

begin

/* Map x to the bin $h_1(x)$ */

$k = h_1(x)$
$b = A[k].blockptr$
$j = A[k].blocksize$

/*
Copy all the words in the memory blocks into set $\mathcal{S}$.
Note that $*(b + l)$ stands for the content of the address $b + l$
*/

for $l = 0$ to $j - 1$
    $\mathcal{S} = \mathcal{S} \cup \{b[l]\}$
endfor
$\mathcal{S} = \mathcal{S} - \{x\}$

/* Release the block and get a new block of size $j + 1$ */

ReleaseBlock(b)
b = GetBlock($j - 1$)
$A[k].blockptr = b$
$A[k].blocksize = j - 1$

/*
Randomly toss all the words in $\mathcal{S}$ into the new block till they land in empty cells.
The routine *permutation* checks if its input is a permutation
*/

$i = 0$
loop
    if ($permutation(h_2^{j-1})(\mathcal{S}, i)$)) exit out of the loop
    $i = i + 1$
forever

[Continued on next page]

```
        /* Copy the words in S into the new memory block */

        for all y ∈ S
                b[h_2^(j-1)(y, i)] = y
        endfor
        A[k].param = i
end
```

Figure 5.3: The *delete* program for dynamic perfect hashing.

to $j+1$, the cost is e^{j+1}.

$$\begin{aligned} R_n &= \sum_{j=0}^{n} \binom{n}{j} \left(1-\frac{1}{N}\right)^{n-j} \frac{1}{N^j} e^{j+1} \\ &= e\left(1+\frac{e-1}{N}\right)^n. \end{aligned}$$

Let $\sigma = 1+(e-1)/N$, so that $R_n = e\sigma^n$. In the case of deletion, we assume that any word in the dictionary is equally likely to be deleted. For $1 \le j \le n$, let N_j denote the number of blocks of size j that are allocated. So, if there are n words in the dictionary, and the next operation is a deletion, the probability that h_1 maps x to a bin of size j is given by

$$\begin{aligned} \Pr[\text{Size of bin } h_1(x) = j] &= \frac{jN_j}{n} \\ &= \frac{jN}{n}\frac{N_j}{N} \\ &= \frac{jN}{n}\binom{n}{j}\left(1-\frac{1}{N}\right)^{n-j}\frac{1}{N^j} \\ &= \binom{n-1}{j-1}\left(1-\frac{1}{N}\right)^{n-j}\frac{1}{N^{j-1}}. \end{aligned}$$

Since the bin size decreases from j to $j-1$, the cost is e^{j-1}.

$$\begin{aligned} D_n &= \sum_{j=1}^{n} \binom{n-1}{j-1} \left(1-\frac{1}{N}\right)^{n-j} \frac{1}{N^{j-1}} e^{j-1} \\ &= \left(1+\frac{e-1}{N}\right)^{n-1} \\ &= \sigma^{n-1}. \end{aligned}$$

We summarize the results of this section in Theorem 5.2.1.

Theorem 5.2.1 *Suppose at an instant the size of the table A employed by the DPH algorithm is N and the number of elements in the dictionary is n. Then the expected cost of insertion R_n is*

$$R_n = e\sigma^n,$$

and the expected cost of deletion D_n is

$$D_n = \sigma^{n-1},$$

where

$$\sigma = 1+\frac{e-1}{N}.$$

5.3 Dynamic Perfect Hashing with Fixed Array Size

In this section, we analyze the performance of the DPH algorithm when the size of array A is a fixed integer N. We model the dynamic dictionary S as a birth and death process [27]. We assume that the arrival of words forms a Poisson process. The process has states E_i, $0 \leq i \leq \infty$, where E_i represents a dictionary containing i words. When the system is in state E_k, the arrival rate is λ_k and the departure rate is μ_k. p_n denotes the steady state probability that the system is in E_n. The expected cost of the next insertion or deletion is given by

$$\lambda_n R_n + \mu_n D_n.$$

The expected cost over all the n is given by

$$C = \sum_{n=0}^{\infty} p_n(\lambda_n R_n + \mu_n D_n). \tag{5.1}$$

We analyze the performance of our algorithm for $M/M/1$ and $M/M/\infty$ queues.

Classical queuing system: $M/M/1$

For a classical $M/M/1$ queuing system [27], the arrival rate is a constant λ, and the departure rate is a constant μ for every state other than E_0. The steady state probability p_n of being in state E_n is $(1-\rho)\rho^n$, where $\rho = \lambda/\mu$. Substituting these values in Equation 5.1, the expected cost of insertion or deletion is

$$\begin{aligned} C &= \sum_{n=0}^{\infty} p_n(\lambda_n R_n + \mu_n D_n) \\ &= \sum_{n=0}^{\infty}(1-\rho)\rho^n \lambda e \sigma^n + \sum_{n=1}^{\infty}(1-\rho)\rho^n \mu \sigma^{n-1}. \end{aligned}$$

This is the sum of two geometric progressions. C is unbounded if $\rho\sigma > 1$. If $\rho\sigma < 1$, then

$$\begin{aligned} C &= (1-\rho)\frac{\lambda e + \mu\rho}{1-\sigma\rho} \\ &= (1-\rho)\lambda\frac{e+1}{1-\sigma\rho}. \end{aligned}$$

Hence, C is a constant. The experimental cost of an insertion or a deletion is thus $O(1)$.

Queuing system with discouraged arrivals: $M/M/\infty$

For an $M/M/\infty$ queuing system with discouraged arrivals [27], the arrival rate is a constant $\lambda/(n+1)$, and the departure rate is a constant μ for every state other than E_0. The steady state probability of being in state E_n is $e^{-\rho}\rho^n/n!$, where $\rho = \lambda/\mu$. Substituting these values in Equation 5.1, the expected cost of insertion or deletion is

$$C = \sum_{n=0}^{\infty} p_n(\lambda_n R_n + \mu_n D_n)$$

$$
\begin{aligned}
&= \sum_{n=0}^{\infty} e^{-\rho}\frac{\rho^n}{n!}\frac{\lambda}{n+1}e\sigma^n + \sum_{n=1}^{\infty} e^{-\rho}\frac{\rho^n}{n!}\mu\sigma^{n-1} \\
&= \frac{\lambda e}{\rho\sigma}\sum_{n=1}^{\infty} e^{-\rho}\frac{\rho^n}{n!}\sigma^n + \frac{\mu}{\sigma}\sum_{n=1}^{\infty} e^{-\rho}\frac{\rho^n}{n!}\sigma^n \\
&= \frac{\mu(e+1)}{\sigma}e^{-\rho}(e^{\rho\sigma}-1).
\end{aligned}
$$

We summarize these results in the following theorem.

Theorem 5.3.1 *For an $M/M/1$ queuing system with $\rho\sigma > 1$, the DPH algorithm requires an expected insertion time of $O(1)$, an expected deletion time of $O(1)$, and a worst case find time of $O(1)$. For an $M/M/\infty$ queuing system, the DPH algorithm requires an expected insertion time of $O(1)$, an expected deletion time of $O(1)$, and a worst case find time of $O(1)$.*

5.4 Dynamic Perfect Hashing with Varying Array Size

In Section 5.3, we analyzed the performance of the DPH algorithm when the stochastic behavior of n is known in advance. In this section, we assume that we do not know the behavior of n, and we vary N as n varies. However, changing N requires remapping all the words in S to a new array A. Since such remapping is expensive, we vary N only when n crosses certain boundaries according to the following rules.

- N can be only one of $\lceil(1+\delta)^i\rceil$, where $i > 0$, and δ is a given positive constant.
- If $N = \lceil(1+\delta)^i\rceil$, then $\lceil(1+\delta)^{i-1}\rceil < n \leq \lceil(1+\delta)^{i+1}\rceil$
- When $n = \lceil(1+\delta)^{i+1}\rceil$, and the next operation is an insertion, N is increased to $\lceil(1+\delta)^{i+1}\rceil$, and all the words in S are remapped to A.
- When $n = \lceil(1+\delta)^{i-1}\rceil + 1$, and the next operation is a deletion, N is decreased to $\lceil(1+\delta)^{i-1}\rceil$, and all the words in S are remapped to A.

Since N always varies in tandem with n, the following relation always holds true.

$$\frac{1}{1+\delta} < \frac{n}{N} \leq 1+\delta$$

Recall from Section 5.2 that

$$\begin{aligned} R_n &= e\left(1+\frac{e-1}{N}\right)^n \\ &< e \cdot e^{(e-1)n/N} \\ &= e^{1+(e-1)(1+\delta)} \\ &= e^{e+\delta(e-1)} \\ &= O(1). \end{aligned}$$

Similarly, $D_n < e^{(e-1)(1+\delta)} = O(1)$. Hence,the expected cost of an individual insertion or deletion is constant. However, this ignores the cost of occasional remapping.

Hence, let us now calculate the expected cost of remapping the words of S into A. The expected number of bins of size j is given by

$$N\left(\begin{array}{c} n \\ j \end{array}\right)\left(1-\frac{1}{N}\right)^{n-j}\frac{1}{N^j}.$$

The expected cost to find a PHF for a bin of size j is e^j. As we prove in the following paragraph, the total cost of finding PHFs for all the bins is linear in N. So let us denote the total cost of remapping by NC_R.

$$\begin{aligned} NC_R &= N\sum_{j=0}^{n}\left(\begin{array}{c} n \\ j \end{array}\right)\left(1-\frac{1}{N}\right)^{n-j}\frac{1}{N^j}e^j \\ &= N\left(1+\frac{e-1}{N}\right)^n. \end{aligned}$$

Whenever the words in S are remapped to A, n is either N or $N-1$. So we can say

$$\begin{aligned} C_R &= \left(1+\frac{e-1}{N}\right)^n \\ &\approx e^{e-1}n/N \\ &\approx e^{e-1}. \end{aligned}$$

In the worst case, the next remapping could occur after $\delta N/(1+\delta)$ deletions. Amortizing the cost of this remapping over these deletions, the amortized cost C_A would be

$$\begin{aligned} C_A &= \frac{NC_R}{\delta N/(1+\delta)} \\ &= \left(1+\frac{1}{\delta}\right)e^{e-1}. \end{aligned} \tag{5.2}$$

Equation 5.2 means that the amortized cost C_A is $O(1)$ as long as δ is $\Omega(1)$. However, if we try to make δ as $o(1)$, C_A becomes $O(1/\delta)$. Recall that N is at most $n(1+\delta)$. This means that if we try to be very efficient by keeping δ very low, we drive C_A very high. On the other hand, if we make δ very high to keep C_A very low, we waste a lot of space. We can summarize this tradeoff between space and time by the following theorem.

Theorem 5.4.1 *If the DPH algorithm guarantees that the size of array A is at most $(1+\delta)n$, then the amortized cost per insertion or deletion is at least $O(1/\delta)$.*

Chapter 6

CONCLUSIONS AND FUTURE WORK

In this chapter, we summarize the results of this dissertation, while identifying some interesting open problems.

The MOS algorithm

Previously, no analysis existed for the running time of the MOS algorithm. In this thesis, we prove that the MOS algorithm requires $\Omega(n^{1+\eta})$ time if SPACE(f) is $n/(\eta \ln 2)$ bits. To find a PHF f for a set of variable length words, the MOS algorithm uses three arrays—A, B, and C. The array C contains the characters of the words. The array B contains pointers to the words in C. B is filled by randomly tossing the words repeatedly into it until they land in empty cells. For future research, an interesting alternative is to increase the size of C and toss the words directly into C to see if this results in improved performance.

The MC algorithm

In this thesis, we develop the MC algorithm, and provide experimental evidence that it runs faster than the MOS algorithm in its entire range. In future research, it would be very interesting if the time complexity of the MC algorithm could be analyzed, and its behavior predicted.

Lower bound on the size of a PHF

Most of the previous perfect hashing schemes have modeled a dictionary as a set of fixed-length words. It is possible to store all the words of a dictionary as null-terminated

strings in an array of characters and use a pointer to the word to provide access to the word via indirection. This model is adequate for dealing with the asymptotic values of the time and space requirements of an algorithm. However, when we need more precise values for the space requirement, we must take the space occupied by pointers into account. Otherwise, the results of comparing two perfect hashing algorithms can be misleading. Our model is the first to recognize this distinction. Hence, consider two kinds of universes, one containing words of fixed-length and another containing words of variable-length. This provides a precise framework for comparing two perfect hashing algorithms.

Mehlhorn [32] proves a bound of $n/\ln 2$ bits on SPACE(f), where f is a PHF for a set of n fixed-length words. Fox et al. [15, 16, 17] prove a bound of $n \log_2 n - o(n)$ bits on SPACE(f), where f is an OPPHF for a set of n fixed-length words. The following are two important open problems on the time and space requirements of perfect hashing functions.

- Is there a better lower bound than Mehlhorn's bound on SPACE(f), if f is a PHF for variable-length words?
- Is there a better lower bound than that of Fox et al. on SPACE(f), if f is an OPPHF for variable-length words?
- At present, there is no perfect hashing algorithm for fixed-length words which runs in $O(n)$ expected time and which finds a PHF f of size $\Theta(n)$ bits. Is there a nontrivial lower bound on the time required to find a PHF of size $\Theta(n)$ bits? If there is one, is there a similar nontrivial lower bound on variable-length words?

Dynamic perfect hashing

In this thesis, we develop a new perfect hashing algorithm(DPH algorithm) for dynamic dictionaries. We analyze its behavior in two cases. In the first case, we keep the size of the array A fixed. We model the dictionary as a birth and death process and prove that the average case insertion and deletion times are $O(1)$ for $M/M/1$ and $M/M/\infty$ queues. In the second case, we vary the size of A as n varies. We prove that the amortized average

case time is $O(1)$ for insertion and deletion. The following is a list of some open problems related to dynamic perfect hashing.

- What kind of queueing models describe the dynamic dictionaries encountered in practical computer applications?
- In the DPH algorithm, for each i, we maintain M_i memory blocks of size i. If a fixed amount of memory is given at the beginning of the algorithm, what fraction of it should be allocated to each size to ensure maximum running time before the algorithm runs of out of blocks of a particular size? How does the variation of the size of the table A affect this allocation?
- One of the problems in maintaining a dynamic dictionary of variable-length words is the maintenance of the character strings of the words itself. This problem is similar to the memory management problem in operating systems. However, there is an important difference between the two problems. In the operating systems problem, the sizes of the memory blocks are quite large. Therefore, the free memory blocks contain enough space for storing a pointer to the next block. However, that is not the case for maintaining memory for character strings. In future research, it would be interesting to explore good strategies for maintaining a dynamic set of character strings.

REFERENCES

[1] A.V. Aho and D. Lee. Storing a dynamic sparse table. In *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science*, pages 55–60, Toronto, Canada, October 1986.

[2] Béla Bollobás. *Random Graphs.* Academic Press, New York, 1985.

[3] N. Cercone, M. Krause, and J. Boates. Minimal and almost minimal perfect hash function search with application to natural language lexicon design. *Computers and Mathematics with Applications*, 9:215–231, 1983.

[4] C.C. Chang. The study of an ordered minimal perfect hashing scheme. *Journal of the ACM*, 27:384–387, 1984.

[5] C.C. Chang. Letter oriented reciprocal hashing scheme. *Information Sciences*, 38:243–255, 1986.

[6] Qi Fan Chen. *An Efficient Object-Oriented Database for Information Retrieval Applications.* PhD thesis, Virginia Polytechnic Institute and State University, Department of Computer Science, March 1992.

[7] R.V. Churchill. *Complex Variables and Applications.* McGraw-Hill Company, New York, 1974.

[8] R.J. Cichelli. Minimal perfect hash functions made simple. *Communications of the ACM*, 23:17–19, 1980.

[9] G.V. Cormack, R.N.S. Horspool, and M. Kaiserworth. Practical perfect hashing. *The Computer Journal*, 28:54–58, 1985.

[10] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Analysis of Algorithms.* MIT Press, Cambridge, Massachusetts, 1990.

[11] Amjad Daoud. *Data Structures for Efficient Information Retrieval.* PhD thesis, Virginia Polytechnic Institute and State University, April 1993.

[12] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer, H. Ronhert, and R.E. Tarjan. Dynamic perfect hashing. In *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 524–531, White Plains, New York, October 1988.

REFERENCES

[13] R.J. Enbody and H.C. Du. Dynamic hashing schemes. *ACM Computing Surveys*, 20:85–113, 1988.

[14] W. Feller. *An Introduction to Probability Theory and Applications*, volume 1. John Wiley & Sons, New York, 1968.

[15] Edward A. Fox, Qi Fan Chen, Amjad M. Daoud, and Lenwood S. Heath. Order preserving minimal perfect hash functions and information retrieval. In *Proc. SIGIR 90, 13th Int'l Conference on R&D in Information Retrieval*, pages 279–311, Brussels, Belgium, September 1990.

[16] Edward A. Fox, Qi Fan Chen, Amjad M. Daoud, and Lenwood S. Heath. Order preserving minimal perfect hash functions and information retrieval. *ACM Transactions on Information Systems*, 9(2):281–308, July 1991.

[17] Edward A. Fox, Qi Fan Chen, Amjad M. Daoud, and Lenwood S. Heath. Order preserving minimal perfect hash functions and information retrieval. Technical Report TR-91-1, Virginia Polytechnic Institute and State University, Department of Computer Science, February 1991.

[18] Edward A. Fox, Qi Fan Chen, and Lenwood S. Heath. A faster algorithm for constructing minimal perfect hash functions. In *Proceedings of the 15th Annual International Conferene on Research and Development in Information Retrieval*, pages 266–273, 1992.

[19] Edward A. Fox, Qi Fan Chen, Lenwood S. Heath, and Sanjeev Datta. A more cost effective algorithm for finding perfect hash functions. Technical Report TR-88-30, Virginia Polytechnic Institute and State University, Department of Computer Science, Blacksburg, VA, September 1988.

[20] Edward A. Fox, Qi Fan Chen, Lenwood S. Heath, and Sanjeev Datta. A more cost effective algorithm for finding perfect hash functions. In *Proc. ACM 1989 Computer Science Conference*, pages 114–122, Louisville, KY, February 1989.

[21] Edward A. Fox, Lenwood S. Heath, and Qi Fan Chen. An $O(n \log n)$ algorithm for finding minimal perfect hash functions. Technical Report TR-89-10, Virginia Polytechnic Institute and State University, Department of Computer Science, Blacksburg, VA, April 1989.

[22] Edward A. Fox, Lenwood S. Heath, Qi Fan Chen, and Amjad M. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1):105–121, 1992.

[23] M.L Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 35:538–544, 1988.

REFERENCES

[24] G.L. Gonnet and P. Larson. External hashing with limited internal storage. *Journal of the ACM*, 35:161–184, 1988.

[25] John E. Hopcroft and Jeffrey D.Ullman. *Automata and Formal Languages and Computability*, chapter 1. Addison Wesley Publishing Company, Reading, Massachusetts, 1979.

[26] G. Jaeschke. Reciprocal hashing—a method for generating minimal perfect hash functions. *Communications of the ACM*, 24:829–833, 1981.

[27] L. Kleinrock. *Queuing Systems*, volume 1, chapter 3. John Wiley & Sons, New York, 1975.

[28] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison Wesley Publishing Company, Reading, Massachusetts, 1971.

[29] V.F. Kolchin, B.A. Sevastyanov, and V.P. Chistyakov. *Random Allocations.* John Wiley & Sons, New York, 1978.

[30] P. Larson and M.V. Ramakrishna. External perfect hashing. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 190–199, Austin, Texas, 1985.

[31] Bohdan S. Majewski. *Minimal Perfect Hash Functions.* PhD thesis, Dept of Computer Science, University of Queensland, Australia, November 1992.

[32] K. Mehlhorn. On the program size of perfect and universal hash functions. In *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*, pages 170–175, Chicago, Illinois, November 1982.

[33] P.K. Pearson. Fast hashing of variable length text strings. *Communications of the ACM*, 33:677–680, 1990.

[34] M.V. Ramakrishna and P. Larson. File organization using composite perfect hashing. *ACM Transactions on Database Systems*, 14:231–263, 1989.

[35] T.J. Sager. A polynomial time generator for minimal perfect hash functions. *Communications of the ACM*, 28:523–532, 1985.

[36] T.J. Schaeffer. The complexity of satisfiability problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 216–226, San Diego, California, May 1978.

[37] Steven S. Seiden and Daniel S. Hirschberg. Finding sucinct ordered minimal perfect hash functions. personal communication.

[38] A. N. Shiryayev. *Probability.* Springer-Verlag, New York, 1984.

[39] R. Sprugnoli. Perfect hashing functions a single probe retrieving method for retrieving static sets. *Communications of the ACM*, 20:841–850, 1978.

[40] R. Wong. *Asymptotic Approximations of Integrals.* Academic Press, New York, 1989.