ALGORITHMIC APPLICATIONS OF DATA COMPRESSION TECHNIQUES

by

Shenfeng Chen

Department of Computer Science
Duke University

Date: 12/4/96

Approved:

John H. Reif, Supervisor

Jeffrey S. Vitter

Henry S. Greenside

Carla Ellis

Ming Y. Kao

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

1996

ABSTRACT

(Computer Science)

ALGORITHMIC APPLICATIONS OF DATA COMPRESSION TECHNIQUES

by

Shenfeng Chen

Department of Computer Science
Duke University

Date: 12/4/96

Approved:

John H. Reif, Supervisor

Jeffrey S. Vitter

Henry S. Greenside

Carla Ellis

Ming Y. Kao

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

1996

Abstract

In the process of algorithm design, an innovative use of certain data structure or computational technique suited for the given problem can greatly improve the performance of the algorithm. In this thesis we investigate the problem of using techniques and data structures in data compression area to aid design and analysis of algorithmic problems in other areas. The study is motivated by the fact that the best compression algorithms are the ones that utilize various techniques to explore and predict the statistical pattern of the inputs well and then tailor the algorithms to suit the input pattern in most efficient way.

First we give an overview on data structures and techniques in data compression area. We then demonstrate the effectiveness of data compression techniques by case studies of four fundamental problems in different areas: sorting, string matching, volume rendering and graph compression. We assume certain statistical properties of the input source that hold widely for practical input data. For example, in the sorting problem we assume that the inputs are binary keys drawn from a stationary and ergodic source with bounded entropy, while in the volume rendering problem we assume the inputs are 3-D image data samples. Based on these assumptions, we present novel algorithms for these problems which utilize various data structures and techniques in data compression area to speed up computation. The time complexities of our algorithms are determined by the specific input assumptions and data compression techniques used. Data compression techniques, such as unique parsing, prefix matching, dictionary construction, lossy image compression and Huffman coding, are used to aid many aspects of algorithm design such as algorithmic procedures, operations on data structures, theorem proving and complexity analysis.

Based on a concrete study of these problems, we propose a theoretical framework

which identifies the class of problems which can be solved more efficiently by applying data compression techniques to adapt to certain statistical assumptions of the input source. Finally, we outline several interesting topics as our future work.

Part of this work was Supported by NSF Grant NSF-IRI-91-00681, Rome Labs Contracts F30602-94-C-0037, ARPA/SISTO contracts N00014-91-J-1985, and N00014-92-C-0182 under subcontract KI-92-01-0182.

Acknowledgements

To begin with, I would express my sincere appreciation to my advisor, Dr. John H. Reif, for his constant support and encouragement during the course of this work. I will always remember the patience he had in teaching me the basics of scientific research when we wrote the first paper. His extensive knowledge and endless pool of new ideas inspired me to write a thesis on applicatinos of data compression algorithms and helped me to overcome many technical difficulties along the way.

I have the pleasure to thank Dr. Ming Y. Kao for his valuable discussions and suggestions which helped me to solve some critical problems, both in research and in personal life. Special thanks to Dr. Jeffrey S. Vitter for his valuable advices on improving my presentation skills which proved to be extremely important in my job seeking. I wish to thank the other members of my committee: Dr. Henry G. Greenside and Dr. Carla Ellis for their kind advices and helpful suggestions.

I also would like to thank my fellow graduate students at Duke. Thanks to Thomas Alexander, Apratim Purukayastha, Tong Luo and Nasrin Azarbayejani for helpful discussions during breaks, office sharing and tennis playing. Entering and graduating in the same year, working under the same advisor, Hongyan Wang has always been there to help when I met problems either in course study or research. Keehang Kwon has always been hard-working and taught me to be devoted to my work by his own example. Dr. Philip Long, Zhiyong Li, Dzung Hoang and Peter Mills kindly provided me with literature on research problems and many useful discussions. Sharing a learning experience with them was a real pleasure.

Lastly, and the most importantly, I have to thank my family for supporting me in five years of study abroad. Every time I went back for a short visit, I was greeted with my favorite delicious dishes and warm conversations. This gave me enough energy to

come back and continue my study thousands of miles away from home.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the process of algorithm design for solving a given problem in computer science, an innovative use of certain data structure or computational technique suited for the given problem can greatly improve the performance of the algorithm. In many cases, the data structure or technique are already well known in solving different type of problems and therefore we only need to show that it is applicable to the new problem. Usually these problems have similar algorithmic properties so that the data structure or technique borrowed is extendible to the new problem only with minor modifications. Some of the complexity analysis techniques often fall into this category. For example, a wide range of problems are similar to one other in the sense that a sequence of operations is performed in these problems and the common goal of algorithms solving these problems is to minimize the cost of an operation averaged over a long sequence. The *amortized* [26, 103] running time analysis is well suited for solving this type of problems. Unlike in average case running time analysis, in amortized analysis the probability distribution of different operations is not needed and the amortized cost of each operation holds even in the worst case.

Another strategy to improve the performance of an algorithm is to exploit certain statistical properties of the inputs and to base the algorithm and its analysis on these specific statistical properties. The new algorithm is tailored to explore these properties and reduce the computation. If the practical inputs can be shown to demonstrate the restricted statistical properties with high likelihood, the algorithm thus achieved can prove to be very efficient in practice. A simple example of this type of assumption is the design of the bucket sort algorithm [26]. The bucket sort

divides the range of input keys into buckets each of equal size. Although the worst case happens when all the input keys fall into a single bucket and the running time of the algorithm degrades to quadratic time, it is reasonable to assume an uniform distribution of input keys which is true for many practical inputs. Based on this assumption, the bucket sort efficiently runs in an expected time of $O(n)$ where n is the number of input keys.

Motivated by the above observations, we study the problem of applying data structures and algorithmic techniques in data compression to develop new algorithms and to improve existing algorithms in other domains. In the design and analysis of the algorithms, we often make assumptions on the statistical properties of the inputs and these assumptions are often empirical facts that are valid for most practical inputs.

We concentrate our study on basic problems such as sorting and string matching. These basic problems are easy to define as follows:

Sorting. Given a set S of n numbers, return a set S' consisting of these numbers in sorted order.

String Matching. Given a text T of length n and a pattern P of length m, find the occurrence or all occurrences of the pattern in the text.

In the case of sorting problem, we propose a new algorithm based on a specific data compression scheme with assumption on compressibility of the input keys. Specifically, the algorithm utilizes some data structure and technique well known in data compression algorithm design and they prove to be extremely useful in the new sorting algorithm as well. We also give a parallel version of the algorithm on certain parallel machine model. In the string matching problem, we develop a new algorithm with the assumption on the compressibility of the strings and extend it to the two-dimensional case. The analysis of the algorithm depends on the assumption on the

compressibility of the input string as well.

Based on a concrete study of these problems, we propose a theoretical framework for applying data compression data structure techniques to solve algorithmic problems in other domains. Later in this thesis, we extend the object we study from simple binary strings to more complex data types (e.g., trees and graphs). In particular, we study the volume rendering problem and the graph compression problem. The problems can be defined as follows:

Volume Rendering. Given three-dimensional arrays of 3D pixels of size $N \times N \times N$ and discrete samples of volume data, output an $N \times N$ rendered image.

Tree and Graph Compression. Given a tree T or graph G, return an encoded tree T' or encoded graph G' along with a dictionary D so that T' is an encoding of T or G' is an encoding of G, respectively, in compressed form.

1.1 Information Theory and Statistical Background

In this section, we review some basic terminologies and definitions in information theory which are of fundamental use throughout this thesis. A *stochastic process* $\{X_k\}_{k=1}^{\infty}$ is a sequence of random variable where X_k is generated from a data source $\mathcal{X}$ according to certain underlying probability mass function $\Pr(x_1, x_2, ..., x_n)$ where x_i is an element of a finite alphabet $\mathcal{A}$ with cardinality $|\mathcal{A}| = A$.

Throughout this thesis, we often make the assumptions that the sequence of the input data is generated from a *stationary* and *ergodic* statistical source. We call $\{X_k\}_{k=1}^{\infty}$ is a *stationary* process if the joint probability distribution of any subset of the sequence is unchanged (*invariant*) with respect to shifts in the time index, i.e.,

$$\Pr\{(X_1^n) = (x_1, x_2, ..., x_n)\} = \Pr\{(X_{1+t}^{n+t}) = (x_1, x_2, ..., x_n)\} \tag{1.1}$$

for every time shift t, for all $(x_1, x_2, ..., x_n) \in \mathcal{A}^n$ and for all $n \geq 0$.

Informally, an *ergodic* source [34]is the most general dependent source for which the strong law of large number holds. To define ergodicity formally, we require some definitions from probability theory. An ergodic source is defined on a probability space (Ω, β, P), where β is a sigma-algebra of subsets of Ω and P is a probability measure. A random variable X is defined as a function $X(\omega), \omega \in \Omega$, on the probability space. We also have a transformation $T : \Omega \to \Omega$, which plays the role of a time shift. We will say that the transformation is *stationary* if $\mu(TA) = \mu(A)$, for all $A \in \beta$. Furthermore, the transformation is called *ergodic* if every set A such that $TA = A$, satisfies the condition $\mu(A) = 0$ or 1. If this transformation T is stationary and ergodic, we say that the process defined by $X_n(\omega)=X(T^n\omega)$ is stationary and ergodic. For a stationary and ergodic source with a finite expected value, Birkhoff's *ergodic theorem* states that

$$\frac{1}{n}\sum_{i=1}^{n} X_i(\omega) \to EX = \int X dP. \tag{1.2}$$

with probability 1.

Entropy [7, 34] is another basic concept in information theory. It is a measure of the amount of order or redundancy contained in a data sequence generated from a given statistical source. Entropy is small when the data sequence contains a large degree of order and big when it contains little order. For example, consider selecting an event from a set of n different events each with probability $p_1, p_2, ..., p_n$ respectively. The function suitable for describing the entropy of this event in this case is

$$H(p_1, p_2, ..., p_n) = -\sum_{i=1}^{n} p_i \log_2 p_i. \tag{1.3}$$

In this case, the entropy is also related to the complexity of description of this event in the sense that it is the minimum length of code necessary to describe the event. Similarly, if a sequence X_1^n is generated by an arbitrary stochastic process $\{X_k\}_{k=-\infty}^{\infty}$,

the entropy $H(X_1^n)$ is defined as the minimum length of code necessary to describe this sequence.

We now define another term, the *entropy rate* of a stochastic process, which is used throughout this thesis. The reason why this concept is of more importance is that it is closely related to the optimal compression ratio of data sources. We will explore this relation further in Chapter 2 of this thesis. Intuitively, the entropy rate of a stochastic process is the average number of bits needed to describe each character of a sequence generated by the stochastic process $\mathcal{X}$. Formally, for arbitrary i and j such that $1 \leq i < j \leq \infty$, let X_i^j denote the substring $(X_i, X_{i+1}, ...X_j)$. Let

$$H_n = \frac{1}{n} H(X_1^n) \tag{1.4}$$

$$\triangleq -\frac{1}{n} \sum_{X \in \mathcal{A}^n} \Pr\{X_1^n = X\} \log(\Pr\{X_1^n = X\}) \tag{1.5}$$

be the nth order (normalized) entropy of $\mathcal{X}$, and let $H(\mathcal{X}) = \lim_{n\to\infty} H_n$ be the entropy of the input source. It is well known [34] that this limit always exists and thus we can encode the data source $\mathcal{X}$ using $(H + \epsilon)$ bits per source symbol for arbitrary $\epsilon > 0$. Throughout this thesis, we will call this H to be the *entropy* of any given stochastic process.

1.2 Machine Models

In the analysis of sequential algorithms throughout this thesis, we assume a generic one-processor RAM (random access machine) [26] model of computation unless otherwise specified. In the RAM model, simple instructions, such as addition, multiplication, logical and/or, and memory access (read/write) can be carried out one after another in a single machine cycle with no concurrent operations.

Even though RAM model ignores many relevant architecture details of machine

(e.g. size of memory, paging, number of registers), a careful asymptotic analysis of sequential algorithm based on RAM model generally is a good indicator of the practical performance of the algorithm. In later chapters of this thesis, when the actual machine structure becomes an issue of the complexity analysis, some of the architecture details will have to be taken into consideration for a more accurate analysis.

For parallel algorithm design, we assume a multi-processor PRAM (parallel random access machine) model. In the PRAM model, there are p ordinary processors $P_0, P_1, ..., P_{p-1}$ which share a global memory. All processors can read from or write to the shared global memory at the same time, each in the same way as in normal sequential RAM. Simple instructions can be executed independently on processors in parallel. The running time $t(n)$ of a parallel algorithm is measured by the longest parallel steps taken by the processors. The *work* $w(n)$ of a parallel algorithm is the total number of arithmetic or logical instructions executed. We call a parallel algorithm is *work efficient* if the total work $w(n) = T(n)$ where $T(n)$ is the running time of the best known sequential algorithm. With different allowance of concurrency in memory access, PRAM are further divided into EREW, CREW and CRCW machines. EREW machines allow no concurrent read or write. CREW machines allow concurrent read only and CRCW allow both concurrent read and write. In addition, various versions of CRCW machines specify different way to resolve multiple writes to the same location of the shared memory. The *arbitrary* CRCW model selects an arbitrary processor to write in the memory location "successfully" while the *priority* CRCW selects the winner according to a pre-assigned set of priorities. In the parallel implementation of our algorithms, we will use arbitrary CRCW as our parallel model.

1.3 Data Compression

There has been extensive research in data compression algorithms and techniques [7, 51, 52, 57, 113, 114]. The primary purpose of the data compression algorithms is to minimize the storage space and the transmission cost of large size data. The compression algorithms can be classified in two groups: lossless compression which encodes the original input without loss of information and the lossy compression with some distortion of the original input in the compressed form.

Lossy data compression are usually implemented by representing an input source by mapping each source item to one of finite number of dictionary elements that minimizes distortion under a suitable metric. Quantization is a fundamental example of lossy compression. Scalar Quantization (SQ) observes a single source datum and selects the nearest approximation value from a predetermined finite set of allowed numerical values. Vector Quantization (VQ) is a generalization of SQ to quantization of a vector, a point in n-dimensional Euclidean or any other metric space, according to some vector space metric such as the metric induced by the ℓ^2 norm. VQ can be used to predict the future data by filtering primary components to maintain correlations among different values. Predictive VQ (PVQ) schemes were introduced by Fischer and Tinnen [42] as well as Cuperman [36].

Lossless compression, on the other hand, can be achieved by assigning shorter codes to frequently occurring source sequence, and vice versa. Lempel-Ziv (LZ) compression [113], which includes a group of lossless compression algorithms, is an well known example of this technique. The importance of LZ compression, including its data structure and techniques, to this thesis stems from the fact that a good prediction of input sequence can be achieved by similar techniques used in LZ compression scheme (see next section for a description on predictive computing). In LZ compression, no two phrases in the parse of the source are identical, a property called unique

parsing. It has been shown[113] that the unique parsing used in LZ compression makes the compression optimal for sequences generated from stationary and ergodic source. In other words, LZ compression scheme achieves optimal compression ratio in the limit as the size fo the inputs goes to infinity. Vitter and Krishnan [108] extend this result by proving that optimal compression algorithms can be converted into optimal prediction algorithms assuming stationary ergodic distribution of the inputs.

The data structure and related techniques used in LZ compression are of critical use in design and analysis of algorithms throughout this thesis. In Chapter 2 of this thesis, we will give a detailed description of LZ compression with its data structures and techniques.

1.4 Predictive Computing

To design efficient algorithms based on certain statistical properties (e.g. probability distribution, entropy and compressibility) of inputs, one needs a good predictor to take advantage of these properties. If the next character of the input sequence can be predicted in advance with high probability, the algorithm can be adjusted accordingly to reduce computation. This idea has spurred a great deal of research in recent years from dynamic resource management to optimal prefetching in cache memory design. In general, an algorithm based on predictive computing model approaches a given problem by exploring the regularity of the input and trying to learn from the past to predict the future input. The core of the algorithm is a predictor engine tuned to dynamically adjust to the input distribution. The result is a reduced computation time.

The predictive computation model for processing large amount of data is relatively new in computer science community. Historically, three paradigms have been adopted in analyzing the performance of algorithms, with different assumption of the inputs.

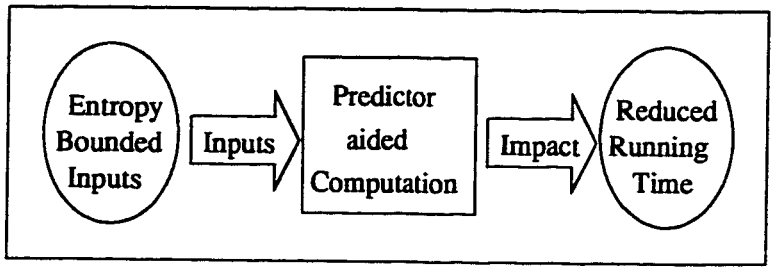

Figure 1.1: Predictive computing model.

Worst case inputs paradigm analyzes the complexity of the algorithm assuming the inputs are of the worst case, e.g. Sleator and Tarjan's [103] competitive model for analyzing on-line algorithms. However, in practice it is too pessimistic to assume that the inputs are always of the worst case. There are many examples of algorithm where the worst case running time is much more than the average while the probability of a worst case is very small. In such cases, it is desirable to analyze the running time based on other assumptions about the inputs instead of worst case. For example, Boyer and Moore [26] gave a string matching algorithm which has a worst case running time of $O(mn)$ where n is the size of the text and m is the size of the pattern. This running time is equivalent to the running time of a brute search. However, the worst case only occurs in very rare situations where there are extremely frequent repetitions in the text. In fact, Bayor and Moore's string matching algorithm is one of the fastest string matching algorithms in practical use.

On the other hand, the second paradigm, *random case inputs* paradigm, assumes that the inputs are generated from a source which is of uniform distribution. This paradigm simplifies the analysis of many algorithms, e.g. the bucket sort algorithm . But the practical inputs are not uniformly distributed in most cases.

Instead, the predictive computing paradigm, which we will use throughout this thesis, only assumes that the inputs has a specific statistical property and this assumption can be shown to widely hold for many large practical data sets.

Prediction algorithms try to predict the future data of an input stream based on the data known from the past. It also assumes that the data stream has a certain probability distribution and the future data has the same distribution as well as the past. As in data compression scheme, a good predictor must learn from the history to form a probabilistic model of the input data and then apply the model to predict the probabilities of occurrence of possible text pieces. The text piece could be a character of the alphabet from which the input data is drawn in a character-based predictor or a word in a word-based predictor.

The similarity between the data compression and prediction algorithms makes many techniques and data structures used in data compression algorithms applicable to prediction. Cover and Thomas [34] provide a theoretical background to information complexity issues establishing a direct relationship between high compressibility and high predictability, particularly for stationary source.

Data compression techniques are designed to compress large amount of data to a small portion of original size. A well known compression scheme designed by Lempel and Ziv [113] represents a piece of string by the position of its previous occurrence. Lempel-Ziv compression is proven to achieve optimal compression ratio when the size of the original text $n \rightarrow \infty$.

Recent work in computational learning theory [16] has shown that prediction is synonymous with generalization and data compression. In order to compress data well, the data compressor has to be able to predict future data well, and hence a good data compression scheme should also be a good predictor. Based on the above reasoning, the data structures and techniques used in highly efficient data compression algorithms can be exploited to produce optimal predictions of a data stream by sampling a prefix of it. These techniques are usually adaptive in the sense that they dynamically modify the prediction on-line as the pattern of the incoming

data stream changes.

The algorithms based on predictive computing model assume that the inputs are generated from a source with certain statistical properties. For example, many algorithms based on predictive computing model assume that the inputs are generated from a stationary source with unknown probability distribution. In general, such algorithms can be partitioned into two parts (see figure 1.1). The first phase is a predictive engine phase which analyzes the previous inputs to determine the underlying probability distribution of the input source and then predicts the future inputs. The second part is an adaptive algorithm which adapts the solution based on the pattern of the inputs to improve the performance of the algorithms. The second part depends specifically on the problem to be solved while the predictive engine can utilize a wide range of techniques in many areas such as data compression and associative matching algorithms for better prediction and thus better efficiency.

1.4.1 Assumptions on Practical Inputs

Several assumptions on practical inputs are used in different algorithms throughout this thesis:

Bounded Entropy. In practical large data sets, it is known that the entropy of the data sets can be bounded within a certain constant. For example, the normal lossless compression ratio is 3–4 for English text, 1.5–20 for system files and generally no more than 20 for files containing business and financial records [7, 102]. As the entropy is inherently related to the compressibility of the data while the compressibility of the data plays an essential factor in determining the complexity of data compression techniques, this assumption greatly reduces the running time of the algorithm based on data compression techniques.

Stationarity and Ergodicity of Input Source. In this thesis, we normally model the input as a sequence of data generated by a stationary and ergodic statistical source (see previous section for definition of stationarity and ergodicity). This assumption holds for most practical data source which is stable and continuous. Under the ergodicity assumption, the analysis of algorithm becomes simpler because the statistical property holds the same for all sequences independent of their starting position.

Probability Distribution of Inputs. In some algorithms, we assume that the input source always follows the same probability distribution which is unknown at the beginning so that the learning phase of the algorithm can gradually gain knowledge of this probability distribution. This assumption directly follows from the stationarity of the input source.

1.5 Other Notation and Terminology

Besides various techniques in data compression area, this thesis also utilizes certain techniques which are not directly related to our topic but nonetheless are important in design and analysis of many algorithms. In this section, we will give a brief review of these techniques and also introduce a well known result of probability analysis - Chernoff bounds.

1.5.1 Randomized Algorithms

Randomization was formally introduced by Rabin [88] and independently by Solovay & Strassen [101] for improving performance of certain algorithms. In a randomized algorithm, the processor is allowed to make decisions according to random bits at different steps of the algorithm. Thus one randomized algorithm actually corresponds to a family of algorithms each member corresponds to a fixed sequence of the random

bits. Two of the most commonly used forms of randomization are the *Las Vegas* algorithms and *Monte Carlo* algorithms. The former type of algorithm ensures that the output of the algorithm is always correct. However, only a fraction (typically greater than 1/2) of the members in the algorithm family finish within certain time bound. The latter type, the Monte Carlo algorithm, in contrast, will always finish within the time bound while only a fraction (normally more than one half) of the members in the algorithm family output correct results. For our discussion, we shall limit ourselves to the Las Vegas algorithms which have been more popular in algorithm design.

Because a randomized algorithm may have different running time each time we run it, the time bound for a randomized algorithm is thus defined as the *expected* running time, i.e., the algorithm will run within the time bound with high probability. In this thesis, we use the term *high probability* or *high likelihood* interchangeably. More precisely, they denote the probability greater than $1 - n^{-\alpha}$ for some $\alpha > 1$ where n is the input size.

In this thesis, some randomized techniques (e.g. random sampling) are used to improve the efficiency of the algorithm and in many cases, make the presentation of an algorithm succinct. It is noteworthy that once a randomized technique is introduced into the algorithm, the running time of the algorithm becomes expected running time instead of worst case time bounds.

1.5.2 Chernoff Bounds

In complexity analysis of a randomized algorithm, Chernoff bounds [23, 33] have been proven to be extremely useful. The general form of Chernoff bounds for a discrete distribution can be stated as follows:

$$\Pr[A \geq x] \leq z^{-x} G_A(z) \tag{1.6}$$

where $G_A(z)$ is the probability generating function. We minimize the bounds by substituting $z = z_0$ to minimize the right side expression.

A Bernoulli trial is an experiment with two possible outcomes: success and failure. The probability of a success is p and a failure is $q = 1 - p$.

A binomial variable X with parameter (n, p) is the number of successes in n independent Bernoulli trials, the probability of success in each trial being p. The probability mass function of X in this case is:

$$\Pr(X \leq x) = \sum_{k=0}^{x} \binom{n}{k} p^n (1-p)^{n-k} \tag{1.7}$$

We can bound the tail end of the Binomial variable using the Chernoff bounds. More frequently used are the following approximations due to Augluin and Valiant:

$$\Pr(X \geq m) \leq \left(\frac{np}{m}\right)^m e^{m-np} \tag{1.8}$$

$$\Pr(X \leq m) \leq \left(\frac{np}{m}\right)^m e^{m-np} \tag{1.9}$$

$$\Pr(X \geq (1+\epsilon)pn) \leq e^{-\epsilon^2 np/2} \tag{1.10}$$

$$\Pr(X \leq (1-\epsilon)pn) \leq e^{-\epsilon^2 np/3} \tag{1.11}$$

for $0 < \epsilon < 1$.

Below we give a list of notations and terminologies that we will use throughout this dissertation:

- We use "log" to mean the natural logarithm, and "lg" to mean the logarithm with base 2.
- We use O-notation to denote the asymptotic upper bound for the complexity of a given algorithm. In the design of parallel algorithms where there are multiple

processors, we use O-notation to denote the asymptotic upper bound for the *expected* running time of the parallel algorithm.

1.6 Related Work

The idea of utilizing techniques and data structures in data compression to improve performance of algorithms is closely related with the framework of predictive computing model. In some cases, the use of data compression techniques can be classified under the predictive computing model. In those algorithms, the data compression technique is used to aid the prediction engine to predict future data based on previous inputs and the predicted input is used to keep the algorithm running during the delay between arrivals of consecutive input data, thus making the algorithm more efficient. There has been an extensive research in applying the predictive computing model in many different areas such as cache memory management [108, 24] and stock market prediction [5].

Krishnan and Vitter [108, 68, 24] developed an innovative approach to do prediction using data compression techniques and analyzed this approach in detail for the problem of prefetching in database design: a prefetcher uses the idle time between the arrivals of consecutive page fetch requests to analyze the pattern of the previous inputs and prefetches the pages from slow memory such as main memory to fast memory such as cache according to its prediction of future page requests. They have shown the theoretical optimality of this prediction technique for prefetching [108, 24]. In practice, a two-level memory hierarchy is assumed and the model for the user (the source which generates the page request) is a general but fixed Markov source. The specific compressor used is a character-by-character version of the Lempel-Ziv data compressor to achieve optimal fault rate in the limit. In [68] they extend the Markov source model of programs to worst-case sequences and show that the prefetcher does

as well in the limit as the best finite state prefetcher specifically tuned for the sequence. That is, the prefetcher is optimal in the limit for *all* sequences of page requests.

1.7 Contributions

This dissertation dresses the problem of applying well studied data structures and techniques in data compression algorithms to solve problems in other algorithmic domains. Because these data structures and techniques exploit internal statistical properties of input source so that the learned knowledge of the input source can be utilized to reduce computation time , we expect that the new algorithms based on them can achieve better performance than previous algorithms. We concentrate on several fundamental algorithmic problems such as sorting and string matching. We also propose a general framework to extend our study to other domains by identifying the suitable algorithmic problems to which the data compression techniques can be applied and show how to design a new algorithm according to different properties of the inputs of the new problems.

This work makes the following contributions:

Methodology

We present a general framework for using data compression techniques to predictively speed up computation in various algorithmic problems. Specifically, we identify the group of algorithm problems where the performance of the algorithm solving the problem is dependent on the probability distribution of the input source and show how to design new algorithms based on data compression techniques according to the specific statistical properties of the input.

Algorithms

We first study the fundamental problems of sorting and string matching. In the

sorting algorithm, we apply the a random sampling technique using a LZW trie data structure to speed up the indexing of the keys. The algorithm is analyzed in terms of underlying machine model and the complexity of the algorithm is shown to be closely related to the compressibility of the inputs, assuming the inputs are generated from a stationary and ergodic source. In the string matching problem, we achieve an efficient string matching algorithm based on the relationship between compressibility of the inputs and the probability of matching. The algorithm is quite simple in terms of preprocessing and matching. The analysis of the expected running time is shown to be dependent on the compressibility of the inputs.

Also we extend our application to more general data type other than simple binary stream. We present a fast volume rendering algorithm which improves a well known volume rendering algorithm - splatting algorithm by performing computation directly in a compressed transform domain. Furthermore, we design a tree compression algorithm which keeps a tree structure of smaller size in the compressed form. The main techniques used are LZ parsing scheme and Huffman coding. Furthermore, we extend this algorithm to compress undirected graphs.

Implementation

Although the main purpose of the thesis is to illustrate the theoretical application in design and analysis of new algorithms using techniques in data compression area, it is important to test whether the new algorithm performs well in practice. In Chapter 3, we implemented the sorting algorithm on UNIX environment and compared our results with those achieved by system routines. Without major optimization of codes, our new algorithm performs closely to the system routine which has been carefully coded.

1.8 Outline of the Thesis

In this chapter, we presented the background of the thesis and gave a survey of related work. In Chapter 2 we give a comprehensive review of known data structures and techniques in data compression which are widely used throughout this thesis. Chapter 3 gives a sorting algorithm based on several observations of statistical properties of input data. The algorithm utilizes some data structures and techniques in LZW data compression. In addition we discuss an application of the new sorting algorithm to other areas such as computational geometry. In Chapter 4, we extend our work to give fast pattern matching algorithms in both one and two dimensions. We also discuss the parallel versions of the algorithms. In Chapter 5, we propose a novel idea of reducing computational cost of algorithm by performing computation directly in compressed transform domain and apply this idea to design a fast volume rendering algorithm in the area of image processing. Finally. we study an interesting topic of applying the data compression techniques to more complex data types in Chapter 6. The result of this investigation is an optimal algorithm for tree compression. As conclusion, in Chapter 7, we summarize the thesis and examine the possibility of carrying our research into new areas such as non-stationary data source, lossy compression, multi-dimensional data.

Chapter 2

Data Compression: Data Structures and Techniques

Before we present the case study of applications of data compression algorithms, in this chapter we give a survey on some basic concepts, data structures and techniques used in design and analysis of data compression algorithms. We concentrate on the data structures and techniques that are used in our future algorithms. In addition to a general description, statistical properties of certain data structure are discussed in details in order to explain why and how the data structure can be useful in designing other algorithms. In later chapters we will apply these data structures and techniques to specific algorithmic problems.

2.1 Data Compression Algorithms

Data compression algorithms compress the original data into a representation of smaller size and thus reduce the storage space and require less bandwidth to transfer the data. There has been extensive research on data compression algorithms [7, 51, 52, 57, 113, 114].

Data compression algorithm is *lossless* if the encoded data contains all the information in the original data. Otherwise the algorithm is called a *lossy* algorithm. Lossy compression is desirable in many areas where the precision of compression can be sacrificed for faster compression and/or higher compression ratio.

Another way of categorizing the data compression algorithm is whether the compression model assumes that the underlying probability mass function is known before the compression begins. If the compression algorithm is based on a fixed model, it is

called *static* modeling compression. If the compression algorithm learns the underlying probability mass function by sampling from the inputs to adjust the model, it is called *adaptive* compression.

2.1.1 Lempel-Ziv Compression

Lempel-Ziv (LZ) coding [113, 114] is a general class of dictionary compression methods using adaptive schemes. The central idea of LZ coding is that better compression could be obtained by replacing a repeated phrase by a reference to an earlier occurrence. A phrase can be a word, part of a word or several words. After the phrase appears in the text, the later appearances of the phrases can be replaced by a pointer to the first occurrence. Decoding a text that has been compressed by LZ coding is done by simply replacing every pointer by the actual decoded phrase that the pointer is pointing to.

There have been many variations of LZ compressions. The main two factors that govern the design decision of various LZ coding algorithms are whether there is a limit to how far backward a pointer can reach, and what kind of substrings within this limit may be the target of a pointer. For example, the coding scheme may restrict a fixed-size window of N characters that a pointer can reach back, or it may simply allow unrestricted window size. The selection of phrases may as well either be unrestricted, or limited to a set of phrases according to certain heuristics. Each combination of these choices produces some compromise between speed, memory requirements and compression efficiency. A growing size window has the advantage of compressing better by use a larger number of substrings for encoding. However, the time to search for a match of substring takes longer time and the space needed to store all substrings also gets bigger. Using a fixed size window eliminates the time and space constraints but sacrifices compression performance.

The family of LZ compression algorithms are generally derived from one of two different approaches developed and Lempel and Ziv in 1977 and 1978 respectively, labeled LZ77 and LZ78. In LZ77 scheme, pointers denote phrases in a fixed-size window that precedes the coding position.

In this thesis, as far as implementation is concerned, we consider a later variation of LZ compression, known as LZW [82] compression, where the dictionary is represented by a digital search tree. This dictionary is constructed incrementally from a nearly empty dictionary {0,1}, by finding in the text the longest prefix match of the remaining text that matches a word in the dictionary, which corresponds to a current leaf of the tree, and then adding to the dictionary a string consisting of this match concatenated with the next character in the text (i.e., creating a new leaf just below the current matching leaf). More details about LZW compression will be presented in Chapter 3.

LZ compression implementation outperforms other dictionary coding schemes in practice. It is also known and proved in [113, 114, 34] that LZ compression achieves optimal compression, which means that for any large file F_n of size n generated by a stationary and ergodic source, the compression ratio ρ achieved by applying LZ compression limits to $1/H(F_n)$ for $n \to \infty$ where $H(F_n)$ is the entropy of the source from which F_n is generated.

Recall from the introduction chapter the concept of entropy rate H. It is closely related to optimal compression ratio in data compression. To compress a source in an optimal way means that we use minimum number of bits possible to describe the statistical source. Thus the entropy of a stationary ergodic source is equal to the inverse of the optimal compression ratio of the source by definition. Formally, let I be the number of bits of the original inputs and C be the number of bits of the

optimally compressed version of the inputs, we have the following equation:

$$H(F) = \frac{1}{\rho_{opt}} \rightarrow \frac{C}{I} \tag{2.1}$$

when $I \rightarrow \infty$.

Since some dictionary-based data compression algorithms such as LZ compression achieve optimal compression when the size of the input goes to infinity, the entropy of the input source can be estimated by using these compression algorithms. Furthermore, those compression algorithms utilize an adaptive learning method by scanning and parsing the input text, the very same idea can be applied to solve other algorithmic problems where the performance of the algorithm depends on how adaptive it is to learn the underlying probability distribution of the input source.

It is noteworthy that the estimation of the entropy of a fixed input source is a non-trivial task. Since the LZ compression is optimal in the limit, it can certainly be used to estimate the entropy of the input source by approximating the optimal compression ratio. However, the rate of convergence by using LZ compression is very slow due to large number of parses needed to build the dictionary of patterns.

Another method of estimating entropy is to use a sliding window. Choose a positive integer N which will be the size of the "window" we are observing from the input source. Given the sequence $X_1, X_2, ...$, we define for every index i, the longest match of the string of the observation to the right of i, into the string of observations in a window of size N to the left of i. Let $X \not\subseteq Y$ denote that X is not a subset of Y. Formally, let

$$L_i = min\{k : X_{i+1}^{i+k+1} \not\subseteq X_{i-N+1}^{i}\} \tag{2.2}$$

This defines a sequence of random variables $\{L_i\}$. The following $\hat{H}$ can be used

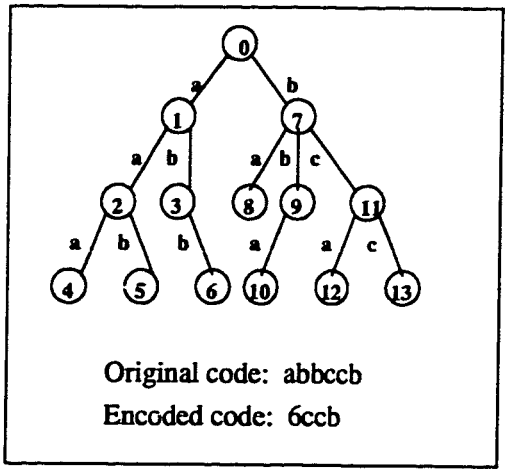

Figure 2.1: Digital search tree (trie).

to estimate the entropy H:

$$\hat{H} = \frac{\log N}{\bar{L}} \tag{2.3}$$

where $\bar{L}$ is the average of L_i. The convergence of this estimation takes place with an error that is $O(1/\log N)$. There is no clear rule to select the correct size of the window. One prudent choice is to let $\log N$ be approximately equal to the standard error of $\bar{L}$.

2.2 Data Structures

The main data structures that we will use throughout this thesis is the *digital search tree* (or *trie*). Another data structure called *suffix tree* is also used widely in design of many algorithms. However, we will refer to suffix tree as a compact representation of trie where each leaf of the suffix tree contains a string of characters while the leaf of the trie only contains one character. Trie and suffix tree share some common statistical properties and therefore we will focus on trie data structure in most part of this thesis.

2.2.1 Digital Search Tree (Trie)

A digital search tree (trie) [64] is a multiway tree with a path from the root to a unique node for each string represented in the tree. Figure 2.1 gives an example of a trie indexing a string. In a trie, only the prefix of each string is stored so a longest match can be found by tracing the tree until no match is found, or the path ends at a leaf.

We start by describing digital search tree T associated with a set of distinct strings $Y_1, Y_2, ..., Y_m$ over the alphabet Σ. Assume that no Y_i is a prefix of some Y_j where $j \neq i$. The digital search tree T is a rooted tree with m leaves such that

1. Each edge of T is labeled from the alphabet Σ, and is directed away from the root.

2. No two edges emanating from the same vertex have the same label.

3. Each leaf u is uniquely identified with a string Y_i, i.e., the concatenation of the labels on the path from the root to u is Y_i.

An efficient implementation of trie is a single hash table [64] which contains an entry for each node of the trie. To determine the location of the child of the node at location n in the table for input character c, the hash function is supplied with both n and c. The hash function which maps the prefix string to a position of the hash table is carefully designed so that the cost of each query search takes only constant time.

The digital search tree T we just described has the problem that its size can be very large. For example, the digital search tree corresponding to the sting $X = a^n b^n \#$ needs $\Omega(n^2)$ vertices. This is clearly undesirable. No string of length n will require more than $O(n^2)$ vertices since the length of the path from the root to an arbitrary leaf is $O(n)$ and there are n leaves. It is desirable to find an alternative data structure to digital search tree which avoids this problem.

2.2.2 Suffix Tree

The *suffix tree* associated with sequence X, denoted for the remainder of this thesis by T_X, is a compact version of the digital search tree T. Formally, T_X is a digital search tree with n leaves and no nodes of outdegree 1 which satisfies the following conditions:

1. Each edge is labeled with a substring of X;

2. No two edges emanating from a vertex are labeled with strings having a common prefix;

3. Each leaf u is associated with exactly one suffix; that is, the concatenation of the labels on the path from the root to u is equal to the suffix.

Note that the suffix tree T_X associated with a string X of length n has at most $2n - 1$ vertices, since the number of internal vertices is bounded by the number of leaves.

Suffix trees have a wide variety of applications in many algorithmic problems mostly involving string processing. The well known examples include string matching, approximate string matching, string comparison, data compression schemes, implementation of Lempel-Ziv adaptive compression, and even genetic sequences and biologically significant motif patterns in DNA. Solution to many of those problems depend on the following problem: given a sequence X (or two sequences X and Y, and two arbitrary suffixes S_1 and S_2 of X (or of X and Y respectively), find the longest common prefix of S_1 and S_2. Suffix trees have been proven to be the most efficient data structure to answer this type of queries.

We now give a brief summary of some applications of suffix trees. Similar techniques to use suffix trees as fundamental data structure to solve algorithmic problem are also used throughout this thesis. Given an input source $\mathcal{X}$, a suffix tree is constructed by scanning the input text. We consider the case when the suffix tree is

built from a random source. The algorithms are parallel algorithms that use a linear number of processors and are based on CRCW PRAM. The followings are some applications of the suffix tree:

On-line string matching. Given a pattern string Y of length m and a text string X of length n, the problem is to answer the query whether string Y contains string X. We can construct a suffix tree T_x associated with X and the query can be answered by the use of suffix tree in $O(\log m)$ time whereas the preprocessing of text string X takes $O(\log n)$ time with a total of $O(n)$ operations.

Longest repeated substring. The problem is to find the longest substring of X that occurs more than once. By combining the point-jumping technique in parallel algorithm, the longest repeated substring can be found in $O(\log \log n)$ time using a linear number of operations.

Substring identifiers. Given i, find the shortest substring u of X that identifies position of X; i.e., u occurs in X at location i but u does not occur anywhere else in X. Using suffix tree T_X, the preprocessing can be done in $O(\log n)$ time and the query can be answered in $O(1)$ time.

In analysis of the complexity of algorithms that are based on suffix trees. some statistical properties of general suffix trees are of fundamental use. Szpankowski [105] presented a complete characterization of suffix tree in a general probabilistic setting assuming independencies on words. Some of his results are critical in analysis of algorithms using suffix tree as an efficient way of performing operations such as string matching and longest common prefix. Here we give a brief review on some of his results which are related to this thesis.

In [105], the suffix tree is assumed to be constructed by words which are generated by a stationary and ergodic source. A list of results concerning important parameters of a suffix tree is presented, namely the *typical depth* M_n, *depth of insertion* L_n, *height*

H_n. For example, the depth of insertion L_n is of prime interest in design and analysis of Lempel-Ziv universal compression schemes as well as in many other algorithms on words. Furthermore, the height of a suffix tree is a good indication about how balance the suffix tree is and thus it is related to the *worst-case* complexity analysis. Note that the length or height considered here includes the length of suffices contained in the leaves.

Let $\{X_k\}_{k=-\infty}^{\infty}$ be a stationary and ergodic sequence of symbols generated from a finite alphabet $\mathcal{A}$. Let $X_m^n = (X_m, ..., X_n)$ be a partial sequence where $m < n$. Furthermore, for every $n \geq 1$ let the nth-order probability distribution of $\{X_k\}$ be

$$P(X_1^n) = \Pr\{X_k = x_k, 1 \leq k \leq n, x_k \in \mathcal{A}\} \tag{2.4}$$

Also let the *entropy* of $\{X_k\}$ be

$$h = \lim_{n\to\infty} \frac{E \log P^{-1}(X_1^n)}{n}. \tag{2.5}$$

The existence of the above limit can be proved by using Kolmogorov-Sinai Theorem [9]. To explain the results, we need to define another two parameters of $\{X_k\}$: h_1 and h_2 where

$$h_1 = \lim_{n\to\infty} \frac{\log(1/\min\{P(X_1^n), P(X_1^n) > 0\})}{n}; h_2 = \frac{\log(\sum_{X_1^n} P^2(X_1^n))^{-1}}{2n} \tag{2.6}$$

Pittel [84] established the existence of these two parameters and also noticed that $0 \leq h_2 \leq h \leq h_1$.

Two forms of conditions, the *mixing* condition and the *strong α mixing* condition, are needed for the analysis of the behavior of the suffix tree. A sequence $\{X_k\}_{k=-\infty}^{\infty}$ is said to satisfy the mixing condition if there exist two constants $c_1 \leq c_2$ and integer d such that for all $1 \leq m \leq m + d \leq n$ the following holds:

$$c_1 \Pr\{\mathcal{B}\} \Pr\{\mathcal{C}\} \leq \Pr\{\mathcal{BC}\} \leq c_2 \Pr\{\mathcal{B}\} \Pr\{\mathcal{C}\} \tag{2.7}$$

where $\mathcal{B} \in \mathcal{F}_{-\infty}^{m}$ and $\mathcal{C} \in \mathcal{F}_{m+d}^{n}$.

It is known that this condition implies ergodicity of the sequence $\{X_k\}_{k=-\infty}^{\infty}$ and therefore the mixing condition is a stronger condition than the ergodicity requirement of the source.

The sequence $\{X_k\}_{k=-\infty}^{\infty}$ satisfies the strong α mixing condition if the following holds:

$$(1-\alpha(d))\Pr\{\mathcal{B}\}\Pr\{\mathcal{C}\} \leq \Pr\{\mathcal{BC}\} \leq (1-\alpha(d))\Pr\{\mathcal{B}\}\Pr\{\mathcal{C}\} \tag{2.8}$$

where $\mathcal{B} \in \mathcal{F}_{-\infty}^{m}$ and $\mathcal{C} \in \mathcal{F}_{m+d}^{n}$ and $\alpha(d) \to 0$ as $d \to \infty$. The following results deal with the typical depth M_n, the depth of insertion L_n and the typical height H_n where n is the number of symbols used in suffix tree construction:

Theorem 2.1 *(i) Let weak mixing condition 2.7 hold. Let h denote the entropy of the input source. Then,*

$$\lim_{n\to\infty} \frac{L_n}{\log n} = \lim_{n\to\infty} \frac{M_n}{\log n} = \frac{1}{h} \tag{2.9}$$

(ii) Let strong α mixing condition 2.8 hold together with $h_1 < \infty$ and $h_2 > 0$. Then,

$$\lim_{n\to\infty} \frac{M_n}{\log n} = \frac{1}{n} \tag{2.10}$$

almost surely and

$$\lim_{n\to\infty} \inf_{n} \frac{L_n}{\log n} = \frac{1}{h_1} \lim_{n\to\infty} \sup_{n} \frac{L_n}{\log n} = \frac{1}{h_2} \tag{2.11}$$

for all stationary and ergodic sequences $\{X_k\}_{k=-\infty}^{\infty}$ provided that

$$\alpha(n) = O(n^{\beta}\rho^{n}) \tag{2.12}$$

for some constants $0 < \rho < 1$ *and* β. *(iii) For large* n *the height* H_n *satisfies*

$$\lim_{n\to\infty} \frac{H_n}{\log n} = \frac{1}{h_2} \tag{2.13}$$

almost surely provided the coefficients $\alpha(d)$ *in 2.8 satisfies*

$$\sum_{d=0}^{\infty} \alpha^2(d) < \infty. \tag{2.14}$$

where h_1 *and* h_2 *are defined in (2.6).* □

Pittel [84] also considered a typical behavior of a trie constructed from independent words and achieved similar results as above. It is not surprising that the trie constructed from independent words and the suffix tree constructed from a continuous string demonstrated similar behavior because the former data structure is just a subset of the latter which imposes dependencies among phrases in the string.

In this thesis, the trie data structure is used in design of novel algorithms based on data compression techniques. Note that a suffix tree considered above is only a compact representation of a trie. The above statistical properties of a certain suffix tree hold for its corresponding trie as well. Therefore in the analysis and theorem proof in our algorithms, we use the above results to trie directly. The advantage of using trie instead of suffix tree is that the trie is a more natural representation of characters in the string with each node in the trie of equal size. Furthermore, the trie is easier to implement in practice than the corresponding suffix tree. From now on, we will focus on trie as our main data structure while applying the above results in algorithm analysis without specific distinction between trie and suffix tree.

In design and analysis of algorithms using trie, it is often assumed that the input source is stationary and ergodic while satisfying the mixing conditions. The above results on statistical properties of typical behavior of trie constructed from the input

source followed from these assumptions provide a strong tool in analysis of expected time complexity of algorithms.

2.3 Techniques

In this section, we review some important techniques in data compression algorithms that will be used throughout this thesis. By no means this survey covers all the techniques in data compression algorithms. We only present the standard techniques that will be crucial in design of our algorithms. Note that some of the techniques are introduced by specific data compression algorithm while others are shared by many algorithms. We are mainly concerned with the techniques used in LZ compression scheme since it is most widely used for practical compression. Also we do not cover full details on the implementation of these algorithms. The details can be referred to standard text books on data compression algorithms [7, 35]. Without loss of generality, we assume that the source alphabet is binary.

2.3.1 Distinct parsing

The LZ compression algorithms divide input string into phrases by using *distinct parsing* technique, i.e. a division of string into phrases so that no two phrases are identical. For example, $0, 1, 11, 111$ is a distince parsing of 0111111, but $0, 1, 1, 11, 11$ is a parsing that is not distinct. The distinct parsing has statistical properties that are crucial in proving the optimality of compression for LZ compression schemes. Let $c(n)$ denote the number of phrases in the distinct parsing of a sequence of length n. The following lemma presents an upper bound on $c(n)$:

Lemma 2.1 *The number of phrases $c(n)$ in a distinct parsing of a binary sequence $X_1, X_2, ..., X_n$ satisfies*

$$c(n) \leq \frac{n}{(1-\epsilon_n)\log n}$$

where $\epsilon_n \to 0$ as $n \to \infty$.

This result is used to bound the compression ratio achieved by such distinct parsing in the proof of optimality of LZ compression [34]. The specific distinct parsing used by a later version of LZ compression, the LZW compression, is a straightforward division of the input string by longest prefix matching (see description of dictionary encoding below). The distinct parsing technique may find suitable in algorithms to analyze properties of a long string by breaking the string into phrases.

2.3.2 Dictionary encoding

Dictionary encoding is a standard technique used in many dictionary based compression algorithms. The basic idea is to replace the occurrences of frequent phrase in the input data by the index of this phrase in the dictionary thus saving storage space. The dictionary encoding will be most efficient if it assigns shorter codes to the phrases that appear more often and assigns longer codes to the ones that appear less. However, it is unlikely that the probability distribution of the input source is known before compression. Therefore, an *adaptive* dictionary encoding normally outperforms a static dictionary encoding because it adapts to the probability distributions of the phrases by monitoring the frequencies of appearances of different phrases.

A variation of LZ compression, the LZW compression, compresses the input text by indexing pieces of the text to an adaptively constructed dictionary. The dictionary is implemented by a digital search tree structure. Without loss of generality, we consider the case where the input text is a binary stream. Initially the dictionary contains only two unit length phrases 0 and 1. The algorithm repeatedly finds the

longest prefix matching of the remaining input text in the dictionary, that is, the longest prefix of the remaining text that is already stored in the dictionary. Let x be this longest prefix match. The algorithm then combine x with the next character in the remaining text to form a new phrase x'. The new phrase is inserted into the dictionary. And in the compressed text the original phrase will now be replaced by its index in the dictionary thus achieving compression.

2.3.3 Prefix string matching

The LZW compression algorithm divides the input string into phrases by repeatedly finding the longest prefix match between the remaining input string and the current phrases in the dictionary. Besides being used in distinct parsing, prefix string matching can be used in partial prefix matching (PPM) compression algorithms, general string matching algorithms and other string processing algorithms.

The performance of the prefix string matching largely determines the efficiency of the parsing. A straightforward implementation of prefix string matching procedure will add one character at a time to the prefix of the remaining input string and search in the dictionary to see whether the prefix has already been stored in the dictionary. However, this method is not efficient to search for long prefix matches. An alternative implementation uses a hash table to stored the indices of the dictionary and a well chosen hash function that computes the hash value given any prefix of the remaining input string. We illustrate the details of this method of prefix matching in next chapter in the context of dictionary construction.

2.3.4 Huffman coding

LZ compression approximates and learns the probability distribution of the input source by adaptively constructing a dictionary. However, if the probability distribu-

tion of the input source is known before compression, an optimal assignment of codes to phrases can be computed using the *Huffman coding* scheme [26].

Huffman coding is greedy in the sense that it makes a locally optimal choice in the hope that this choice will lead to a global solution for the problem. Nonetheless, Huffman coding is extremely efficient in practice, normally achieving a compression ratio from 5 to 10.

Huffman coding uses a variable length code that assigns to different phrases with codes of different length. Also the codes are all *prefix codes* where no codewords is a prefix of some other codewords. It can be shown that any optimal data compression achievable by a character code can always be achieved by prefix codes. Using a binary tree with each node representing different phrases, the coding works in a bottom-upper manner and merges two nodes with smallest probabilities together at any single step. The result is an optimal coding assignments that assigns short code to frequent phrase and long code to infrequent phrase.

2.4 Summary

This chapter presented data structures and techniques in data compression algorithm design that are used throughout this thesis. We concentrated on the LZ compression scheme and showed some important statistical properties of the suffix tree data structure. We also gave a general review of data compression techniques for both adaptive and static algorithms. In the remaining of this thesis, we will demonstrate that we can use these data structures and techniques to aid many aspects of the design and analysis of algorithms in other areas. In next chapter we will began with the first case study of the sorting problem.

Chapter 3

Fast Sort for Entropy Bounded Text

Sorting is one of the most heavily studied problems in computer science. Given a set of n keys, the problem of **sorting** is to rearrange this sequence either in ascending order or descending order. There has been extensive research in sorting algorithms, both in sequential and parallel settings (see next section for detail).

Sorting is of great practical importance in scientific computation and data processing. It has been estimated that twenty percent of the total computing work on mainframes is sorting. Therefore, even an improvement of a constant factor will have a large impact in practice. Though the theoretical research has already had large impact, there are still fundamental problems remaining. For example, in the study of sequential sorting algorithms, the *comparison tree* model assumes that the only operation allowed is comparison and it is well known that the *comparison sort* (i.e. merge sort and heapsort [26]) based on comparison tree model has a lower bound of $\Omega(n \log n)$ for sorting n elements. However, this bound can be relaxed by allowing operations other than comparison to achieve time bound less than $O(n \log n)$. For example, radix sort and bucket sort which are not based on comparison tree model have a linear running time assuming that the input keys are drawn from uniform distribution.

As parallel algorithms now play a bigger and bigger role in algorithm implementations, proposed parallel sorting algorithms must be capable of being optimally sped up (so the product of time and processor equals the total work of optimal sequential algorithm to parallel machines). This is essential in implementations on real machines since most high performance machines have parallelism. For example, the powerful

Sorting algorithms	*Assumptions on inputs*	*Running time*
Counting sort	integers in range $[1, k]$	$O(k)$
Radix sort	bounded length d of key	$O(nd)$
Bucket sort	random distribution	$O(n)$
Our sort	bounded entropy H	$O(n \log(\frac{\log n}{H}))$
Andersson's sort	bounded length B of distinguishing prefix	$\Theta(n \log(\frac{B}{n \log n} + 2))$

Table 3.1: Sorting algorithms not based on comparison-tree model.

CRAY computer can be viewed as a large vector machine.

Another important aspect in designing fast sorting algorithms is to explore the input data so that the sorting algorithm can take advantage of input data satisfying certain properties (e.g. certain probability distribution). This type of algorithms includes bucket sort and radix sort, with different assumptions on the input data. In this chapter, we design a sorting algorithm based on a specific statistical property, namely the bounded entropy, which is true for most practical files. We give both sequential and parallel versions of the algorithm. In order to achieve high performance in practice, we use the digital search tree (trie) data structure used in data compression algorithm to speed up operations on binary strings. We present a sorting algorithm that runs in expected running time $O(nlog(logn/H))$. We then give a parallel version of our algorithm which runs in $O(log(logn/H))$ parallel time using $O(n)$ processors. We also give several applications based on the sorting algorithm such as the priority queue problem and convex hull problem.

3.1 Previous Work and Our Approach

Before we present our sorting algorithm, we first summarize previous approaches for sorting problems and explain why we choose various theoretical assumptions for our algorithm design.

Computational Model. The *Comparison tree* model assumes that the only operation allowed in sorting is comparison between keys to gain order information [26]. That is, given two keys x_i and x_j, we perform one of the following tests $x_i < x_j$, $x_i \leq x_j$, $x_i = x_j$, $x_i \geq x_j$, or $x_i > x_j$ to determine their order. Other ways such as inspecting the values of the keys are not allowed to tell the order. Though an excellent model for theoretical analysis, the comparison model has a drawback that it requires a $\Omega(\log n)$ lower bound per key for sorting even with the assumption that keys are uniformly distributed. Many sorting algorithms are based on comparison tree model and achieve this optimal time bounds (e.g. quicksort, mergesort [26]).

On the other hand, there have been sorting algorithms (bucket sort, radix sort, etc) proposed which are not based on comparison trees (see Table 3.1 for a list of sorting algorithms which are not based on comparison tree model). For example, many algorithms instead adopt a more general *unit cost RAM* model which assumes that the usual operations such as addition, shift, multiplication, bit comparison are regarded as one single step. The unit cost RAM is a more realistic model for the actual machine architecture and thus has the advantage to be used to design algorithms which do not have the striction of $O(\log n)$ lower time bound.

Two well known examples are counting sort, radix sort and bucket sort. Counting sort assumes the input consists of small integers to achieve a linear running time. The radix sort makes the assumption that every input key is a d-digit integer where d is a constant and also runs in linear time. The bucket sort runs in linear expected time because it assumes that the input is generated by a random process that distributes elements uniformly over the interval $[0, 1)$.

In our sorting algorithm, we assume the unit cost RAM as our computational model with the general assumption that the input keys are drawn from a bounded-entropy stationary and ergodic source.

Randomized vs. Determininistic Algorithms. Randomized sorts, such as quicksort which were initially considered only of theoretic interest, are used on many if not most system sorting routines for large inputs. In general, randomized algorithms (sequential or parallel) are simpler and easier to implement than the deterministic algorithms which achieve the same time complexity, though the time complexity for the algorithm is with high probability in the case of randomized algorithms. The parallel variants of randomized algorithms, such as FLASHSORT [95] and SAMPLESORT[59], have given some of the best implementations of parallel sorting (see [60] and [11] respectively). Therefore, considering the actual performance of the algorithm, it is reasonable to adopt the randomized method for parallel version of our algorithm.

Assumptions on Input Keys (i) *Maximum number of bits per key.* On the unit cost RAM, the input keys of sorting algorithms can be regarded as a sequence of binary numbers with different length (the keys can be easily converted to binary representation if it is not the case). Some sorting algorithms, such as the algorithms based on data structure of digital search tree (trie), and Radix Sort, assume that the number of bits per key is bounded by L. Radix Sort [64] requires $O(L/\log n)$ sequential work for sorting each key. The sorting algorithms using trie approach [10, 64] requires $O(\log L)$ sequential work per key. However, the assumption of bounded maximum length does not seem to hold widely in practice: in fact, many major users of sorting routines, e.g., data base joins, require moderately large key size L. In our algorithm, we does not impose an upper bound on the maximum length of input keys. In other words, the

running of our algorithm is independent of the maximum length of the input keys. Instead, since we use prefix matching to find the order of the input keys, the complexity of the algorithms depends on the length of the first prefix match between two given keys.

(ii) *Random input distribution vs. worst-case analysis.* Another commonly used assumption in sorting algorithm is to analyze the performance of the sorting algorithm based on certain distribution of the input, e.g. random distribution or worst case input. The apparent advantage of the worst-case input assumption is that the sorting time bound can be guaranteed even in the worst case. As we illustrated in the introduction (see Chapter 1), in practice the probability of sorting a worse case input is very small. On the other hand, if we assume random distribution of the inputs, in the bucket sorting algorithm [64], sorting each key needs only $O(1)$ expected time. But the random input assumption also does not hold very frequently in practice as well. Therefore, we do not make assumption on the probability distribution of the inputs, nor do we base our algorithm analysis on the worst case inputs. Instead, we only assume that the input keys are drawn from a stationary and ergodic source with the underlying probability unknown before running the algorithm.

(iii) *Entropy bounded input sets.* As computer scientists, we would like to design algorithms which are applicable to files found in practice. Therefore it is a good question to ask what general properties of the practical inputs we can exploit to speed up the sorting algorithm. There have been various sorting algorithms which are not based on comparison-tree model and make different assumptions about the input source. For example, Manilla [75] gave an optimal sorting algorithm by measuring presortedness of input. As discussed in the introduction of this thesis, for large files occurring in practical applications, the

lossless compression ratio is often at least 1.5 and no more than a relatively small constant, say 20 (see definition of entropy and compression ratio in Chapter 1). It is not reasonable to assume uniformly distributed sets of keys for practical files (otherwise, compression ratio would almost always be 1, which is not the case), so bucket sort is not appropriate or at least has limited applicability for sorting large files in practice. Since the performance of our sorting algorithm depends on the compressibility of the input keys, we will assume that the source from which the input keys are generated has a bounded compressibility.

Based on the above observations, we choose assumptions carefully for our new sorting algorithm to suit the purpose of designing a practical sorting algorithm. We assume the randomized unit cost RAM as the computational model. In order for the algorithm to have general applications (e.g., database join applications). we assume no upper bound for maximum number of bits that each key can have. Since we utilize data structures and techniques used in data compression algorithms, the time complexity of our algorithm depends on the compressibility of the input source. We assume that the compressibility is bounded which is true for practical inputs so that the algorithm runs well in practice.

3.1.1 Computational Model

Recall from the introduction that the unit cost RAM model we use allows the basic operations such as addition, shift, and multiplication to be accomplished in one single step. We regard the input keys as strings of binary bits. Our analysis depends on the total number of keys we are processing instead of the total number of bits the input has. Therefore, we assume no bound on the maximum length of a given input key. The actual machine memory layout of the key is irrelevant to the discussion of the algorithm. Specifically, we make the following assumption to the computational

model. Any input key x, represented by a binary string can be regarded as a binary number representing an integer. In other words, the input keys can also be viewed as a sequence of integers with no upper bound. This assumption is needed to insure that operation on trie data structure, such as finding the longest prefix match in the trie for a new key, can be accomplished efficiently. We will describe in detail such operations in next section.

3.1.2 Statistical Properties of Input Keys

We address the problem of relating the complexity of sorting to some statistical properties of the input keys such as optimal compression ratio and entropy. Suppose the input source from which the input keys $X_1, X_2, X_3, ..., X_n$ are generated is a stationary and ergodic sequence built over a binary alphabet {0,1}. Recall from the introduction that informally, "ergodicity" means that as any sequence produced in accordance with a statistical source grows longer enough, it becomes entirely representative of the entire model. On the other hand, "stationarity" means the underlying probability mass function of the input source is not changing over time

Let $\{X_i, i = 1...n\}$ be the set of input keys which are generated by a stationary source with the *mixing condition.* Recall from the introduction that a sequence $\{X_k\}_1^\infty$ is said to satisfy the mixing condition if there exists two constants $c_1 \leq c_2$ and integer d such that for all $1 \leq m \leq m + d \leq n$ the following holds:

$$c_1 \Pr\{\mathcal{B}\} \Pr\{\mathcal{C}\} \leq \Pr\{\mathcal{BC}\} \leq q c_2 \Pr\{\mathcal{B}\} \Pr\{\mathcal{C}\} \tag{3.1}$$

where $\mathcal{B} \in \mathcal{F}_{-\infty}^m$ and $\mathcal{C} \in \mathcal{F}_{m+d}^n$. This mixing condition implies ergodicity of the sequence $\{X_k\}_{-\infty}^\infty$.

We now define F_n to be the input file of keys $X_1, X_2, ..., X_n$ concatenated together. In our algorithm, we only need to consider the first prefix match of any given key to the dictionary. However, the definition of compression ratio is consistent with the

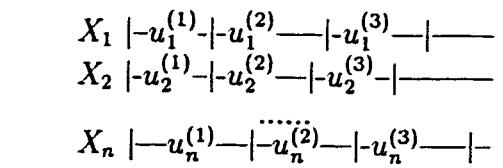

Figure 3.1: Prefix matches of input keys.

compression ratio that we achieve when we consider compressing the concatenated file F_n, since F_n can also be viewed as generated by the same statistical source. In Figure 3.1 below, X_i denotes the ith input key with arbitrary length and u_i^j denotes the jth prefix match of the ith key achieved by matching against the LZ dictionary (see section below for details). In our sorting algorithm, only the first prefix match u_i^1 will be involved in the analysis of the algorithm.

Let $\bar{L}$ be the average length of the first prefix match, i.e., $\bar{L} = E(u_i^{(1)})$. The optimal LZ code for n keys has $\log n$ bits [104]. Thus, the expected optimal compression ratio when we consider the sequence of concatenating $u_1^{(1)}, u_2^{(1)}, u_3^{(1)}...$ is $\frac{\bar{L}}{\log n}$ as $n \to \infty$ [105]. However, as the whole input is generated from a stationary and ergodic source, this compression ratio also limits to $\frac{\bar{L}}{\log n}$ when we consider compressing the input file F_n. Let H denote the entropy of the input source. We have the following lemma:

Lemma 3.1 *For large n as $n \to \infty$, $H = Entropy(u_i^{(1)}) = Entropy(F_n)$ for each i where F_n is the concatenation of n input keys.*

3.1.3 Random-Sampling Techniques

In our algorithm, a well known technique of randomization is used to make the algorithm run efficiently in practice. Randomization has been successfully used in a large number of applications and has recently been used to obtain efficient algorithms in sorting. For example, sample sort [26] is a sorting algorithm using random sampling techniques to adapt well to inputs with different distributions.

The general approach taken by random sampling algorithms is as follows. We randomly choose a subset R of the input set S to partition the problem into smaller ones. The partitioned sub-problems can be proven to have a bounded size with high likelihood. We then solve the sub-problems recursively using known techniques and form the solution of the whole problem based on the solutions of the sub-problems. Clarkson [21] has proved that for a wide class of problems in algorithm design, the *expected* size of each subproblem is $O(|S|/|R|)$ and more over the *expected* total size of the subproblem is $O(|S|)$. Clarkson's results show that by using a straight-forward random sampling technique any randomly chosen subset is within the expected size with constant probability, implying that it may be not with constant probability. Consequently, his methods yields *expected* resource bounds but cannot be used to obtain high-likelihood bounds (i.e. the bounds that hold with probability greater than $1 - n^c$ for any $c > 0$). This makes it very difficult to extend his methods to the context of parallel algorithms.

Later a random sampling technique called *over-sampling* is introduced which chooses $s|R|$ number of *splitters* and then partitions the set into $|R|$ groups separating by the first key, the s-th key, the $2s$-th key and so on. The number s is called *oversampling ratio*. This technique overcomes the shortcomings of Clarkson's method. Reif and Sen [100] also proposed a random sampling technique called *polling* to obtain "good" samples with high probability with small overhead.

These random sampling techniques are especially useful for designing parallel algorithms where the primary goal is to distribute the work evenly across the processors. There has been extensive research on implementing various sorting algorithms on parallel architectures [96, 95, 54]. Also various parallel sorting algorithms have been implemented on real parallel machine architectures. The random sampling techniques are used in parallel sorting to distribute the input keys to processors according

to selected buckets where the size of each bucket is bounded within a constant of expected size with high likelihood. The main advantage is that the *load balancing* between processors can be achieved and the parallel running time which depends on the longest running time of processor can also be bounded. Reif and Sen [100] used random sampling techniques to solve many parallel computation geometry problems (e.g. Voronoi diagram, trapezoidal decomposition) where the problems are divided into subproblems by random sampling and the size of the subproblem is crucial to the parallel running time of the algorithm.

Another area in which the random sampling techniques are found useful is designing efficient algorithms whose performance depends on the distribution of the inputs. In these algorithms, the inputs are normally randomly subsampled in order to learn the underlying probability distribution of the inputs. Our sorting algorithm uses random sampling mainly as a tool to learn the unknown probability distribution of the input source and use this information to aid our computation.

3.2 Sequential Sorting Algorithm

Our sequential sorting algorithm consists of two phases. In the first phase, we use randomized sampling technique to learn an approximation of the probability distribution of inputs. A digital search tree (trie) is constructed using the sampled keys in a way similar to the dictionary construction in LZ compression. The trie is used to reflex an approximation of the actual probability distribution of the input keys. The trie is also used to partition the inputs into separate buckets.

In the second phase, the input keys are indexed into buckets based on its prefix match in the trie. We use a standard $O(n \log n)$ time sorting algorithm (e.g. quicksort) to sort each bucket separately and link buckets together using trie structure.

3.2.1 Trie Data Structure

The basic idea of our sorting algorithm is to divide the input keys almost evenly into partitioned buckets. We use a method similar to the construction of a *dictionary* in LZ compression to index the input keys into buckets. It differs from the original dictionary construction in the way that the main purpose for our algorithm is approximating the unknown distribution of the input keys using the dictionary instead of actually compressing the inputs.

We now show how to build the dictionary D using a *trie* data structure (see Chapter 2 for a detailed presentation on trie). A trivial implementation of the trie is to use a simple binary tree whose nodes correspond to phrases of the dictionary. However, the binary tree method is inefficient because the comparison between an input key and the phrases in the tree is done only on a bit-by-bit basis, taking $O(L)$ time to find the maximum prefix match between the key and trie where L is the length of the maximum prefix match.

In our algorithm, we adopt the *hash table* implementation of the trie where comparisons can be done more efficiently than the simple binary tree implementation. Each phrase in the trie is now stored in a certain slot of the hash table where the slot number is determined by a hash function. For the purpose of a clear presentation, we use the terminologies dictionary (in the context of data compression), trie (the data structure) and hash table (the actual implementation) interchangeably for the same data structure in the following description of the algorithm.

We now show how to find the longest prefix match for each input key. The running time analysis is based on our assumption of the computational model. Specifically, we assume that any given key can be viewed as an integer and comparison of integers and hash function computation can all be accomplished in constant time. Let D_s be the current dictionary we have constructed. For any new input key x, let $Index(x)$

be the longest prefix of x that is already stored in the dictionary and d_x is the length of $Index(x)$. We can find $Index(x)$ in $O(\log d)$ time by searching in the hash table where d_i is the length of $Index(x_i)$ (see [10]). The basic idea of the search is that $Index(x_i)$ can be identified by doubling the length of prefix of x for comparison each time (similar to binary search). Without loss of generality, we assume $d = 2^k$ (k is an intger) be the length of the maximum prefix match of x. We first search in hash table to check whether the prefix consists of only the first character of x is already stored stored in the trie. This is done by computing the hash function $h(x, 1)$ to find the index number in the hash table where this character belongs. If we find that the first character is already stored in the hash table, we double the length of the prefix under inspection to check the prefix which consists of the first two characters of x_i and so on. When we find that the prefix of length 2^{k+1} of x is not in the trie, we reverse the order of search using the same doubling (or sometimes halving) method to locate the longest prefix of x which is already in the trie. Since the query of whether a given prefix is in the hash table can be answered in constant time, the total search time depends on the length of the first prefix match. Specifically, we find the longest prefix match of input key x in $O(\log d)$ time where d is the length of the longest prefix match.

3.2.2 Description of the Algorithm

We are now ready to present the sequential sorting algorithm. We assume the binary strings representing input keys are generated by the same stationary and ergodic source whose probability distribution is initially unknown. Our algorithm consists of two phases. In the first phase, the unknown distribution of the inputs is learned by subsampling from the input and a dictionary is built using a trie structure from the sampled keys. In the second phase, all input keys are indexed to buckets associated with the leaves of the dictionary and each bucket is sorted independently and then

concatenated together to produce the output. A probabilistic analysis (Lemma 3.1) shows that the size of the largest bucket can be bounded by $O(\log n/H)$ with high likelihood (by high likelihood, we mean with probability $\geq 1-1/n^{\Omega(1)}$). We also show that the sequential time complexity of this algorithm is expected to be $O(n \log\log n)$, given that the input keys are not highly compressible, i.e., the compression ratio $\rho \leq (\log n)^{O(1)}$.

We present a randomized algorithm for sorting n keys $X_1, X_2, ..., X_n$. In the first phase, the distribution of the input keys is learned by building a data structure similar to a dictionary used in LZ compression.

Random Sampling

First we randomly choose a set S of $m = n/\log n$ sample keys from the input keys and arrange them in random order. We will apply a modification of the usual LZ compression algorithm to construct a dictionary D_s from the first prefixes of the sample keys in S as described just below. The leaves of D_s represent the set of points which partition the range of inputs into separate buckets according to the prefixes of input keys. Once D_s is constructed (and implemented by an efficient hash table [26]), we consider each key X_i for $1 \leq i \leq n$ and index it to the corresponding bucket which is represented by leaves in the trie. Let $X_i = u_i^{(1)}u_i^{(2)}...$ be the LZ prefix decomposition of X_i using the dictionary D_s. Let $u_i^{(1)}$ be the longest prefix of X_i that is already stored in the dictionary D_s.

Dictionary Construction

We now describe the construction of dictionary D_s. It is similar to original LZ algorithm except that we do not actually compress the input key. D_s will be incrementally constructed from an initially nearly empty set $\{0, 1\}$ by finding the first prefix match

$u_{s_i}^{(1)}$ of each sampled key X_{s_i} (considered at random order) in the current constructed dictionary, and then adding the concatenation of $u_i^{(1)}$ with first bit of $u_i^{(2)}$ to the dictionary. This is different from the standard LZ scheme as we stop processing this key now and go on to the next key. Each step takes time $O(\log |u_{s_i}^{(1)}|) = O(\log d_{s_i})$. The main purpose of this dictionary is to form an approximation of the probability distribution of the input source so that we can partition the range of input keys into buckets of nearly even size.

Indexing of Keys

Let a *leaf* of D_s be an phrase in D_s which is not a strict prefix of any other phrases in D_s. Let LD_s be the set of leaves in D_s (these are the leaves of the suffix tree for D_s had it been explicitly constructed). Let $Index(x_i)$ be the maximal element of LD_s which is lexicographically less than x_i. Let d_i =length of $Index(x_i)$. Both $u_i^{(1)}$ and $Index(x_i)$ can be found in $O(\log d_i)$ sequential time by using trie search techniques.

Sorting

After all keys have been indexed to the corresponding bucket, the buckets are sorted separately using a standard sorting algorithm (e.g. quicksort) and then chained together according to the order of sample keys.

The sequential sorting algorithm is described as follows:

Sequential Sorting Algorithm

Input n keys $X_1, X_2, ..., X_n$, generated by a stationary ergodic source.
Output sorted n keys.

Step 1: Random sampling
Randomly choose a set S of $m = n/(\log n)$ sample elements from n input elements.
Step 2: Building buckets

2.1 Randomly permute $S = \{X_{s_1}, X_{s_2}, ..., X_{s_m}\}$.
2.2 Construct the dictionary D_s as described.
2.3 For each leaf w_j of D_s (recall a leaf is an element of D_s which is not a prefix of any other element of D_s), Bucket$[w_j] \leftarrow \emptyset$.

Step 3: Indexing keys

Indexing using D_s.
for each key $X_i, i = 1, ..., n$ do
3.1 Use trie search to find the $Index(X_i)$ (where index is defined above).
3.2 Insert X_i into the corresponding Bucket$[Index(X_i)]$ associated with the index of X_i.

Step 4: Sorting

4.1 Sort leaves of D_s using a standard sorting algorithm (e.g. Quicksort).
4.2 Sort elements within each bucket using standard sorting algorithm (e.g. Quicksort).
4.3 All buckets are linked under the order of the sorted leaves.

3.2.3 Analysis of Complexity

To bound the maximum size of the buckets, we have the following lemma:

Lemma 3.2 *The size of each resulting bucket is* $O(\log n/H)$ *with probability* $\geq 1 - \frac{1}{n^{\Omega(1)}}$ *for large* n *where* H *is the entropy of the input source.*

Proof: Recall that LD_s is defined to be the set of leaves of dictionary D_s. In [104] it is shown that there exist c, c' where c and c' are two positive numbers such that any leaf $y \in LD_s$ has probability between $c/|LD_s|$ and $c'/|LD_s|$ when $|D_s| \rightarrow \infty$.

This result of [104] implies that for each key X_i $(1 \leq i \leq n)$ with the same probability distribution, for each leaf $y \in LD_s$, there exist constants c'' and c''', such that

$$c''/|LD_s| \leq Prob(y = Index(X_i)) \leq c'''/|LD_s|.$$

Furthermore, [104, 105] shows that the size of the set of leaves $|LD_s|$ limits to $H|D_s|$ for large $|D_s|$. Recall that in our algorithm $|D_s| = |S| = O(n/\log n)$. Combine the above results and the fact that the the number of indexed keys falling into a

certain bucket follows binomial distribution, we conclude that the number of keys in

$$Bucket(y) = \{x_i | Index(x_i) = y\}$$

is less than $O(\log n/H)$ with probability $\geq 1 - \frac{1}{n^{\Omega(1)}}$. □

We state the time complexity theorem of our algorithms as follows.

Theorem 3.1 *The sorting algorithm takes $O(n \log(\frac{\log n}{H}))$ sequential expected running time where H is the entropy of the input keys.*

Proof: Step 1 of the algorithm takes $O(|S|)$ time where $|S| = n/\log n$ is the number of the sampled keys. In step 3, we define average length of the first prefix match

$$\bar{L} = \frac{\Sigma_{i=1}^{n} d_i}{n} \tag{3.2}$$

Note that the optimal compression ratio ρ of the inputs is related to the expected length of first prefix match of the inputs such that

$$\rho = \frac{\bar{L}}{\log n}.$$

Also recall that searching the index of X_i takes $O(\log d_i)$ time where d_i is the length of $Index(X_i)$. Inserting key X_i into the corresponding bucket takes $O(1)$ time. Therefore the time complexity for step 3 is $\Sigma_{i=1}^{n} \log(d_i) = \Sigma_{i=1}^{n} \log(d_i) = \log \Pi_{i=1}^{n} d_i \leq \log \bar{L}^n = n \log \bar{L} = n \log(\rho \log n) = n \log(\frac{\log n}{H})$.

Similarly, the time for step 2 can also bounded by $n \log(\log n/H)$. Finally In step 4, sorting leaves in D_s only takes time no more than $O(n)$ as $|S| = n/\log n$ and $|LD_s| \leq |S|$. Then each bucket is sorted separately using a sorting algorithm which takes time $O(u_i \log u_i)$ for each bucket where u_i is the size of the ith bucket. Note that

$$|LD_s| \leq |S| = n/\log n.$$

By Lemma 3.2, the size of each bucket is bounded by $O(\log n/H)$ with high likelihood, the expected time complexity of step 4 therefore is

$$\begin{aligned} T_4 &= \Sigma_{i=1}^{|LD_s|} t_i \\ &= \Sigma_{i=1}^{|LD_s|} u_i \log u_i \\ &\leq O(\Sigma_{i=1}^{n/\log n} u_i \log(\log n/H)) \\ &= O(n \log(\tfrac{\log n}{H})) \text{ as } \Sigma_{i=1}^{|LD_s|} u_i = n. \end{aligned}$$

Thus, the total sequential time complexity of the algorithm is $O(n \log(\frac{\log n}{H}))$. □

We immediately have the following corollary assuming that the entropy H of the input keys is bounded:

Corollary 3.1 *The algorithm takes $O(n \log\log n)$ expected time if $H \geq \frac{1}{(\log n)^{O(1)}}$.*

3.2.4 Experimental Testing

We implemented the sequential version of our sorting algorithm on SPARC-2 machine. The standard sorting algorithm we use for sorting the keys inside each bucket is the Sun OS/4 system call *qsort*. We tested the algorithm on 20 files with 24,000 to 1,500,000 keys from a wide variety of sources, whose compression ratios ranged from 2 to 4. We also derived an empirical formula for the runtime bound of our algorithm. For comparison, we also did this for the UNIX system sorting routine - Quicksort. The results are shown in Figure 3.2.

It is well know that the time of sorting each key using Quick sort can be expressed in terms of the total number of keys: $T_q = c_0 + c_1 \log N$. As we already showed in the description of our sorting algorithm, the corresponding empirical formula for our sequential algorithm with fixed sample size is (regard the entropy as a small constant): $T_e = d_0 + d_1 \log\log N$.

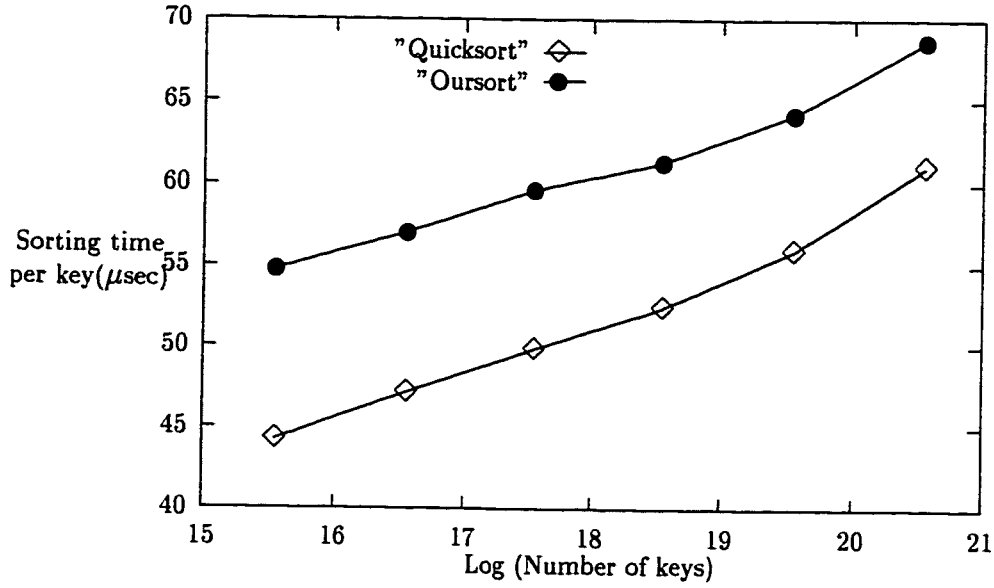

Figure 3.2: Comparison of our sorting algorithm with Quick sort.

We determined the constants using the experimental results:

$$c_0 = 14.2, c_1 = 1.68, d_0 = 40.00, d_1 = 3.73.$$

From the empirical formulas, it is easy to see that our sorting algorithm has a larger constant for the linear factor and that explains why our algorithm does not perform very well when the size of the input is small. The larger constant for the linear factor is due to the large overhead caused by random sampling and dictionary indexing. However, the linear factor becomes less dominant when the size of the input grows large enough, as the formula shows. As predicted, our sorting algorithm performs better and better when the input size becomes larger and larger. By extrapolation from these empirical tests we found our sequential sorting algorithm will beat quicksort when the total number of input keys $N > 3.2 \times 10^7$.

3.2.5 Discussion

In this section, we have presented the sequential sorting algorithm and its implementation. Like other non-trivial algorithms, our algorithms performs well when the

size of the input becomes large enough to cancel out the effect of large overhead. The major cost of any sorting algorithm is the time spent for any key to decide its relative position in the whole input. Our basic idea is based on this observation and we try to eliminate unnecessary comparisons by dividing keys into ordered subgroups so that the only information needed for any key to be sorted is to find its relative position in its subgroup. Since the input key itself, especially its prefix, contains informations on which subgroup it belongs to, the preprocessing, or learning phrase of our algorithm uses this information to index keys to buckets instead of redundant comparisons between two keys.

We utilize the trie data structure to execute string operations such as finding the maximum prefix of any given key in the trie and inserting new phrases to the trie. We also use the idea of constructing a dictionary using a method similar to dictionary construction in LZ compression. The dictionary we construct has a nice property that it becomes a better and better approximation of the probability distribution of the input source as it incrementally inserts more and more input keys. Intuitively, if certain prefix has a high probability of appearing in input keys, there will be proportionally many input keys with this prefix that are sampled. The dictionary will reflex the underlying probability distribution with high likelihood if the number of input keys is large enough. Furthermore, as a property of the trie, the probability of an input key reaching to any leaf of the trie is about the same (see Lemma 3.2). This makes the task of finding a good partition of buckets especially easy.

Studies have indicated that sorting comprises about twenty percent of all computing on mainframes. Perhaps the largest use of sorting in computing (particularly in business computing) is the sort required for large database operations (e.g., required by joint operations). In these applications the keys have length of many machine words. Since our sorting algorithm hashes the key (rather than compare entire keys

as in comparison sorts such as quicksort), our algorithm is even more advantageous in the case of large key lengths in which case the cutoff is much lower. In case that the compression ratio is high (i.e. the input source is highly predictable), which can be determined after building the dictionary, we just adopt the previous sorting algorithm, e.g., quick sort.

In later sections, we will demonstrate that the same techniques used in sequential sort (trie data structure, fast prefix matching etc) can be extended to other problems (e.g., computational geometry problems) to decrease computation by learning the distribution of the inputs.

3.3 Priority Queue Operations

A *priority queue*[26, 10, 64] is a data structure representing a single set S of elements each with an associated value called a *key*. A priority queue supports the following operations:

Insert(S, x): insert the element x into the set S. This operation can be written as $S \leftarrow S \cup x$.

Max(S): return the element of S with the largest key.

Extract-Max(S): remove and return the element of S with the largest key.

These operations can be executed *on-line* (i.e., in some arbitrary order and such that each instruction should be executed before reading the next one). The priority queues are useful in job scheduling and event-driven simulator. In some applications, the operations **Max** and and **Extract-Max** are replaced by operations **Min** and **Extract-Min** which returns the element with the smallest key.

Our sequential algorithm can be modified to perform priority queue operations. The bottleneck of the algorithm of [10] is the time for prefix matching which depends

on the maximum length of all possible keys. Applying our algorithm to build an on-line hash table which contains all the keys presently in the priority queue, the time for prefix matching an arbitrary key can be reduced to $O(\log\log n/H)$ by searching through the hash tables. As an example, given a priority queue S and a key x to be inserted, we use the precomputed hash table to search for the maximum prefix match for x in S. This effectively gives the position of key x in S. We then proceed to insert x into S which takes constant time once the position of x in the hash table is known. After key x is inserted into the trie, we use a pointer to link it to the key which is *left neighbor* of x had all the keys been explicitly sorted (i.e. the biggest key in all the keys that are smaller than x). The left neighbor can be identified in the process of searching. We keep a pointer pointing to the maximum key of the input keys. The Extract-Max operation can be performed in constant time by moving the pointer pointing to the maximum key to its left neighbor.

The time complexity of each operation does not depend on the length of the longest input key. Instead, since the running time depends on the length of the maximum prefix match for any key in the priority queue, the complexity is only related to the entropy of input keys. We have the following corollary.

Corollary 3.2 *Assume that the input keys are generated by a stationary and ergodic source satisfying mixing condition, the priority queue operations can be performed in* $O(\log(\log n/H))$ *expected sequential time where* H *is the entropy of input keys.* □

3.4 Randomized Parallel Sorting Algorithm

The sequential sorting algorithm we presented was based on a divide and conquer strategy. That is, the problem is divided into subproblem after learning certain statistical properties of the input keys. The division of the subproblems is carefully carried out so that the subproblems are of even size. The subproblems are then solved

independently and the results are combined together to form the overall solution of the problem. Therefore, this parallalism underlying this algorithm can be utilized to design an efficient parallel version of our algorithm which will be presented in this section.

There has been extensive research and also implementations in the area of parallel sorting. Reischuk [96] and Reif & Valiant [95] gave randomized sorting methods that use n processors and run in $O(\log n)$ time with high probability. The first deterministic method to achieve such performance was an EREW comparison PRAM algorithm based on the Ajtai-Komlós-Szemerédi sorting network. However, the constant factor in this time bound is still too large for practical use though it has an execution time of $O(\log n)$ using $O(n)$ processors. Later, Bilardi & Nicolau [14] achieved the same performance with a practical small constant. Cole [27] also has given a practical deterministic method of sorting on an EREW comparison PRAM in time $O(\log n)$ using $O(n)$ processors. Reif & Rajasekaran [90] presented an optimal $O(\log n)$ time PRAM algorithm using $O(n)$ processors for integer sorting where key length is $\log n$ and Hagerup [54] gave an $O(\log n)$ time algorithm using $O(n \log\log n/\log n)$ processors for integer sorting with a bound of $O(\log n)$ bits per key.

List ranking problem is that given a singly linked list L with n objects, we wish to compute, for each object, the distance from the end of the list. The known lower bound for list-ranking is $\Omega(\log n/\log\log n)$ expected time (Beame & Hastad [8]) for all algorithms using polynomial number of processors on CRCW PRAM. If the results of the sorting algorithm are required to rank all the keys instead of just a relative ordering of them, this lower bound applies to the sorting problems as well. However, in many problems only the relative ordering information of the keys is needed and the sorting algorithm does not include a list-ranking procedure at the end, the sorting can be accomplished much faster. If this is the situation, the sorted keys can be loosely

placed in the final list with empty places between any two of them. For example, [79] gives a *Padded Sort* (sorting n items into $n + o(n)$ locations) algorithm which requires only $\Theta(\log\log n)$ time using $n/\log\log n$ processors, assuming the items are taken from a uniform distribution.

We present a parallel version of our algorithm on CRCW PRAM model (see introduction for CRCW). We have the following theorem.

Theorem 3.2 *Let L_{max} be the number of bits of the longest sample key. If $L_{max} \leq n^{O(1)}$, we get $O(\log n)$ expected time using $O(n\log(\frac{\log n}{H})/\log n)$ processors for parallel sorting where H is the entropy of the input.*

Proof: Assume that $L_{max} \leq n^{O(1)}$ and all keys are originally put in an array A_1 of length n.

The first part of our algorithm is indexing n keys to $|LD_s|$ buckets in parallel using $P = O(\frac{n\log\bar{L}}{\log n})$ processors. There are $\log\log n + 1$ stages which we will define below.

Stage 0: here we do random subsampling of size $n/\log n$ from the n input keys, build up the hash table for indexing, precompute all random mapping functions for $i = 1...\log\log n$,

$$R_i : [1...n_i] \rightarrow [1...n_i]$$

where $n_i = \frac{n}{(c_3)^i}$ for some constant c_3 which will be determined below. All those operations take $O(\log n)$ parallel time using $O(n/\log n)$ processors [80, 62].

Stage 1: we arrange n keys in random order on an array with length n and randomly assign $O(\frac{\log n}{\log\bar{L}})$ keys to each processor. Each processor $j(1 \leq j \leq P)$ then works on keys from $(j-1)\frac{\log n}{\log\bar{L}}$ to $j\frac{\log n}{\log\bar{L}}$ assigned to it separately for $c_0\log n$ time where c_0 is a constant greater than 1. In particular, each processor indexes keys to buckets using the same technique described in our sequential algorithm. We can show at the end of

this time there will be at most a constant factor decrease in the number of all keys left unindexed, say n/c_1 where $c_1 > 1$. Then we use the precomputed random mapping function R_1 to map all keys to another array A_1 with the same size of n which takes $\log n/\log \bar{L}$ time using P processors. We now subdivide A_1 into segments each with size $c_2 \log n$. Then a standard prefix computation is performed within each segment to delete the already indexed keys which takes $O(\log \log n)$ parallel time using n processors, so we can slow it down to $O(\log n)$ time using $n\log \log n/\log n$ processors. Define $c_3 = c_2/(1+\epsilon)$ where ϵ is a given constant say $1/3$. The size of array A_1 now becomes n/c_3. Then we assign processor j keys from $(j-1)\frac{\log n}{c_3 \log \bar{L}}$ to $j\frac{\log n}{c_3 \log \bar{L}}$. To bound the number of unindexed keys per processor, we have the following lemma :

Lemma 3.3 *The number of unindexed keys for each processor after random mapping is at most* $\frac{c_2}{c_1}\log n(1+\epsilon)$ *with probability* $1 - 1/n^{\Omega(1)}$ *for some small constant* $\epsilon(0 < \epsilon < 1/2)$ *and large* c_2.

Proof: As the mapping function is random, the number U of unindexed keys falling into a given segment in A_2 is binomial distributed with probability $1/c_1$ and mean $\mu = \frac{c_2}{c_1}\log n$. Applying Chernoff Bounds (see [55] and also introduction), we have

$\Pr(U \geq \frac{c_2}{c_1}(1+\epsilon)\log n) \leq e^{-\frac{\epsilon^2\mu}{2}} = \frac{1}{n^{\Omega(1)}}$

for large c_2 and given ϵ. □

Stage i: now we apply all operations similar to those in stage 1 on the array A_{i-1}. We assume A_{i-1} has length $n/(c_3)^{i-1}$ and contains only remaining keys still to be indexed. Each of these stages takes $\frac{c_0}{(c_3)^{i-1}}\log n$ time using P processors. For example, now the prefix sum can be done in $O(\log \log n)$ time using $\frac{n}{(c_3)^{i-1}}$ processors. So slowing down the time to $\frac{O(\log n)}{(c_3)^{i-1}}$, we only need $n\log \log n/\log n$ processors which is less than P. All other time bounds are achieved similarly.

Repeat this stage until the number of keys left unindexed becomes $n/\log n$ which is less than P. Then we can finally assign each key a processor to finish the job. The final work takes $O(\log n)$ time using P processors as long as $L_{max} \leq n^{O(1)}$.

It is easy to see that the number of stages required is $\log\log n$. The total parallel time complexity of these stages then is

$$O(\Sigma_{i=0}^{\log\log n}(\frac{1}{(c_3)^i}c_0\log n)) \leq O(\log n)$$

.

Next, we sort each bucket separately using a standard comparison based parallel sorting algorithm. Each bucket i of size B_i takes $O(\log B_i)$ time using $O(B_i)$ processors. As each bucket size is bounded by $O(\log n/H)$ and $\Sigma_{i=1}^{|LD_s|}B_i = n$, this sorting of buckets takes $O(\log(\log n/H))$ time using $O(n)$ processors. To relax the time bound to $O(\log n)$, we require $O(n^{\frac{\log(\log n/H)}{\log n}})$ processors which is upper bounded by P.

Finally, we use standard list-ranking algorithm to rank each key which takes only $O(\log n)$ time using $O(n/\log n)$ processors (see [62]).

The initial construction of the dictionary D_s takes $O(\log n)$ time using P processors by similar techniques. Thus, the total time is $O(\log n)$ using P processors. □

3.5 Application: Convex Hull Problem

In this section, we apply our sorting algorithm to a computational geometry problem - the convex hull problem. We show that the efficient sorting algorithm can improve the performance of convex hull algorithm with certain statistical assumptions on the inputs.

The convex hull problem is defined as follows.

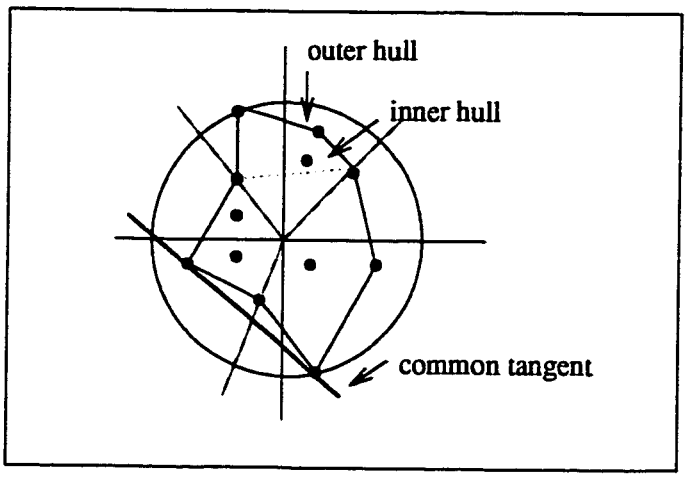

Figure 3.3: Two dimensional convex hull.

Convex hull problem: Given a set of points in E^d (the Euclidean d-dimensional space), find the smallest polygon containing these points.

The sequential convex hull problem takes $\Omega(n \log n)$ operations for n points in two dimensions and there are algorithms with matching bound. However, It is well known that the two dimensional convex hull problem can be solved in linear time if the points are presorted by their x coordinates. Hence we have the following corollary if we apply our sorting algorithm to sort the x-coordinates of the input points:

Corollary 3.3 *Given a set of points, the two dimensional convex hull problem can be solved in* $O(n(\log(\log n/H)))$ *sequential time where* H *is the entropy of the* x*-coordinates of the input points.*

Since the sequential algorithm is straightforward, we are interested in designing an efficient parallel algorithm for the convex hull problem, using some well known techniques in parallel algorithm design. Below we present a parallel version of the algorithm for solving the two dimensional convex hull problem. The input is given as a set of n points represented by their polar coordinates (r, θ). We also assume that the set of points has a random distribution on the angular coordinate θ while the radial coordinate r has an arbitrary distribution. Also we assume that the points

have been presorted by their angular coordinates. The parallel machine model we use is CRCW PRAM.

First we choose independently $O(\log n)$ sets of random samples each of size $O(n^{\epsilon})$ (ϵ is a small constant, say 1/3) and apply the *polling* technique of [91] to determine a good sample which divides the set into $O(n^{1/3})$ sections (with respect to θ) each with $O(n^{2/3})$ points with probability $\geq 1 - n^{-\alpha}$ where α is a constant. The polling is done by chosing $O(log^2 n)$ subsamples and testing for a good sample on a fraction $n/(\log n)^{O(1)}$ of the inputs and it can be proven that by applying the polling technique a good sample of the inputs can be obtained with high probability (refer to [91] for further details).

Within each section, the convex hull is constructed by applying recursively the merging procedure which we will describe below. This merging procedure is also applied to construct the final convex hull. The convex hull for the base case where each section contains less than three points can be constructed in constant time by a linear number of processors. We now describe the merging procedure (after constructing the convex hull of each section) in detail as follows:

Let ϵ' be another small constant, say 1/9. We assign $n^{\epsilon'}$ processors for each pair of sections to find their common tangent. The total number of processors used is $(n^{1/3})^2 n^{\epsilon'} = O(n)$. As the problem of finding common tangent between any pair of sections are virtually the same, we focus our attention on only one pair of sections, say section i and j. However, all the common tangents between pairs of sections are to be found in parallel. Finding the common tangent between section i and j is carried out in a constant number of steps, precisely $\frac{2/3}{\epsilon'}$ steps. In the first step, we divide both sections into $O(n^{\epsilon'})$ subsections and solve the problem of finding the common tangent between these two sections after we replace each part by the line segment linking its two endpoints (the points with minimum and maximum value of

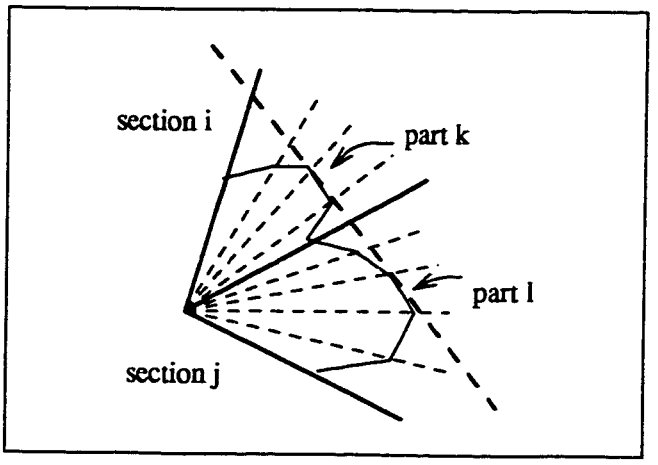

Figure 3.4: Finding common tangent in a refining process.

θ in the subsection). This problem can be solved in constant time using the method discussed in [62] which finds the common tangent in $O(1)$ parallel time using a linear number of processors.

This common tangent (between subsection k in section i and subsection l in section j) we found is not exactly the one we need but very close because the real common tangent (between section i and j) must go through subsection k and subsection l by the convex property (see Figure 3.4).

Then subsection k and l are further divided into $n^{\epsilon'}$ parts. The size of the problem is now reduced to $O(n^{2/3-\epsilon'})$. In the following steps, we apply the same method ($O(1)$ time for each step) to refine the common tangent until we find the one needed. The total time is still a constant.

After we find all the common tangents between pairs of sections, we construct the convex hull by allocating edges of convex hull in a counter-clockwise direction. For each section, find the common tangent with the minimum counter-clockwise angle with respect to the zero-degree line of the polar system. This can be accomplished in $O(1)$ time by assigning each section $O(n^{2/3})$ processors to find the minimum-angle common tangent on a CRCW PRAM [26].

So the merging step takes only $O(1)$ time and $O(n)$ processors. The time complexity for the parallel algorithm is therefore

$$\bar{T}(n) = \bar{T}(n^{2/3}) + O(1) = O(\log\log n).$$

In each recursive step, the expected number of processors required is always $O(n)$. As the work in each recursive step is bounded by $O(n)$ with high probability, the total work for the parallel algorithm is thus $O(n \log\log n)$ with high probability $\geq 1 - 1/n^{\Omega(1)}$.

The above algorithm can be summarized as follows:

Theorem 3.3 *Given a set of two dimensional points represented by their polar coordinates (r, θ) and assuming that the angular coordinates are presorted, the convex hull of n points can be constructed in $O(\log\log n)$ expected time using $O(n)$ processors.*

We immediately have the following corollary by applying our parallel sorting algorithm to sort the angular coordinates of the input points:

Corollary 3.4 *Given a set of two dimensional points represented by their polar coordinates (r, θ) and let H be the entropy of the angular coordinates, the convex hull of n points can be constructed in $O((\log(\log n/H)))$ expected time using $O(n)$ processors on CRCW PRAM.* □

3.6 Related Work

Following our entropy based sorting algorithm, Andersson and Nilsson [4] later presented a radix sort algorithm with improved time complexity. Their algorithm analyzes the complexity of the algorithm in terms of a more general measurement, *distinguishing prefixes*. Specifically, their algorithm sorts a set of n binary strings in $\Theta(n \log(\frac{B}{n \log n} + 2))$ time, where B is the number of all distinguishing prefixes, i.e., the minimum number of bits that have to be inspected to distinguish the strings.

The distinguishing prefix is a more general measurement of difficulty of sorting than the entropy of the source from which the inputs are generated. For a stationary ergodic process satisfying a certain *mixing condition* (see [4] for details), the following equation holds:

$$\lim_{n\to\infty} \frac{E(\bar{B})}{\log n} = \frac{1}{H}.$$

Therefore, for any algorithm where the complexity can be expressed in terms of $\bar{B}$, the complexity can be expressed in terms of entropy H as well. The reverse is not true for general inputs. Therefore, the approach analyzing the complexity of an algorithm in terms of distinguishing prefix is more general than one using the entropy of a stationary ergodic source.

However, it is noteworthy that their computational model used in the radix sort is different from ours in that their model assumes that rearranging the binary strings (the input keys) can be all accomplished by moving pointers. Also, more importantly, their algorithm depends on the assumption that the expected length of distinguishing prefix of each key is bounded by a constant machine words, i.e. $w = \Omega(\bar{B})$, while our sorting algorithm does not depend on this assumption. In fact, in our algorithm, the expected length of distinguishing prefix is $O(\log n/H)$ rather the length of a constant machine words. If we impose such assumptions, our algorithm runs in $O(n)$ expected time.

3.7 Summary

In this chapter we investigated the first problem, sorting problem, in our study of applying data compression techniques to improve efficiency of algorithms. After a survey of previous work on sorting problem, we introduced a new randomized sorting algorithm with certain statistical assumptions based on trie data structure and binary

string processing widely used in data compression algorithms. The algorithm was implemented and showed good performance comparing with system sorting routine. In addition, we gave a parallel version of the randomized algorithm in the hope that the divide and conquer nature of our algorithm can lay a solid ground for a highly parallel implementation. Finally we extend our method to convex hull problem to reduce the overall time complexity of the algorithm to $O(\log(\log n/H))$.

Our primary motivation of the study of the sorting problem is not trying to design a fastest sorting algorithm. Instead, we concentrated on studying the problem of how to apply data structures and associated computational techniques to design efficient sorting algorithm and how to evaluate the validity of certain statistical assumptions on the input data and utilize these assumptions to aid algorithm design. Also we presented several results in parallel settings because the divide-and-conquer nature of our algorithm makes a parallel implementation extremely attractive.

In the next chapter, we will study another fundamental algorithmic problem - string matching. The motivation to study string matching is the sheer importance of this problem and its difference from the sorting algorithm. As sorting is an important operation in scientific computations, string matching is crucial in many areas such as data processing, information retrieval and database design. Furthermore, unlike the sorting problem where the inputs are a set of independent keys, the string matching problem regards the input as a consecutive data stream. Thus a different approach in handling the input data is needed to design new algorithms based on data compression ideas. Instead of applying certain data structure in design of new algorithms, we will emphasize on how to analyze the time complexity of a new algorithm based on statistical assumptions of the inputs.

Chapter 4

Fast Pattern Matching for Entropy Bounded Text

In Chapter 3, we examined how data compression data structures and associated computational techniques can be utilized in designing efficient sorting algorithm based on statistical assumption of the inputs. We especially concentrated on applying trie data structure as a tool to learn an approximation of the input's probability distribution and using this information to aid computation. In this chapter, we study another important algorithmic problem, string matching. We will emphasize on how to base the analysis of the performance of the new algorithm on certain statistical assumption of the inputs. The expected running time of this algorithm is $O(\frac{nlogm}{\phi(H)m})$ where H is the entropy of the text and $\phi(H)$ is a function of H. Our algorithm and its complexity analysis illustrate the relationship between the compressibility of the inputs and the performance of the algorithm.

4.1 The Problem

Given a text of length n and a pattern of length m, the pattern matching problem is to find all occurrences of the pattern in the text. The characters of the text and the pattern are drawn from an alphabet Σ.

There have been various efficient algorithms designed for both one and higher dimensional pattern matching problems [63, 66, 97]. Knuth, Morris and Pratt [66] gave an algorithm which runs in $O(n + m)$. assuming the text is generated from a source of uniform distribution. In practice, the algorithm due to R.S. Boyer and J.S. Moore [13] (also independently by R.W. Gosper) seems to be the most efficient

algorithm. Their algorithm uses two effective heuristics, known as the "bad-character heuristic" and the "good-suffix heuristic", to allow the algorithm to skip the unnecessary examinations of many characters of the text. For a binary alphabet, Boyer and Moore's algorithm only needs an average of $O(n \log m/m)$ bit inspections to determine a match.

Two general approaches have been adopted in designing pattern matching algorithms: to base the analysis of the algorithm either on the frequency of matching or on the frequency of mismatching. We adopt the latter one in analysis of algorithms complexity. The reason behind this is that it is unlikely for the pattern to match an arbitrary section of the text which has the same length, i.e. general input does not contain too much redundancy. Thus we are able to skip more text sections when a mismatch happens. In other words, when we try to match a pattern against a text, we keep on comparing the characters of the pattern against those of the text until there occurs a mismatch. Then we decide how far we can move the pattern ahead without comparisons.

In this chapter, we consider the problem of string matching with the assumption that the text has bounded compressibility. For classes of large data file found in practice, the compressibility of the file is indeed bounded within a constant, no more than say 4 up to 20 (see discussion in [28] and also introduction). This fact has been widely accepted and some novel algorithms have been designed based on this assumption.

In contrast, some previous algorithms assume that the text is generated from a source which is uniformly distributed (i.e. each character appears in the text with equal probability). We find that this is not a reasonable assumption for practical files and we are not assuming uniform distribution for the text throughout this chapter. Instead, we assume that the text is generated from a stationary and ergodic source

String matching algorithms	*Assumptions on inputs*	*Running time*
KMP	uniform distribution	$O(n+m)$
Boyer-Moore	none	$O(nm)$
Our algorithm	bounded entropy H	$O(n(\frac{\log m}{cm}))$

Table 4.1: Comparison of string matching algorithms.

S_T. We assume the text has a bounded entropy, i.e. it is not highly compressible. Our algorithm takes a similar approach as Boyer-Moore algorithm while we investigate the relationship between complexity of our string matching algorithm and the entropy of the text.

In this section, we will survey previous methods, notably the well known KMP algorithm and the efficient Boyer-Moore algorithm. These two algorithms employ different strategies yet have a common idea of reducing the computation by preprocessing the pattern to skip unnecessary comparisons. Boyer-Moore algorithm is more complicated but more efficient in practice which combines preprocessing with two heuristics. The analysis of running time of these algorithms is based on worst-case situation. However, the discrepancy between the actual performance and the worst case running time invites a more careful study of the expected time analysis which will be the focus of our algorithm.

4.1.1 KMP string matching algorithm

The linear-time string-matching algorithm designed by Knuth, Morris and Pratt (KMP algorithm) is perhaps the most well-known string matching algorithm. The basic idea of this algorithm is to save unnecessary comparisons by preprocessing the pattern before matching it against the text. The preprocessing phase of the algorithm computes an auxiliary *prefix function* $\pi[1...m]$ from the pattern. Intuitively, this function computes for each character q in P, the length of the longest prefix of P that is a proper suffix of P_q.

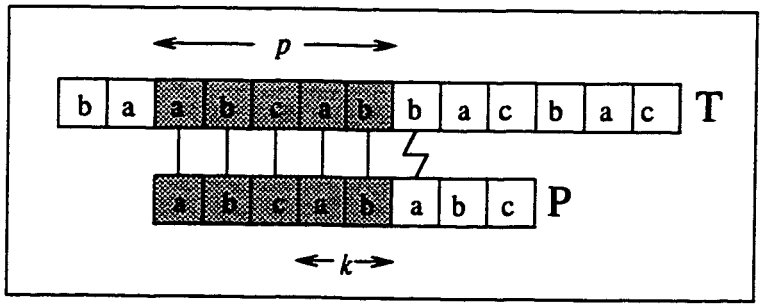

Figure 4.1: The prefix function in KMP algorithm.

The precomputed function $\pi[1...m]$ contains all the suffix information of the pattern so that the algorithm can skip unnecessary comparisons during the matching process. To see how the prefix information of the pattern can be used to reduce number of comparisons between the pattern and the text, let us consider the following example. Figure 4.1 shows a pattern P = abcababc against a text T. Note that the first five characters in P match to T. What is the next possible valid position for pattern P to match the text? That is the longest suffix of P which is also a prefix of P. In this example, the length of such a suffix is 3. Therefore, we can shift the pattern to the right of the text by 7 positions, skipping all the comparisons for those positions because they are invalid. As the information we use only concerns the pattern, we can precompute the information by comparing the pattern to itself in different alignments.

Formally, given a pattern $P[1...m]$ and let $x \sqsubset y$ denotes that string x is a prefix of string y, we define the prefix function $\pi : \{1, 2, ...m\} \rightarrow \{0, 1, ..., m-1\}$ for the pattern such that

$$\pi[q] = \max\{k : k < q, P_k \sqsupset P_q\}$$

After precomputing the prefix function, the matching algorithm scans the text while comparing the characters of the pattern to those of the text. The precomputed prefix function is used to determine the maximum number of characters that can be skipped without comparison. The computation of the prefix function can be done in

$O(m)$ time and the matching algorithm runs in $O(n+m)$ in the worst case.

4.1.2 Boyer-Moore string matching algorithm

The Boyer-Moore algorithm incorporates two heuristics (*bad-character* and *good-suffix*) to skip unnecessary comparisons between the characters in the pattern and the text. When a mismatch occurs, each of these heuristics calculates the next position in the text without comparing the characters between the current position and the next one.

The Boyer-Moore algorithm chooses the next position which skips more characters in the text. We call the character in the place where a mismatch occurs a "bad" character. In case of a mismatch, the bad-character heuristic uses the precomputed information about where the bad text character $T(s+j)$ occurs in the pattern (if it occurs at all) to propose a new shift. The best situation happens when the mismatch occurs in the first character of pattern there is no such bad character occurring in the pattern. In this case, we can safely move the pattern forward by the length of the pattern m because there is no possible match between the current place and the next. Boyer-Moore algorithm will only have to compare $\lceil n/m \rceil$ times to determine if the best case occurs all the time.

The other heuristic, the good suffix heuristic, works in the opposite way. This heuristic tries to move the pattern to the right *without* causing any mismatches between the already matched part of the pattern and the text. Formally, let us define the relation $Q \sim R$ (read "Q is similar to R") for strings Q and R to mean that $Q \sqsubset R$ or $R \sqsubset Q$. If two strings are similar, we can align them with their rightmost character without any mismatched pair of aligned characters. If in case of a mismatch, we find that $P[j] \neq T[s+j]$ $(j < m)$, then the good-suffix heuristic says

that we can safely move pattern to the right by

$$\gamma[j] = m - \max\{k : 0 \leq k < m, P[j+1...m] \sim P_k\} \tag{4.1}$$

In other words, $\gamma[j]$ is the least amount we can move pattern to the right without causing any characters in the "good suffix" $T[s+j+1...s+m]$ to be mismatched against the new alignment of the pattern. The computation of the "good-prefix" function γ takes $O(m)$ time by scanning through the pattern from left to right (see [26] for details).

By incorporating these two heuristics, the Boyer-Moore algorithm is able to effectively avoid many unnecessary comparisons between the text and the pattern. The algorithm works by moving the pattern against the text from left to right but comparing in each position the characters of the pattern against the text from rightmost character to the left. When a mismatch occurs, each heuristic computes an amount by which the pattern can be safely moved to the right. The algorithm chooses the one which is larger and shifts the pattern to the right by this amount. This process repeats until a pattern matching is found or the rightmost character of the text is reached.

Although the worst case running time of Boyer-Moore algorithm is $O((n-m+1)m + |\Sigma|)$ where Σ is the alphabet, the algorithm is effective in practical use. The explanation behind what seems to be a contradiction is that for most practical input string, the probability of a random matching of a certain number of characters between pattern and text is small. In other words, the chance of a mismatch is large enough to allow the Boyer-Moore algorithm to work efficiently in real world.

This raises a natural question whether the analysis of the string matching can be based on the practical statistical property of the input text instead of worst-case input, and how efficient the algorithm is expected to run in practice. These questions motivates our study of a string matching algorithm based on some assumptions on

the input text and pattern.

4.1.3 Other related work

There have been studies done combining the two areas of string matching and data compression. For example, Amir, Benson and Farach [3] studied the text matching problem on compressed text, i.e. finding the first match of pattern without decompressing the text. They gave several implementations of the algorithm where the text is compressed using LZW compression. If u is the length of the *compressed* text and m is the length of the pattern, their algorithm finds the first pattern occurrence in time $O(u + m^2)$ or $O(u \log m + m)$ (the optimal solution of $O(u + m)$ time and space remains open). In the two-dimensional case, Amir and Benson [1] developed an almost optimal $O(u \log m)$ algorithm for two dimensional run-length compression, and then later Amir, Benson and Farach [2] gave an optimal $O(u)$ algorithm for this problem.

Another problem which bears more similarity to the problem we study is to find the relationship between "redundancy" in the text and the time complexity of the string matching algorithm. It has been previously noticed that the degree of the redundancy in the text is a major factor of the running time of string matching algorithms. Galil and Seiferas [49] gave a string matching algorithm which takes the *repetition factor* of a given text into account. Let y^i denote the concatenation of string y with itself i times. For example, $(abc)^2 = abcabc$. We say that a string $x \in \Sigma^*$ has repetition factor r if $x = y^r$ for some string $y \in \Sigma^*$ and some $r > 0$. Let $\rho(x)$ denote the largest r such that x has repetition factor r. For any pattern $P[1...m]$, we define $\rho^*(P)$ as $\max_{1 \leq i \leq m} \rho(P_i)$. It is provable that if the pattern P is chosen randomly from the set of all binary strings of length m, then the expected value of $\rho^*(P)$ is $O(1)$. Based on this discovery, Galil and Seiferas extended the

idea to achieve a linear-time string-matching algorithm that only uses $O(1)$ extra storage beyond what is required for P and T. Their algorithm is an example that a more careful analysis on the running time of a given algorithm is possible given certain assumptions on the inputs. In this chapter, however, we adopt a more general assumption on the compressibility of the input text instead of the repetition factor.

In our algorithms, the text is considered relatively long and subject to change from time to time. Thus only the pattern is preprocessed in order to make fast matching while the text remains unprocessed. This is actually the case in practice where preprocessing text and keeping the preprocessed data structure for the text can be too expensive comparing to the time saved in pattern matching. We assume that the pattern is generated from a probabilistic source S_P which may be or may not be independent of S_T.

4.2 String Matching Algorithm

Our algorithm works in a similar fashion as the Boyer-Moore algorithm in the sense that it tries to skip as many unnecessary comparisons as possible when a mismatch occurs. Instead of employing heuristics to help to skip unnecessary comparisons, we keep both the preprocessing and processing in our algorithm simple and easy to implement. We emphasize on analyzing the *expected* performance of the algorithm with respect to the entropy (which is related to optimal compressibility) of the text. By combining some heuristics into our algorithm, it is possible that we can make the algorithm to be more efficient. However, this is not the concentration of the study of this thesis which emphasizes the utilization of data compression ideas and techniques in algorithmic applications.

Let the text be a sequence of characters generated from an alphabet Σ of size c. Also let n and m denote the size of the text and the pattern respectively. Assume

the text has a bounded entropy H. Here we define the entropy with respect to the optimal compression ratio of the text ρ_{opt} where $H = 1/\rho_{opt}$. It is well known that the Lempel-Ziv compression scheme achieves optimal compression ratio with sufficient large inputs, i.e. $\rho \to \rho_{opt}$ when $n \to \infty$. Thus the entropy of the inputs can be estimated by applying Lempel-Ziv scheme to compress the inputs.

We also assume that the pattern is a sequence of characters generated from the same alphabet Σ which has length of m. We denote pattern P as $P[1...m]$ where $P[i] \in \Sigma$ for $1 \leq i \leq m$. Let $T[i,j]$ where $1 \leq i \leq j \leq n$ denote the subsection of text between the i-th character and the j-th character, inclusively. Similarly we define $P[i,j]$ where $1 \leq i \leq j \leq m$ as the subsection of pattern between the i-th character and the jth character, inclusively. We say two subsections are matched if they have the same length and every character is the same if we align them from the leftmost character. For the simplicity of our analysis, we assume that the pattern is generated from an independent source of S_T which has a uniform distribution. This assumption holds well in practice since in practical use, the pattern is generated on line and normally independent from the previous patterns.

Intuitively, the compressibility of the text string determines the frequency of repeated patterns in the text. If the text is highly compressible, it is more likely that the text contains a large number of redundant or repeated text. Furthermore, given an arbitrary string interval in the text, the probability that the interval gets a match in the pattern can be bounded using the optimal compressibility and thus the entropy of the text. Formally, a subsection $T[i,j]$ gets a match in the pattern P if there exists k, where $1 \leq k \leq m$ such that $T[i,j] = P[k, k+j-i]$.

We have the following lemma which relates the entropy of the text to the probability of an arbitrary interval with fixed length gets a match against the text at a random position of the text.

Lemma 4.1 *Assuming that the entropy of the text is H, an arbitrary interval Δ in the text of length $(1 + 1/H)\log m$ gets a match in text with probability $p' \leq 1 - H^2$.*

Proof: We prove this lemma by contradiction. Suppose this is not the case. Then the probability that an arbitrary interval Δ gets a match in the pattern is $p' > 1 - H^2$. Note that the optimal compression ratio $\rho_{opt} = 1/H$. We now show a compression method to compress the text.

The compression works by replacing intervals in the text by a pointer to a position in the pattern if there is a match. We divide the text into equal intervals each of length $(1 + 1/H)\log m$. For each such interval that gets a match in the pattern, we encode it using $\log m$ bits to represent the position of the leftmost character where it gets a match in the pattern. We then copy the other intervals of the original text into the compressed text directly without compression. The length of the whole text after the new compression scheme will be the sum of the length of the compressed text $(\frac{p'n}{1+\frac{1}{H}})$ and the uncompressed text $((1-p')n)$. The compression ratio of the new scheme can be calculated as:

$$\rho' = \frac{n}{\frac{p'n}{1+\frac{1}{H}} + (1-p')n} > 1/H = \rho_{opt}$$

We have a compression scheme that achieves a compression ratio greater than the optimal compression ratio, which is a contradiction. □

The following corollary enables us to bound the probability of matching for intervals of length $\log m$:

Corollary 4.1 *An arbitrary interval of length $\log m$ in the text does not get a match in the pattern with probability $p \geq 1 - (1 - H^2)^{H/(1+H)}$.*

Proof: Assume p' is the probability of an arbitrary interval of length $\log m$ in the text getting an match in the pattern. From Lemma 4.1, we have $(1-p)^{(1+1/H)} \leq 1 - H^2$. Thus the corollary follows. □

Our sequential pattern matching algorithm works in the following way. Initially, the pattern is aligned with the left end of the text. Moving the pattern gradually toward the right end of the text, we repeatedly match the text against the pattern from the right end of the pattern, similar to the Boyer-Moore algorithm. When a mismatch occurs or the pattern is fully matched against the text (in this case we get one match of the pattern in the text and output the position of the match in the text), the pattern is shifted to the right by a maximum distance. We try to maximize the length of each shift, skipping as many unnecessary inspections as possible. Note that in the worst case, the probability of getting a match for the pattern in the text can be small though the compressibility of the text is high. For example, a text composed by alternatively repeating a single word of length m (length of the pattern) and a random sequence of length m. The compression ratio is close to 2 while the probability of matching that single word is $1/m$. Therefore, we analyze our sequential algorithm in terms of expected running time instead of worst-case input.

We expect to shift the whole pattern to the right to skip as many unnecessary matches as possible when a mismatch occurs. In case of a match, we go on to compare the next pair of characters without shifting the position of the pattern until there is a mismatch. Then we shift the pattern to a certain distance to the right and restart the matching process from the leftmost position of the pattern.

Preprocessing of the Pattern

Since the time complexity of the algorithm depends on the probability of matching which is related to the compressibility of the text, we need to preprocess the pattern so that we can look up the next position in the pattern to skip unnecessary comparisons. One way of preprocessing the pattern is to construct a shift table similar to KMP algorithm so that we know where to restart matching process when a mismatch is

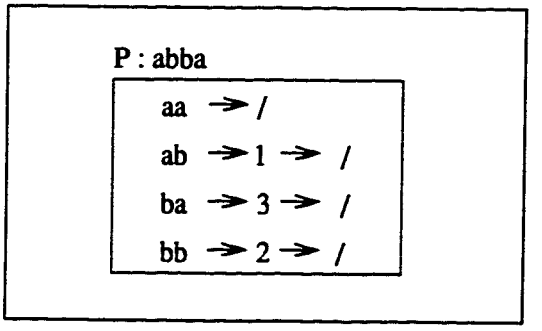

Figure 4.2: Example of a shift table.

found. However, it is somehow difficult to analyze the expected shift distance using this shift table.

We adopt another simpler way of preprocessing the pattern which produces a shift table that stores all occurrences of substrings of the pattern with length $\log m$. This method proves to be efficient and easier for complexity analysis. The shift table is constructed as follows. Consider all possible substrings of length $\log m$ in Σ^*. There are $m^{\log c}$ such substrings where c is the size of the alphabet Σ. In the shift table, we reserve one slot for every such substring according to alphabetical order. Each slot indexed with substring s will contain a linked list of position numbers of all occurrences of s in the pattern, in right-to-left order. The size of this shift table therefore is $O(m^{\log c})$ where c is the size of the alphabet because there are $m - \log m$ different substrings in P.

We scan the pattern from the right end to the left. For each substring s of length $\log m$ with leftmost character $P[i]$, we find the slot for s in the shift table and insert the position number i of s in the pattern to the end of the linked list. In later section, we will show that the expected number of a given substring of length $\log m$ occurring in the pattern is a constant provided that the pattern is generated from a random source. Therefore, the expected time to search for the next occurrence (in the right-to-left order) of a given substring in the pattern is also a constant.

Description and Analysis of the Algorithm

We start the processing by first aligning the left end of the pattern to the left end of the text. We denote "block" of the text or the pattern as a substring of length $\log m$. We compare the pattern to the text on a block-to-block basis (i.e. compare the substrings of length $\log m$ at a time instead of individual character). We start from the rightmost block of the pattern. The *position* of a given block is defined as the position of its leftmost character in the pattern, in left-to-right order. We use B_x to denote the block of position x. If the rightmost block $B_{m-\log m+1}$ matches the corresponding block of the text, we move on to check its left neighboring block, and so on, until we find a mismatch occurs between block of position x and its corresponding block in the text. Let B_x denote the block in P prior to the mismatch. If no such mismatch occurs, we have found a match of P in position x. We output x as the result.

In case of a mismatch, we find the position of next occurrence of the block B_x in the pattern. This position information is retrieved from the precomputed shift table by finding the index of B_x in the shift table and searching for the next occurrence of B_x in right-to-left order in the pattern. Let y be the position of next block B_x in the pattern and let B_y denote this next block. We then shift the pattern to the right by a number of $y - x$ characters and start the comparison from the left neighboring block of B_y.

There are two special cases need to be considered. When we start the comparison from B_x and find out all blocks to the left of B_x match to the text, we have found a successful match and output the current leftmost character of text where the leftmost character of pattern is matched against.

Another case is when there is no other occurrences of B_x to the left of it. In this case, we then simply shift the whole pattern P to the right by $x + \log m$ characters

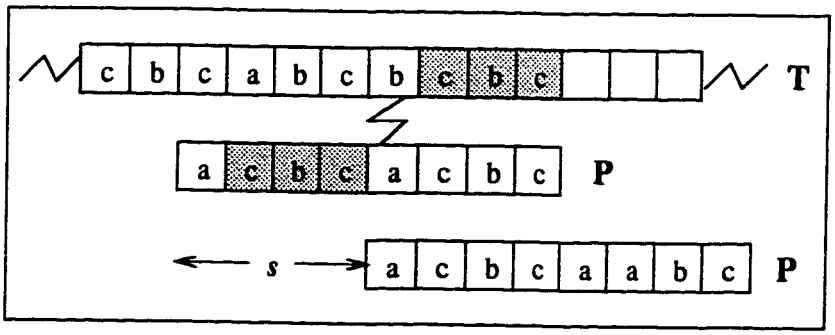

Figure 4.3: Example of shifting pattern in case of a mismatch.

and restart the comparison from the rightmost block of the pattern.

The algorithm proceeds until a match is found or the pattern moves out of the boundary of the text, in which case we have found no match between the text and the pattern.

We show the complexity analysis and prove its correctness of the algorithm in next theorem. First we show a lemma concerning the expected shift distance.

Lemma 4.2 *Assuming that the pattern of length m is generated from a random distribution and the alphabet size is c, the expected shift distance after each mismatch is $\Theta(m)$ for $c \geq 2$.*

Proof: The total number of all substrings of length $\log m$ is $m^{\log c}$. Therefore the expected number of one single string of length $\log m$ appearing in the pattern of length m is $max(1, m/m^{\log c})$. The expected distance between two consecutive identical substrings is therefore $min(m, m^{\log c})$. For $c \geq 2$, this expected distance becomes $\Theta(m)$. According to the algorithm described above, this is also the expected shift distance when mismatch occurs. □

Theorem 4.1 *The pattern matching algorithm takes $O(n \log m/pm))$ expected running time where H is the entropy of the text and $p = 1 - (1 - H^2)^{H/(1+H)}$.*

Proof: Recall from the algorithm, we shift the pattern only after there is a mismatch. The number of successful matches between blocks before a mismatch

occurs is a geometric distribution with probability p. Thus the expected number of comparisons before each mismatch is $O(1/p)$. Each comparison between blocks of length $O(\log m)$ takes time $O(\log m)$. By lemma 4.2, the expected distance of each shift is $\Theta(m)$. Therefore the expected total number of shifts is $O(n/m)$. As each shift takes place only when a mismatch occurs, the total sequential time follows.

Next we show that the algorithm finds all matches of the pattern to the text correctly. It suffices to show that there is no possible match missed due to each shift we make. We shift a patten only when a mismatch occurs. Let x be the position of the block prior to the mismatch in the pattern and B_x be this block. Also let T_x be the corresponding position in T that x matches against. We either find the next position y of B_x in the pattern or there is no more occurrence of B_x.

In the first case, we shift the pattern to the right by a number of $y - x$ characters. Let T_y be the position of the character in T that matches against $P[y]$ before the shift. Let B_y be the new block which is moved to the previous position of B_x. We show that there is no valid match between text and pattern in the skipped portion of the text. Suppose not, we have a valid match occurring when the pattern is aligned with the text by matching $P[x]$ to a character in T at some position T_z such that $T_y < T_z < T_x$. Therefore the block of the text starting from T_z must match to B_x. But B_y is the second block to the left identical to B_x in the pattern. This is a contradiction.

The same method can be applied to prove by contradiction in the second case where there is no block to the left of $P[x]$ which is identical to block B_x. We conclude that all our shifts are valid and the algorithm will not miss any possible match due to shifting of the pattern. □

It is obvious that $\lfloor n/m \rfloor$ is a lower bound of sequential string matching problem since the algorithm needs to inspect at least one character in every block of length

m in the text. There has been no proof of tight lower bound for the string matching problem. However, Knuth, Morris and Pratt [66] proposed a *conjecture* that patterns of length m exist for sufficiently large m, such that an expected number of $cn(\log m)/m$ inspections must be made to solve the string matching problem for text of sufficiently large size n. Here c denotes a positive constant independent of m and n. The intuition behind this conjecture is that for each interval of size m in the text, we need to do at least a binary-like search in order to determine if there is a match in that interval of text.

The expect running time of our string matching algorithm described above can be written as $O(n(\log m)/(\phi(H)m))$ where $\phi(H) = 1-(1-H^2)^{H/(1+H)}$ is a function of the entropy of the text. If the conjecture by Knuth, Morris and Pratt [66] is indeed true that the lower bound for expected running time of pattern matching problem is $\lfloor cn(\log m)/m \rfloor$, our string matching algorithm described above matches this lower bound for $c = 1/(1-(1-H^2)^{H/(1+H)})$, provided that the entropy of the text is bounded by a certain constant. It is easy to see that the assumption of a bounded entropy of text is independent of the length of the text or the pattern. The assumption is rather only based on a statistical property of the text.

The algorithm described above has an expected running time which is related to the entropy of the text. Now we consider the case where the variance of the distribution of the shift distance in the pattern can be used to bound the worst case running time.

We define the random variable X as follows: given an arbitrary string of length $O(\log m)$ in the pattern, X is the distance between this string and the next identical string on the left side in the pattern (in case there is no such identical string, the distance is m). Note X is also the random variable for the shift distances when mismatch occurs in our algorithm. The variance we are considering is $Var[X]$. Assume

the mean of the distribution X is μ and the length of the i-th shift is x_i. Note we have the following property:

$$Var[X] = \frac{\sum_{i=1}^{k}(x_i - \mu)^2}{k} \leq \frac{(\sum_{i=1}^{k}(x_i - \mu))^2}{k} \tag{4.2}$$

Thus,

$$\sqrt{kVar[X]} \geq \sum_{i=1}^{k}(x_i - \mu) \geq -\sqrt{kVar[X]} \tag{4.3}$$

and

$$k\mu + \sqrt{kVar[X]} \geq \sum_{i=1}^{k} x_i \geq k\mu - \sqrt{kVar[X]} \tag{4.4}$$

Since the complexity of our algorithm depends on the shift distance after each mismatch, the above bound on the total shift distance gives the worst case bound of our algorithm. Applying the above to the time complexity analysis of our algorithm, we have the following:

Theorem 4.2 *Assuming that the variance of the shift distance (as defined above) is V and pattern is of uniform distribution, the running time T of the one dimensional matching algorithm is bounded by the following:*

$$\frac{n \log m}{p(m + \sqrt{V})} \leq T \leq \frac{n \log m}{p(m - \sqrt{V})}.$$

Proof: First we loose the bound in 4.4 to

$$k(\mu + \sqrt{Var[X]}) \geq \sum_{i=1}^{k} x_i \geq k(\mu - \sqrt{Var[X]}) \tag{4.5}$$

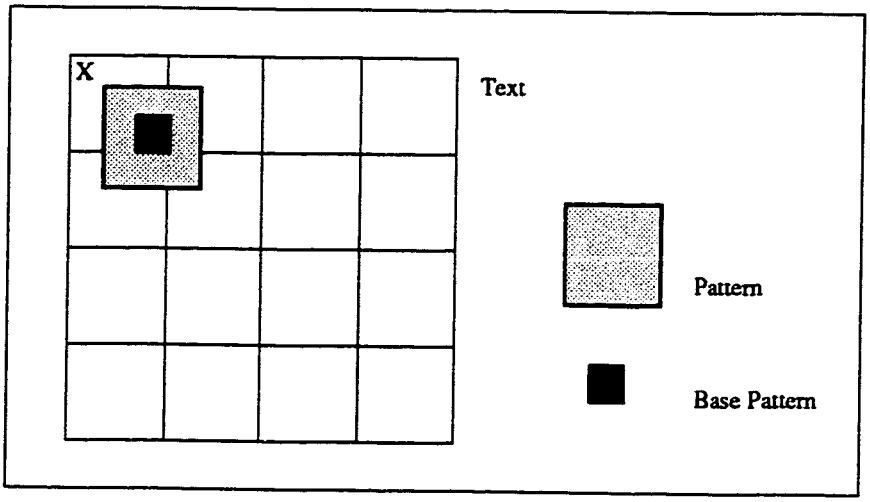

Figure 4.4: Two-dimensional pattern matching.

Then the total number k of shifts can be bounded by

$$\frac{n}{p(\mu + \sqrt{V})} \leq k \leq \frac{n}{p(\mu - \sqrt{V})} \tag{4.6}$$

Note n denotes the length of the text and $\mu = m$ by Lemma 4.2. Then we apply the same analysis of Theorem 4.1 to achieve the lower and upper time bounds. □.

4.3 Two Dimensional Pattern Matching

In this section, we extend our sequential string matching algorithm to two-dimensional case. We assume that the characters in the two dimensional text is generated from an alphabet of size c. Without loss of generality, we assume the size of the text is n where $n = s^2$ for some integer s. We denote the character in coordinates x, y of the text as $T[x, y]$. Since the basic idea in the two-dimensional matching is similar to the sequential case, we will only give an outline of our two dimensional matching algorithm.

In two-dimensional case, we need to search for a possible match of the pattern in two different directions: horizontal and vertical. The search path is a tree-like path

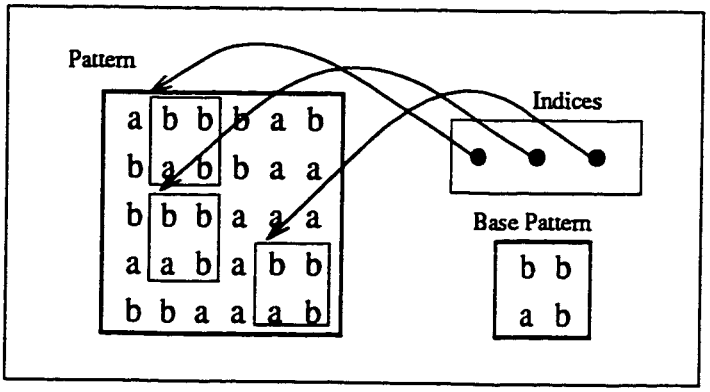

Figure 4.5: Preprocessing of a pattern.

starting with the root of the upper-left corner of the text. Each node d in this tree is represented by a point in the text with coordinates (x, y). Each child of d is the next possible matching with coordinates (d_x, d_y) where $d_x \geq x$ and $d_y \geq y$. The matching is accomplished in a fashion similar to a breadth-first search in this tree.

Preprocessing of Pattern

Given a pattern P of size $m = t^2$ generated from an alphabet of size c, we produce a data structure to speed up the matching process by preprocessing pattern P. Let *base pattern* be a portion of the pattern which is of size k where $k = \log m$. Without loss of generality, we assume that $k = b^2$ for some integer b. The base pattern is used in the similar way as the substring of length $\log m$ in the sequential case. We create a shift table whose indices are all possible base patterns of size k. Associated with each slot for base pattern B in the shift table is a linked list containing coordinates for all occurrences of B in P. pattern. For example, the base pattern in Figure 4.4 have three different occurrences in the pattern.

The positions of the base pattern occurring in P are represented by the coordinates of the upper-left character in P. We need to go through every position in the whole pattern P to recognize all occurrences of the base patterns. Each comparison takes $O(k)$ time and we need to go over every character in the pattern. Therefore, the

preprocessing time is $O(mk)$.

Description of the Algorithm

The algorithm works similarly to the sequential case, except that the search is now carried out in two dimensions. Specifically, the whole search process is similar to a breadth first traversal of a tree where each node of the tree represents a possible match the algorithm needs to look at. First we divide the text into blocks each of size m. Then we proceed the search separately in each block. We illustrate the process of search in the most upper-left block X in the text (see Figure 4.5).

We first find the index of the base pattern in the lower-right corner of that block. We then match the pattern against the entire block of the two-dimensional text. If there is a match, we output the coordinates of the upper-left corner of X as the result. If not, we slide the pattern to the next possible matching position where the base pattern in the lower-right corner overlaps with the block. This position is just the next pointer for the base pattern in the preprocessed shift table. Then we check if there is a match for the pattern. This process repeats until there is no further possible position to match.

We will show the complexity of this algorithm is also related to the compressibility of the text. It is not clear whether this algorithm is optimal or not since the lower bound for the two dimensional pattern matching is not known. We also define entropy of the two-dimensional text as the inverse of the optimal compression ratio of the text, i.e. $H = 1/\rho_{opt}$.

Lemma 4.3 *Assuming the text has a bounded entropy H, the probability of an arbitrary base pattern of size $k = (1 + 1/H) \log m$ appearing in an arbitrary position of the text is $\leq 1 - H^2$.*

Proof: The proof is similar to the one-dimensional case. We assume the optimal

compression ratio is $\rho_{opt} = 1/H$. Suppose not, we design a new compression scheme by the following method. For each block of size $k = (1 + 1/H) \log m$ in the text that gets a match in the patter, we use the position of the matched block (coordinates of the up-left character of the matched block) to index the block. The index takes $2 \log \sqrt{m} = \log m$ bits. The rest of the proof is similar to the proof in sequential case and the new compression scheme achieves a higher compression ratio which is a contradiction. □

Theorem 4.3 *Assuming the pattern is generated from a random source and the entropy of the text is bounded by H, the two-dimensional pattern matching algorithm takes an expect time of $O(nk(1 - H^2)/c^k)$ where H is the entropy of the text and $k = (1 + 1/H) \log n$.*

Proof: The number of expected matching in each block is $O((1 - H^2)m/c^k)$. The number of total blocks is $O(n/m)$. We assume that each match checking between a particular part of the text and the base pattern takes an expected time of $O(k)$. The time complexity thus follows. □

4.4 Parallelization of the Algorithms

The sequential pattern matching algorithms we described above can be extended to a parallel algorithm on PRAM model in a straightforward fashion.

Theorem 4.4 *The parallel one-dimensional pattern matching algorithm takes $O(\log m/p)$ expected parallel time using $O(n/m)$ processors where $p = 1 - (1 - H^2)^{(}H/(1 + H))$, n and m are the size of the text and the pattern respectively.*

Proof: We assume that the parallel model we use is PRAM CREW (concurrent-read and exclusive-write). The preprocessing of the algorithm can be done by assigning one processor to each position in the pattern. Specifically, the i-th $(1 \leq i \leq m)$

processor is assigned to the i-th position in the pattern. The processor then in parallel searchs for the matching in the pattern for the block of size $\log m$ starting from assigned position. The index table is constructed by concurrently allowing only one processor to write in any table slot at a time. In the processing stage, we divide the text into n/m intervals each of length m. We assign one processor to work independently on each interval using our sequential string matching algorithm. For example, the i-th processor works on the i-th interval of the text which is of length m. Initially, the processor aligns the pattern against the interval and examines block by block from right to left until a mismatch is found. Then the pattern is shifted to the right according to the precomputed shift table. Note that the processor now starts working on the $(i+1)$-th interval of text. This process ends until the processor finds a successful match or the pattern reaches $(i+2)$-th interval. The expected parallel running time is therefore $O(\log m/p)$. The total work bound is the same as in the sequential case. □

Theorem 4.5 *The parallel two-dimensional pattern matching algorithm takes* $O(mk(1-H^2)/c^k)$ *expected parallel time using* $O(n/m)$ *processors where* H *is the entropy of the text,* n *and* m *are the size of the text and the pattern respectively,* $k = (1+1/H)\log n$.

Proof: We assign each processor for each block and work on it simultaneously. Since the processor works independently on each block, the expected parallel time follows directly from the analysis of the sequential algorithm. The total work is the same as the sequential time. The number of the processors needed is the same as the number of the blocks which is $O(n/m)$. □

4.5 Discussion

From the design and analysis of the string matching algorithm and its extension to two-dimensional case, we observed that the analysis techniques used for data

compression algorithms, such as making assumptions on input and analyzing the performance of the algorithm based on such assumptions, are well suited for solving problems in other areas. In some cases, the assumptions on the inputs make the algorithm analysis easier to carry out and more focused on the analysis of expected running time instead of worst case time bounds. Therefore, the time complexity analysis we achieved is a closer indication of the actual performance of the algorithm compared with the analysis based on no specific assumptions of the inputs. By showing that these assumptions hold widely for practical inputs, we expect the new algorithms to have good performance in practice.

The second aspect of applying data compression technique in algorithm analysis we exploited in this chapter is that data compression algorithm can be used as a novel tool in proving some of our key theorems. For example, the proof of Lemma 4.1 uses a data compression scheme to develop a contradictive argument. This technique seems to be useful in analyzing algorithms whose performance is closely related to the compressibility or other statistical properties of the inputs. It is an interesting open problem to find an algorithm that has a complexity independent of pattern distribution. Also the special case where the complexity of the algorithm is independent on the size of alphabet remains open.

Another interesting problem is to extend our algorithm to lossy pattern matching with certain assumptions on statistical properties of the text and the pattern, perhaps combined with ideas from lossy compression schemes. Pattern matching algorithm can be *lossy* in the sense that there could be less than or equal to k mismatches in the matched text where k is a preset constant. Precisely, suppose pattern is $x_1x_2...x_m$ and the matched string from the text is $y_1y_2...y_m$, the number of i that satisfies $x_i \neq y_i$ where $1 \leq i \leq m$ must be less than or equal to k. The sequential algorithm proceeds basically the same way but we now allow matching process to continue until exactly

k mismatches have appeared. Then we shift pattern as much as possible according to the shift table.

The second category of lossy matching is the algorithms that use randomized techniques where the output is the exact match of the pattern with high likelihood. [63] gave three randomized algorithms based on fingerprinting techniques. A fingerprint of a substring in the text is defined as an unique coding of the string computed by fingerprint function. Normally a fingerprint is much shorter than the original string. The fingerprint function δ has the following property: if A and B are two strings and $\delta(A) = \delta(B)$, then $A = B$ with high likelihood. After encoding patterns and substrings of the text into fingerprints, the match is then between fingerprints which are much shorter than the original strings. Such algorithms find exact match of the pattern in the text more efficiently than the normal string comparison algorithms and require less storage space. Using the fingerprint technique, we can modify our algorithms to be randomized algorithms with high likelihood of finding a correct match. We can first encode pieces of string of length $\log m$ using fingerprint functions and then compare between the fingerprints instead of original string.

4.6 Summary

In this chapter, we studied another algorithmic problem of string matching and demonstrated several novel uses of data compression techniques in design and analysis of simple but efficient algorithms. Different from the previous study of sorting algorithm, we focus the applications of data compression techniques on the analysis of the expected time complexity of the algorithm and the proof of theorems instead of data structures and algorithmic procedures.

We presented novel string matching algorithms and their analysis for fast matching text with respect to the entropy of the text in one and two dimensional cases. We

found that the complexity of the algorithms is directly related to the compressibility of the pattern as well as the text. If we assume that the entropy of the text is bounded by a certain constant, the expected performance of our algorithm may be close to optimal due to a conjecture on the optimality of expected running time of string matching problems. We also extend our algorithm to two dimensional case and parallel algorithms. The simplicity of our algorithms makes the parallel implementation especially straightforward.

In next chapter we will extend our application of data compression techniques to image processing problems. There has been extensive research on compression algorithms on image. We propose a novel idea of speeding up computation by computing directly in compressed transform domain. Since image data can be lossily compressed well with small error, our methodology of computing directly on compressed representation of image may find useful in practice.

Chapter 5

Fast Volume Rendering in Compressed Transform Domain

In the previous two chapters, we have studied two fundamental algorithmic problems, sorting and string matching, and applied data compression techniques in various ways to design new and efficient algorithms. These problems are similar to data compression in the sense that the input data in the problem can all be viewed as a stream of characters drawn from a finite alphabet. The goals in these problems are then to either process and rearrange the text as in the sorting problem and data compression, or answer a query as in the string matching problem. The techniques of data processing used in data compression algorithms are thus directly applicable in solving these new problems as we have demonstrated in previous chapters. We also expect more algorithmic problems where the input data can be modeled similarly can use data compression techniques with little modification.

However, there are also algorithmic problems where the input data can not be modeled in the similar way. For example, in a large class of computational geometry problems, the input can be a set of points in high dimensional space or a set of lines in two dimension. Although the input can still be abstracted represented by a serial of binary numbers, the input has its meaning only in the context of the particular setting of the problem, unlike the input data in data compression. In other words, we can not always have a binary representation of the input and even if we have, the representation makes sense only after certain imposed interpretation. Therefore, it is natural to ask whether the data compression techniques can be extended to algorithms with more complex data structures.

In this chapter, we will study the problem of applying data compression techniques to image processing algorithms. For many image processing applications, the use of data compression is so pervasive that we can assume the inputs and outputs are in a compressed domain, and it is intriguing to consider doing computations on the data entirely in the compressed domain. We speed up processing by doing computations, including dot product and convolution on vectors and arrays, in a compressed transform domain. To do this, we make use of sophisticated algebraic techniques for evaluation and interpolation of sparse polynomials. We illustrate the basic methodology by applying these techniques to image processing problems, and in particular to speed up the well known splatting algorithm for volume rendering. The splatting algorithm is one of the most efficient of existing high quality volume rendering algorithms; it takes as input three dimensional volume sample data of size N^3 and outputs an $N \times N$ image in $O(N^3 f)$ time, where f is a parameter known as footprint (which often is hundreds of pixels in practice). Assuming that the original sample data and the resulting image are stored in the transform domain and can be lossily compressed by a factor ρ with small error, we show that the rendering of the image can be done entirely in the compressed transform domain in decreased time $O(\rho N^3 \log N)$. Hence we obtain a significant speedup over the splatting algorithm when $f \gg \rho \log N$. The same methodology of computing in compressed transform domain can be applied to speed up other algorithms in image processing problems.

5.1 Background

There has been extensive research on lossless and lossy image compression to represent image with less data in order to save storage space and reduce the time and cost of transmission [38, 85]. Various transform compression techniques, such as Discrete Cosine Transform, Fast Fourier Transform, wavelet transform, Huffman coding, have

been developed to design efficient image compression algorithms. For example, the well known image compression standard - JPEG, uses the Discrete Cosine Transform to encode the image by dividing the image into small blocks and encoding each block individually.

It is also well known that many image processing tasks, such as filtering and convolutions, can be speed up by computing directly in the transform domain instead of spectral domain [20].

In this chapter, we apply the novel techniques of speeding up running time of an algorithm by representing data and performing computation in *compressed* transform domain.

However, there are a number of elements of our work that draw on prior work. For example, the representation we use for compression of the transformed images is similar to the standard image encoding methods used in image compression algorithms [85]. This is an advantage, in that we can expect to obtain similar image compression and corresponding degradation as would be obtained by similar conventional transform image techniques.

There has been prior work done which reduces computation of an algorithm by taking into account of certain statistical properties of the input data, such as the bounded entropy (see [28]). Furthermore, there has been research on computing directly in the *compressed* domain. For example, [3] gave an string matching algorithm that finds the pattern string in the compressed text without decompressing the text. There has been extensive research on computation in transform domain but with the exception of searching algorithms, computing in a compressed transform domain is a novel idea.

We assume that the inputs are vectors or arrays already stored in the compressed transform domain. Similarly, we assume that the outputs in the compressed trans-

form domain, and we analyze the computational cost of our algorithm accordingly. The operations we perform on vectors and arrays include addition (which is easily done either in the uncompressed domain or the compressed domain), as well as convolution and dot product. These operations are common in image processing applications, where for example, we may perform sequences of filtering and density masking on the image domain. In prior practice one maps to and from the transform domain when applying the convolution in combination with dot product. Since the compressed transform domain is assumed to be small, ideally we would like to instead do all our computations in the compressed transform domain. We note that certain operations, such as convolution, can be performed easily if the inputs are mapped to the transform domain. On the other hand, other operations such as dot product can be done easily on the inputs in the compressed domain, but not so straightforwardly if the inputs are mapped to the transform domain (since the naive method for performing dot product needs to convert the vectors to the original domain and convert the result of dot product in the original domain back into the transform domain, which requires $O(n \log n)$ time in one dimension and $O(n^2 \log n)$ time in two dimension). Instead, we perform these non-trivial operations such as the dot product by sophisticated algorithm techniques using sparse polynomials.

For problems in the areas such as image and speech processing where the input data is known to have high lossy compression ratio with low L_2 error, our idea of computing in compressed transform domain can be applied to greatly reduce computational cost. In particular, we apply this methodology to the problem of designing fast and compact volume rendering algorithms in compressed transform space. Furthermore, we expect that this novel idea of reducing computation by computing in compressed transform domain can be applied to other image processing problems requiring sequences of filtering and density masking in the image domain.

A *volume rendering* algorithm takes as input three-dimensional arrays of voxels (3D pixels) of size $N \times N \times N$, giving the discrete input sample volume to be rendered, and outputs an $N \times N$ image rendering. Volume rendering is an ideal application of our techniques: (i) the inputs are generally very large (N often can be over $1,000$, so the volume input size is often a number of megabytes), so it is advantageous to have compressed the inputs, (ii) the volume may be viewed at many viewing angles, creating a multiplicity of output images so it is also often advantageous to also compress the outputs for subsequent viewing, and furthermore (iii) the operations required by volume rendering algorithms are similar to many other image processing applications, and in particular include operations which we may speed up by computing in the compressed transform domain.

The "splatting" algorithm by Westover [109] is perhaps the most efficient existing high quality volume rendering algorithms, and runs faster than ray casting methods by use of an approximation technique known as footprints. We present an improved volume rendering algorithm which reduces the computation of the "splatting" algorithm by computing in compressed transform domain. We will describe how to render the image efficiently by computing in compressed transform domain. Assuming that the original sample data and the resulting image are stored in the transform domain, we show that the rendering of the image can be performed in $O(\rho N^3 \log N)$ where N^3 is the size of the three dimensional sample data and ρ is the compression factor, compared to the $O(N^3 f)$ running time of the original splatting algorithm where f is the size of footprint. Our algorithm is more efficient when $f \gg \rho \log N$.

This chapter is organized as follows. In Section 2, we will describe the general problem of computing in compressed transform domain. In Section 3, we apply our method to the splatting volume rendering algorithm and give a new algorithm which performs computation in the compressed transform domain. We also present the

performance analysis of our algorithm and discuss the advantages of our algorithm. We summarize our results in Section 4.

5.2 Computing in Compressed Transform Domain

We will use *Discrete Fourier Transform* (DFT) as the transform for the presentation. Let $\omega_n = \exp(2\pi\sqrt{-1}/n)$ be the nth root of unity over the complex numbers. Let $\vec{a}$ be an n-vector $\vec{a} = (a_0, a_1, ..., a_{n-1})$. We define the vector $\vec{y} = (y_0, y_1, ..., y_{n-1})$, where $y_k = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$, to be the DFT of vector $\vec{a}$. We also write $\vec{y} = \mathrm{DFT}_n(\vec{a})$. As is well known, the DFT_n is the vector of values $(p(\omega_n^0), p(\omega_n^1), \ldots, p(\omega_n^{n-1}))$ of the polynomial $p(x) = \sum_{j=0}^{n-1} a_j x^j$ with coefficients $\vec{a}$. Using the Fast Fourier Transform algorithm (see [26]), which takes advantage of the special properties of the complex roots of unity, the DFT and inverse DFT of vector $\vec{a}$ of length n can be computed in $O(n \log n)$ time.

In practice where the input data set will most likely contain real data, an alternative transform such as *Cosine Transform* or *Fast Hartley Transform* (FHT) may be more advantageous. For example, the FHT is defined as follows:

$$H(k_x, k_y, k_z) = \frac{1}{N^3} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} \sum_{z=0}^{N-1} f(x, y, z)[\cos(2\pi \frac{xk_x + yk_y + zk_z}{N}) + \sin(2\pi \frac{xk_x + yk_y + zk_z}{N})]. \quad (5.1)$$

We will use Discrete Fourier Transform in the discussion of this chapter while the other transforms can be applied as well. We will first show the standard operations in transform domain and then describe how to execute these operations in the compressed transform domain.

5.2.1 Computing in Transform Domain

Without loss of generality, we consider the problem of computing in transform domain in one dimension. These operations are easily extendible to higher dimensions. Let $\mathcal{U}_1$ denote the untransformed original one-dimensional domain and let $\mathcal{T}_1$ denote the one-dimensional transform domain. The operations we consider are vector operations such as addition of vectors, dot product of vectors and convolution of vectors in $\mathcal{U}_1$. Let $\vec{u}$ and $\vec{v}$ be vectors of length n and the result of the operation be vector $\vec{r}$. We define these operations as follows:

addition "+" $\vec{u} + \vec{v} = \vec{r}$, $r_i = u_i + v_i$.

dot product "$\cdot$" $\vec{u} \cdot \vec{v} = \vec{r}$, $r_i = u_i v_i$.

convolution "$\star$" $\vec{u} \star \vec{v} = \vec{r}$, $r_i = \sum_{k=0}^{i} u_k v_{i-k}$.

The addition and dot product can be performed easily in $O(n)$ time in original domain. By the *Convolution Theorem* (e.g., see[26]), the convolution of two $n-$vectors can be computed by mapping the vectors to the transform domain using the DFT_{2n}, performing dot product on the transformed DFT_{2n} vectors and then computing the inverse Discrete Fourier Transform (DFT_{2n}^{-1}) of the resulting vector, both taking $O(n \log n)$ time. This is routine for signal processing applications. On the other hand, the convolution of two vectors given in the transform domain $\mathcal{T}_1$ can be computed by computing an inverse transform (e.g. an inverse DFT) of the two vectors, computing a dot product of the resulting vectors in the original domain $\mathcal{U}_1$, and then converting the result back into DFT in $\mathcal{T}_1$. In other words, the convolution operation and dot product operation are dual problems in original and transform domain. But note that conversion to and from these domains $\mathcal{U}_1, \mathcal{T}_1$ appears to be required if we wish to do combinations of dot product and convolution operations.

5.2.2 Compressed Transform Domain

To create a compressed transform domain $\mathcal{T}_1$, we need to drop data without introducing large error. A technique known to provide good performance in practice for many classes of images and speech data, is to remove high frequency terms in the Discrete Fourier Transform of the vector which are close to zero. Specifically, let $\vec{v} = (v_0, v_1, ..., v_{n-1})$ be a n-vector which consists of the coefficients of the polynomial $q(x) = \sum_{i=0}^{n-1} v_i x^i$. We encode $\vec{v}$ by Discrete Fourier Transform $s_0, s_1, ..., s_{n-1}$ where $s_i = \sum_{j=0}^{n-1} x_j \omega^{ij}$. We keep k low frequency terms, $s_0, s_1, ...s_{k-1}$. In the remaining terms, we keep s terms of highest magnitude. Let $n' = k + s \ll n$. The total storage space needed is ρn where $\rho = \frac{n'+s}{n}$ since we need to store the position indices of s terms of the highest magnitude as well. We consider the problem how to compute the addition, dot product and convolution of two vectors in the original domain $\mathcal{U}_1$ by using their corresponding compressed vectors in the compressed transform domain $\mathcal{T}_1$. The result is stored in the compressed transform domain $\mathcal{T}_1$. (Our compression scheme is similar to many one dimensional transform compression schemes in that only the high frequency (sparse) terms which contain non-vital information are dropped. The similarity is advantageous, since it indicates that we may obtain compression ratio comparable to that of these conventional transform compression methods. The main difference between our compression scheme and typical one dimensional transform compression method is that such methods may apply the transform only on blocks of consecutive subsequences of some bounded size, in part to limit computational costs. In practice, to further decrease our computational costs, we can also apply this decomposition method.)

A *sparse polynomial* is a polynomial of degree n which has s non-zero terms where $s \ll n$. The compressed transform of a vector $\vec{a}$ in the transform domain has a corresponding sparse polynomial $p(x) = \sum_{i=0}^{n-1} a_i x^i$ whose coefficients form the same

vector. In [15, 17, 65], it is shown that the evaluation and interpolation of a sparse polynomial with degree n and s terms at s arbitrary points can be accomplished in $O(s \log^2 s + \log n)$ time. Furthermore, if the evaluation or interpolation points are the s roots of unity (or more generally, at s chirp points ζ^i $(i = 0, ..., s-1)$, for some fixed complex number ζ), the cost is $O(s \log s + \log n)$.

5.2.3 Operations using Compressed Transform Domain

We consider the problem how to execute operations such as addition, convolution and dot product on vectors in the original domain $\mathcal{U}_1$ by using their corresponding representations in the compressed transform domain $\mathcal{T}_1$. Without loss of generality, we consider the operations on two n-vectors $\vec{u}$ and $\vec{v}$ in one dimension (we later will extend these operations to two dimensional arrays) using their corresponding vectors $\vec{a}$ and $\vec{b}$ in the compressed transform domain $\mathcal{T}_1$ which are encoded using the compression method described above. Let k be the number of low order coefficients in $\vec{a}$ and $\vec{b}$ which we store (that is, we always store low order coefficients $a_0, a_1, \ldots, a_{k-1}$ and $b_0, b_1, \ldots, b_{k-1}$ as dense vectors), and let s be the number of additional sparse high order terms in $\vec{a}$ and $\vec{b}$ which we store (that is, we also store the lists of coefficients $a_{i_0}, a_{i_1}, \ldots, a_{i_{s-1}}$ and $b_{j_0}, b_{j_1}, \ldots, b_{j_{s-1}}$ of highest magnitude) Thus the *uncompressed size* is the length n of the vector in the untransformed domain $\mathcal{U}_1$, and the *compressed size* is the number n' of non-zero coefficients of the vectors in the compressed transform domain $\mathcal{T}_1$. We assume that the input and output vectors are both stored in the compressed transform domain $\mathcal{T}_1$ and the computational cost is analyzed accordingly, in terms of n', where possible.

Addition The addition of two vectors $\vec{u}$ and $\vec{v}$ in the untransformed domain $\mathcal{U}_1$ can be performed by adding the corresponding terms of $\vec{a}$ and $\vec{b}$ in $\mathcal{T}_1$. Note that in the worst case, the number of non-zero terms in the resulting vector $\vec{z}$ may become

$k+2s$ where k is the number of dense frequency terms and s is the number of sparse terms in vector $\vec{a}$ or $\vec{a}$. We keep the k dense terms and choose s sparse terms of highest magnitude out of the possible $2n$ sparse terms. Thus the addition in $\mathcal{U}_1$ will be computed in the compressed transform domain $\mathcal{T}_1$ which takes $O(n')$ time. We denote the approximated computation of addition by $ApproxAdd()$.

Convolution Given vectors $\vec{u}$ and $\vec{v}$ in untransformed domain $\mathcal{U}_1$, recall that their convolution can be computed by an inverse transform of the dot product of their compressed transform vectors $\vec{a}$ and $\vec{b}$ in the transform domain $\mathcal{T}_1$. We first perform pointwise multiplication of the corresponding terms of $\vec{a}$ and $\vec{b}$ in $\mathcal{T}_1$. Let the resulting vector be $\vec{r}$ of length n such that $r_i = a_i b_i$. The number of non-zero elements in $\vec{r}$ is equal to or less than n'. Thus the convolution in $\mathcal{U}_1$ will be approximately computed in compressed transform domain $\mathcal{T}_1$ in $O(n')$ time. We denote the approximated computation of convolution by $ApproxConv()$.

Dot Product The dot product of two given $n-$vectors $\vec{u}$ and $\vec{v}$ in the untransformed domain $\mathcal{U}_1$ is a vector whose ith element is the sum of the i elements of u and v. To compute the dot product of two vectors $\vec{u}$ and $\vec{v}$, we need to compute the convolution of the two corresponding vectors $\vec{a}$ and $\vec{b}$ in the compressed transform domain $\mathcal{T}_1$. Let $p_a(x) = \sum_{i=0}^{n-1} a_i x^i$ be the corresponding sparse polynomial of $\vec{a}$ and $p_b(x) = \sum_{i=0}^{n-1} b_i x^i$ the sparse polynomial of $\vec{b}$. We evaluate $p_a(x)$ at $2n'$ evenly spaced roots of unity. Specifically, let $\omega_{2n'}$ be the $2n'$-th root of unity. We will evaluate $p_a(x)$ at $2n'$ points: $\omega_{2n'}^{-i}$ where $0 \leq i \leq 2n'-1$. We interpolate a sparse polynomial $h(x)$ of $2n'$ non-zero terms and degree $2n$ such that $h(x_i) = p_a(x_i)p_b(x_i)$ for each $x_i = \omega_{2n'}^{-i}$, where $0 \leq i \leq 2n'-1$. By this sparse polynomial interpolation and the convolution theorem, the coefficients of h can be shown to be the convolution of the two vectors $\vec{a}$ and $\vec{b}$ in the compressed transform domain $\mathcal{T}_1$. To insure the required

compression, we keep all coefficients of h of degree $< k$ and keep only s of the coefficients of h of largest magnitude of degree $\geq k$. We use this resulting vector $\hat{h}$ to approximate the DFT of the dot product of the two vectors $\vec{u}$ and $\vec{v}$, as required. Thus, by use of known sparse polynomial evaluation and interpolation algorithms (see [15, 17, 65]), the total cost of computing the dot product is $O(n' \log n' + \log n)$. We denote the approximated computation of dot product in the transform domain by $ApproxDotProd()$.

We summarize the above results in the following theorem:

Theorem 5.1 *The following operations on vectors in the untransformed domain $\mathcal{U}_1$ can be approximately performed in the compressed transform domain $\mathcal{T}_1$: (i) the operations of addition and convolution, in $O(n')$ time and (ii) the dot product, in $O(n' \log n' + \log n)$ time, where n is the uncompressed size (the length of the vector in the untransformed domain $\mathcal{U}_1$) and n' is the compressed size (the number of non-zero coefficients of the vectors in the compressed transform domain $\mathcal{T}_1$).*

Note that the error and degradation due to lossy compression depends on the degree of compression factor just as in standard lossy compression algorithms. The above computations may not be appropriate for general algebraic computations if vectors can not be lossily compressed with small error. However, in many 1D applications such as speech processing the input data is known for a large class of inputs to have excellent lossy compressibility, with small error, using similar compression techniques. So for these applications, our techniques for computing in the compressed transform domain can be applied to produce a reduction in the amount of computation for the solution of problem with limited error. We next extend the above method for computation in the compressed transform domain to 2D.

5.2.4 Extension to Two-dimensions

The operations in compressed transform domain are readily extendible to two dimensions. We will summarize this extension in the following theorem:

Theorem 5.2 *The following operations on arrays in the untransformed domain $\mathcal{U}_2$ can be approximately performed in the compressed transform domain $\mathcal{T}_2$: (i) the operations of addition and convolution in cost $O(n')$ time and (ii) the dot product with cost $O(n' \log n' + \log n)$ time, where the arrays are of size $n \times n$ in the 2 dimensional untransformed domain $\mathcal{U}_2$ is and n' is the number of non-zero coefficients of the arrays in the 2 dimensional compressed transform domain $\mathcal{T}_2$.*

Note that (as in the 1D case), the error and degradation due to lossy compression depends on the degree of compression factor just as in standard lossy compression algorithms. In our applications of choice, such as image processing in $2D$, the input data is known for a large class of inputs to have excellent lossy compressibility, with small error, using similar compression techniques. We apply this method to the volume rendering problem where the resulting image is already known to be highly compressible, and so we can greatly reduce the amount of computation by avoiding costly operations in the original domain by computing directly using the compressed forms of the images in the transform domain. In the following section we apply this generalized 2D method to the volume rendering problem where the resulting image is already known to be highly compressible. So we can greatly reduce the amount of computation by avoiding costly operations in the original domain by computing directly using the compressed forms of the images in the transform domain.

The two dimensional Discrete Fourier Transform 2D DFT_n of an array in the original domain can be obtained by computing one-dimensional DFT by rows and then computing one-dimensional DFT for the resulting array's columns [85]. Let

$\omega_{2n} = \exp(\pi\sqrt{-1}/n)$ be the $2n$th root of unity over the complex numbers. Let M be an $n \times n$ array such that $M = (m_{i,j})$. Consider the bivariate polynomial $Q(x,y) = \sum_{i,j=0}^{n-1} m_{ij}x^i y^j$ with variables x and y. Then the 2D DFT_{2n} of M is the $2n \times 2n$ array A where $A_{i,j} = Q(\omega_{2n}^i, \omega_{2n}^j)$ that is $A_{i,j}$ is the evaluation of $Q(x,y)$ at $x = \omega_{2n}^i$ and $y = \omega_{2n}^j$. Using Fast Fourier Transform (see [26]), the DFT and inverse DFT of an $n \times n$ array can be computed in $O(n^2 \log n)$ time.

We compress the transformed array A using a method similar to the one-dimensional case. We assume, w.l.o.g., that k is a perfect square and keep a dense $\sqrt{k} \times \sqrt{k}$ subarray of elements of the transformed array (those terms that are in both first $\sqrt{k}$ rows and first $\sqrt{k}$ columns). These coefficients correspond to the low frequency terms in the original image. We keep s remaining entries in the array of highest magnitude. Let $P(x,y) = \sum_{i,j=0}^{n-1} a_{ij}x^i y^j$ be the corresponding sparse bivariate polynomial of variables x, y. Let $n' = s + k$. We assume w.l.o.g. $2n'$ is a perfect square. The total storage space needed is ρn^2 where $\rho = \frac{n'+s}{n^2}$, since we need to store the position indices of s terms of the highest magnitude as well. (Our compression scheme is similar to the JPEG image compression scheme in that only the high frequency (sparse) terms which contains non-vital information are compressed. The similarity is advantageous, since it indicates that we may obtain compression ratio comparable to to that of conventional methods such as JPEG. The main difference between our compression scheme and JPEG compression is that JPEG and other related image transform compression schemes is that they apply the transform only on blocks of consecutive subsubarrays of some bounded size. In practice, to further decrease our computational costs, we can also apply this decomposition method, which limits computational costs.)

A *sparse bivariate polynomial* is a polynomial of degree n of two variables, say x, y which has s non-zero terms where $s \ll n$. The compressed transform array A

in the transform domain has a corresponding sparse polynomial $p(x) = \sum_{i=0}^{n-1} a_i x^i$ whose coefficients form the same array. It is known (see [15, 17, 65]) that the results for evaluation and interpolation of a sparse univariate polynomial with degree n and s terms at s arbitrary points, hold also for bivariate polynomials, so again evaluation and interpolation can be done in $O(s \log^2 s + \log n)$ time. Furthermore, the cost again decreases to $O(s \log s + \log n)$ if the evaluation or interpolation points are the s roots of unity or s chirp points.

We consider the problem how to execute operations such as addition, convolution and dot product on arrays in the original domain $\mathcal{U}_2$ by using their corresponding representations in a two dimensional compressed transform domain $\mathcal{T}_2$. Without loss of generality, we consider the operations on two $n \times n$ arrays M, M' using their corresponding arrays A, A' in the 2D compressed transform domain $\mathcal{T}_2$, which are encoded using the compression method described above. Let $n' = s + k$ be the total number of coefficients stored by A, and A', where s is number of sparse high order terms stored and the dense subarray of low frequency terms stored is of size $\sqrt{k} \times \sqrt{k}$. We define the *dot product* of two arrays M, M' to be the array M'' such that $M''_{i,j} = M_{i,j} M'_{i,j}$ (note that this definition differs from some other definitions, but is convenient for our applications). The result is to be stored in the 2D compressed transform domain $\mathcal{T}_2$. Again, the approximate addition and convolution of two $n \times n$ compressed arrays can be performed easily in $O(n')$ time in the compressed transform domain $\mathcal{T}$, just as previously described for one-dimensional vectors. By the *2D Convolution Theorem*, the convolution of two $n \times n$ arrays M, M' can be computed in the transform domain using the 2D DFT_{2n}, performing dot product on the transformed DFT_{2n} arrays A, A' and then computing the inverse 2D Discrete Fourier Transform (DFT_{2n}^{-1}) of the resulting array, both taking $O(n^2 \log n)$ time. This is routine for image processing applications. In our applications, we need not apply the forward or inverse

2D Discrete Fourier Transform, and instead simply perform dot product on the prior transformed DFT_{2n} arrays A, A' in time $O(n')$ proportional to the number of stored coefficients n'. We again denote the approximated computation of addition and convolution in the transform domain by $ApproxAdd()$ and $ApproxConv()$. Again, the convolution operation and dot product operation are dual problems in original and transform domain. The dot product of two $n \times n$ arrays M, M' convolution of two arrays A, A' given in the transform domain $\mathcal{T}_2$ can be computed by computing an inverse 2 D DFT of the two arrays, computing a dot product of the resulting arrays A, A' in the original domain $\mathcal{U}_2$, and then converting the result back into the 2D DFT_{2n} in $\mathcal{T}_2$. Again, we appear to require the conversion to and from these domains $\mathcal{U}_2, \mathcal{T}_2$ if we wish to do combinations of dot product and convolution operations, but we again avoid this by use of sparse polynomial evaluation and interpolation. Let $p_A(x,y) = \sum_{i,j=0}^{n-1} A_{i,j} x^i y^j$ be the corresponding sparse polynomial of array A and $p_{A'}(x,y) = \sum_{i,j=0}^{n-1} A'_{i,j} x^i y^j$ the sparse polynomial of array A'. Let $\omega_{2n'}$ be the $2n'$-th root of unity. We evaluate $p_A(x,y)$ at $2n'$ evaluation points; that is at all $x_i = \omega_{2n'}^i$ and $y_j = \omega_{2n'}^j$ for each $0 \leq i,j \leq \sqrt{2n'} - 1$. We interpolate a sparse bivariate polynomial $h(x,y)$ of $2n'$ non-zero terms and degree $\leq 2n'$ in each of the variables x, y such that $h(x_i, y_j) = p_a(x_i, y_j) p_b(x_i, y_j)$ for each $x_i = \omega_{2n'}^{-i}$, and each $y_j = \omega_{2n'}^{-j}$ where $0 \leq i,j \leq \sqrt{2n'} - 1$. By this sparse polynomial interpolation and the convolution theorem, the coefficients of h can be shown to be the convolution of the two arrays A, A' in the compressed transform domain $\mathcal{T}_2$. To insure the required compression, we keep all coefficients of h of degree $< \sqrt{k}$ in each of x, y and keep only s of the coefficients of h of largest magnitude of degree $\geq \sqrt{k}$ in each of x, y. We use this resulting vector $\hat{h}$ to approximate the DFT of the dot product of the two arrays M and M', as required. Thus, by use of known sparse bivariate polynomial evaluation and interpolation algorithms (see [15, 17, 65]), the total cost of computing the dot

product is $O(n' \log n' + \log n)$. We again denote the approximated computation of dot product in the transform domain by $ApproxDotProd()$.

5.3 Application to Volume Rendering Problem

We apply our computation methods described above to volume rendering problem. We first give a detailed description of the volume rendering problem and splatting algorithm. We then present our application of computation in compressed transform domain to the splatting algorithm in this section.

5.3.1 Volume Rendering Algorithms

In recent years, scientific visualization has become a fast growing area in image processing which is concerned with various techniques to help scientists and engineers to extract meaningful information from large set of data from simulations and experiments. The volume rendering problem is to directly display image of scalar and vector data samples defined in three or higher dimensions. It is one of the most actively researched subfields of scientific visualization. Some of the most commonly used direct volume rendering algorithms are ray casting [69, 99], splatting [109] and volume shearing [37, 71]. Among these algorithms, the splatting algorithm is the most efficient. However, due to the large size of the input data set, it is important to further decrease the amount of computation of splatting algorithms.

There are two primary approaches to solving the volume rendering problem: (i) the feed-backward methods that map the image space onto the data set, and (ii) the feed-forward methods that map each volume element onto the image plane. The feed-forward methods are more favorable than the feed-backward methods because in feed-forward methods each data sample only needs to know about a small surrounding neighborhood of other samples, not the whole data set as in the feed-backward

methods. This makes the feed-forward methods readily to be implemented in parallel environment.

Forward mapping algorithms using the feed-forward methods normally consist of four steps: reconstruction, transformation, shading and resampling. The image renderer reconstructs the continuous volume function from the input samples by convolving the samples with a carefully chosen reconstruction filter kernel. The volume function is then transformed into image space and the transformed data is shaded to reflect individual sample's contribution to the final image. Finally the sampling process generates a regular collection of discrete data values from a continuous signal.

The processes of sampling and reconstruction are central to the volume rendering process because the deviation from the ideal case in both processes causes errors in the final results. Furthermore, most of the computation occurs in the reconstruction process. Therefore it is desirable to design a volume rendering algorithm that speeds up the reconstruction process by reducing the computation time while minimizing the amount of errors caused by different sources.

5.3.2 Splatting Algorithm

Westover [109] presented a feed-forward volume rendering algorithm named "splatting". The name is derived from the description of the algorithm where each individual sample contributes to the overall image one-by-one, a process similar to throwing snowballs at a thick wall making "splat" sounds. In the actual algorithm, the term "splat" refers to the process of determining a sample's image-space footprint on the image plane, and adding the sample's effect over that footprint to the image.

In the splatting algorithm, the sample is treated as a reflective, light-emitting, semi-transparent cloud. The sample is first *classified* to determine the discrete values for the primary properties that represent the sample. Then it is *illuminated* using

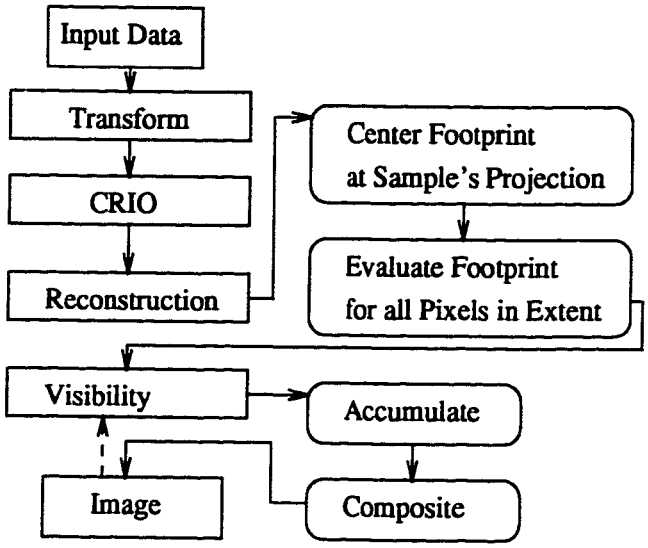

Figure 5.1: Block Diagram of Splatting Algorithm.

certain illumination model. The process of classifying a sample and applying an illumination model to calculate the sample's illumination effects is called the *CRIO* process.

The splatting algorithm consists of four main parts: transforming, CRIO, reconstruction and visibility (Figure 5.1). The transformation process converts the input tuple's mesh space (i, j, k) into an image space (x, y, z). The renderer then runs the CRIO process on the image-space tuple to generate a CRIO tuple for each data sample which contains information of color, opacity and image coordinates for this sample. Only the data sample whose opacity is not zero is passed to the reconstruction process. The reconstruction process determines the image-space contribution of each CRIO tuple to produce a single splat tuple. Finally the renderer combines the splat tuples to form the image using certain visibility rule. The visibility rules are different for front-to-back and back-to-front traversals, but the rules are equivalent.

Formally, let I_{xy}^{ℓ} denote the output intensity of image at point (x, y) after ℓ-th iteration, let F_{xy}^{ℓ} denote the intensity of data sample at point (x, y, ℓ) calculated in the reconstruction process, let A_{xy}^{ℓ} denote the output opacity of image at point

$< x, y >$ after ℓ-th iteration, and let D^{ℓ}_{xy} denote the opacity of data sample at point (x, y, ℓ) calculated in the reconstruction process using the footprint function. The algorithm accumulates the intensity I_{xy} at image point (x, y) by adding contribution F^{ℓ}_{xy} of each data sample at coordinates (x, y, ℓ). For example, the intensity that a data sample at coordinates (x, y, ℓ) contributes is computed by the amount of light emitting from the data sample $F^{\ell}_{xy} \times D^{\ell}_{xy}$ times a factor of $1 - A^{\ell-1}_{xy}$ (the portion that is not blocked by current opacity of image point (x, y)). The opacity at image point (x, y) is increased by the amount of light that goes through by the previous opacity $1 - A^{\ell-1}_{xy}$ but is blocked because of the opacity of the new data sample (D^{ℓ}_{xy}). The formulae for the back-to-front traversal can be explained similarly.

For a front-to-back traversal, the algorithm first initialize the intensity and opacity arrays of image and then compute the output intensity and opacity incrementally as follows:

Splatting Volume Rendering

for $x = 1...n$; **for** $y = 1...n$ **do**

$$I^0_{xy} = 0, A^0_{xy} = 0$$

for $\ell = 1...n$ **do**

for $x = 1...n$; **for** $y = 1...n$ **compute**

$$I^{\ell}_{xy} = I^{\ell-1}_{xy} + ((1 - A^{\ell-1}_{xy}) \times (F^{\ell}_{xy} \times D^{\ell}_{xy}))$$

$$A^{\ell}_{xy} = A^{\ell-1}_{xy} + ((1 - A^{\ell-1}_{xy}) \times D^{\ell}_{xy})$$

Output I^n, A^n .

Similarly, for a back-to-front traversal, the algorithm works as follows:

for $x = 1...n$; **for** $y = 1...n$ **do**

$$I^{n+1}_{xy} = 0, A^{n+1}_{xy} = 0$$

for $\ell = n, n-1...1$ **do**

for $x = 1...n$; **for** $y = 1...n$ **compute**

$$I^{\ell}_{xy} = ((1 - A^{\ell+1}_{xy}) \times I^{\ell+1}_{xy}) + (F^{\ell}_{xy} \times D^{\ell}_{xy})$$

$$A_{xy}^{\ell} = ((1 - A_{xy}^{\ell+1}) \times D_{xy}^{\ell}) \times A_{xy}^{\ell+1})$$

Output I^1, A^1.

Among these processes, the reconstruction part where the renderer must determine each sample's contribution to the final image is most compute-intensive. The reconstruction is normally performed by convoluting the sample function with the kernel function. The kernel function is a discrete array is carefully chosen to balance the tradeoff between the amount of computation and the image quality. For example, a high-quality and computationally expensive kernel is the one using the Gaussian density function ϕ which is defined by

$$\phi(x) = e^{-\frac{x^2}{2\sigma^2}} \tag{5.2}$$

Instead, in the reconstruction process of the splatting algorithm, a *footprint table* is built for each data sample to spread the sample's energy onto the image plane. Since the footprint function is constant for all input samples, the renderer builds a footprint table at the beginning of the reconstruction process and then uses it for every sample. One easy way to build the footprint table is to determine the image-space extend of the projection of the reconstruction kernel and then select a sub-pixel sampling rate. However, the renderer must still integrate the reconstruction kernel to build the footprint table once per image. An alternative way is to model the result of the integration with some simple function, for example, a Gaussian, since the result of integrating one dimension of a three-dimensional Gaussian is still a Gaussian. This method evaluates the simple function at the table entries without integration. The kernel needs to be truncated in this case since the extent of the kernel is infinite.

Formally, let $f()$ be the input samples, $h()$ be the reconstruction kernel, and $g()$ be the reconstructed signal, the volume reconstruction equation for a discrete input,

$f(i,j,k)$, is

$$g(x,y,z) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} f(i,j,k)h(x-i,y-j,z-k) \tag{5.3}$$

The footprint function centered at the origin of three dimensional space is defined as

$$f(x,y) = \int_{-\infty}^{\infty} h(x,y,z)dz. \tag{5.4}$$

Thus the footprint function defined as above can be viewed as a two-dimensional image-space projection of the reconstruction kernel. The footprint table is built only once at the beginning of each rendering process and the renderer uses the same footprint table for every sample.

After the contribution of each sample to the image has been determined, for each pixel in the footprint, the renderer weights the sample by the footprint value and composite the sample's color and opacity into the *accumulation buffer*. The compositing can be accomplished by either front-to-back or back-to-front traversal and a *sheet buffer* is used to correctly calculating the value of a reconstructed point that lies between samples with overlapping kernels. After the image is constructed, the user has the freedom to interactively change viewing parameters and the renderer regenerates image by recomputing the footprint table.

5.3.3 Volume Rendering in Compressed Fourier Transform Domain

In our volume rendering algorithm, the original image data sample and the footprint are all stored in the compressed transform domain $\mathcal{T}_2$. The outputs will also be stored in the compressed transform domain $\mathcal{T}_2$. In the reconstruction process of our algorithm, we consider the input sample data as a 3-D $N \times N \times N$ array s^{ℓ}_{xy} which

is stored in compressed transform domain $\mathcal{T}_2$. Similarly, the footprint function f_{xy} is stored in the compressed transform domain $\mathcal{T}_2$. The approximate convolution $\hat{F}$ of the data sample and the footprint in the spatial domain can be performed as dot product in the transform domain $\mathcal{T}_2$, and stored in this domain. Specifically, $\hat{F}$ can be computed using operation $ApproxConv(s, f)$ in the compressed transform domain $\mathcal{T}_2$ as described in Section 2, where s is the image data sample, and f is the footprint function. The total computation time for this approximate convolution is $O(\rho N^3)$ where ρ is the compression factor.

The key operations in the image rendering process are addition and dot product of two-dimensional arrays in the uncompressed domain $\mathcal{U}_2$. Therefore, the image rendering in the splatting algorithm can also be performed in compressed transform domain $\mathcal{T}_2$ using the 2-D array operations. In our algorithm, we assume that the inputs are the intensity arrays and opacity arrays stored in compressed transform domain $\mathcal{T}_2$. We denote the intensity arrays by $\hat{F}$ which is a compressed representation in transform domain $\mathcal{T}_2$ of the intensity arrays F in the untransformed domain $\mathcal{U}_2$. Similarly, we denote the opacity arrays by $\hat{D}$ which is a compressed representation in $\mathcal{T}_2$ of the intensity arrays D in $\mathcal{U}_2$. We let $\hat{I}^\ell$ be the output image intensity at the ℓ-th level which is in compressed transform domain $\mathcal{T}_2$. Also we let $\hat{A}^\ell$ be the output opacity of image at the ℓ-th level which is in compressed transform domain $\mathcal{T}_2$.

We compute the operations (addition, dot product and convolution) using two-dimensional extensions of the approximated computations described in Section 2. We assume that the input and output are both stored in compressed transform domain $\mathcal{T}_2$ and we analyze the computational cost of our algorithm accordingly. Applying a front-to-back traversal similar to the splatting algorithm, our rendering algorithm for front-to-back traversal in the compressed transform domain is the following:

Volume Rendering in Compressed Transform Domain

$\hat{I}^0 = 0, \hat{A}^0 = 0$

for $\ell = 1...n$; **compute**

$$\hat{I}^{\ell} = ApproxAdd(\hat{I}^{\ell-1}, ApproxDotProd(1 - \hat{A}^{\ell-1}, ApproxDotProd(\hat{F}^{\ell}, \hat{D}^{\ell})))$$

$$\hat{A}^{\ell} = ApproxAdd(\hat{A}^{\ell-1}, ApproxDotProd(1 - \hat{A}^{\ell-1}, \hat{D}^{\ell}))$$

Output $\hat{I}^n, \hat{A}^n$.

Also, applying a back-to-front traversal similar to the splatting algorithm, our rendering algorithm for back-to-front traversal in the compressed transform domain is as follows:

$\hat{I}^{n+1} = 0, \hat{A}^{n+1} = 0$

for $\ell = n, n-1...1$; **compute**

$$\hat{I}^{\ell} = ApproxAdd(ApproxDotProd(1 - \hat{A}^{\ell+1}, \hat{I}^{\ell+1}), ApproxDotProd(\hat{F}^{\ell}, \hat{D}^{\ell}))$$

$$\hat{A}^{\ell} = ApproxDotProd(ApproxDotProd(1 - \hat{A}^{\ell+1}, \hat{D}^{\ell}), \hat{A}^{\ell+1})$$

Output $\hat{I}^1, \hat{A}^1$.

5.3.4 Performance Analysis

We consider the performance of our algorithm for volume rendering on a given $N \times N \times N$ three-dimensional data array. In the original splatting algorithm, the dominant part of the time complexity is the reconstruction phase. Let f be the size of the footprint (normally say 15×15 pixels) and P_I (a constant say $1/2$) be the frequency of the non-zero intensity (non-transparent) volume pixels which contributes to the final image. Since the integration is performed along all three dimensions, the convolution of the sample function and the kernel takes time $O(N^3 f P_I)$. Because P_I is a constant, the time complexity is $O(N^3 f)$. The size of the footprint f is the major factor in the tradeoff between the computation time and the resulting image quality. To demand a higher image quality, the corresponding larger size of the footprint may cause the computation to be too expensive.

In the preprocessing stage of our algorithm, the $N \times N \times N$ spatial data array and the footprint function are all transformed into transform domain using Fast Fourier Transform in 3-D which takes $O(N^3 \log N)$ time and compressed using the method described in Section 2. This preprocessing needs to be done only once. By computing in compressed transform domain, the computation of the reconstruction (convolution) stage can be reduced to $O(\rho N^3)$ where ρ is the compression factor. In the final rendering process, the time complexity is $O(\rho N^3 \log N)$ where ρ is the compression factor. Therefore, the time complexity for our algorithm is $O(\rho N^3 \log N)$, given this preprocessing.

The volume rendering algorithm using compressed transform domain we presented has several advantages over the original splatting algorithm. Because our algorithm works in compressed transform domain instead of the original spatial domain, the reconstruction process is easier to visualize. The convolution in the spatial domain corresponds to a dot product in the frequency domain. By transforming the sample data into Fourier transform, we avoid the computational cost of convolution.

Our volume rendering algorithm takes $O(\rho N^3 \log N)$ time while the original splatting algorithm runs in $O(N^3 f)$ where f is the size (number of non-zero terms) of footprint. In practice, the compression factor ρ is typically 1/30 because image data can be lossily compressed well. Since the typical image size is 1000×1000 pixels, $\log N$ is approximately 10. The footprint size is normally around 200 [109]. There are constant factors in the complexities of both the splatting algorithm and our algorithm but these constant factors are similar, say about 5 to 10. Therefore, our algorithm may improve the running time of splatting algorithm by a significant factor in practical use. Furthermore, the preprocessing is only done once for any given input data set. The transformed sample data is used for subsequent repeated image construction. Since each individual image rendering is very fast, it allows the user to

change parameters of the rendering algorithm and view the result after a short time. Therefore, the progressive refinement of the final image is a natural consequence of the transform method.

By using the compressed transform representation, the renderer can build preview images generated by computing a highly compressed transform representation of the sample data in the frequency domain. The computation time decreases as the image data becomes more compressed. When a researcher uses the renderer in an interactive way and wants to get feedbacks whenever any viewing parameters change, the renderer can provide an image, which is not necessary of high quality but good enough to reflect the parameter changes, by choosing a proper compression factor for the compressed transform representation. This preview update in our algorithm is much faster than the original splatting algorithm where the convolution has to be computed expensively whenever any view parameters change.

5.4 Summary

We proposed a new computation method using compressed representation of inputs in transform domain. We applied this method to the splatting volume rendering problem and designed an improved algorithm which saves computation time by performing operations in compressed transform domain. This new method can also provide similar speed up when view parameters are changed. In practical implementation, we suggest that the DCT or FHT is used instead of FFT to further reduce storage requirement and improve running time. We expect that the same methodology of computing in compressed transform domain can be applied to speed up other algorithms in areas such as image and speech processing.

In the next chapter, we will study the problem of extending the input data type to more complicated ones and applying data compression techniques to design efficient

algorithms for these data types. We will focus on the problem of applying techniques in data compression algorithm to design compression algorithms for trees and undirected graphs. We will discuss the applicability of certain known techniques in data compression and new problems encountered that demand novel solutions.

Chapter 6

Efficient Lossless Compression of Trees and Graphs

In this chapter, we will study the problem of compressing trees and undirected graphs using techniques similar to those used in data compression algorithms. This is a relatively novel problem and we proceed by formulating the problem using terminology similar to data compression algorithms and giving compression algorithms utilizing compression techniques found in compression algorithms of binary strings. We present a new algorithm that compresses trees using a parsing scheme similar to LZ compression. We also extend the algorithm to compress undirected graphs.

6.1 Motivation and Background

Most of the data compression algorithms regard the input as a sequence of binary numbers and represent the compressed data also as a binary sequence. However, in many areas such as programming language (e.g. LISP and C)and compiler design, it is more desirable to have a compression algorithm which compresses a data structure which is not a binary sequence and keeps similar data structure in compressed form as the original data. In this chapter, we extend our work to more general data types by studying the problem of compressing tree and graphs into a similar but smaller form so that many properties of the original data structure are preserved in the compressed form. Based on the compressed data structure, queries such as matching and searching can be answered using existed techniques for solving similar problems in the original data structure.

We choose trees and graphs as the data types to be studied because they are

commonly used in many areas of algorithm design. Undirected and directed graph are used in representing data structures in programming languages. For example, directed graph is used in logic programming for expressing recursive relations. In Holm's system [58], recursively typed languages are represented by directed graphs. Thus the task of checking whether a given recursive type exists in the data structure is reduced to a pattern matching problem on the directed graph. In addition to reducing storage space, a compressed graph also has the benefit of searching for a pattern match more efficiently than in the original graph. Furthermore, the techniques and algorithms used for searching in normal graphs can be used in searching the compressed graph which has a similar structure.

We first review some terminologies that are used in design and analysis of our compression algorithm for trees and undirected graphs.

6.1.1 Tree and Graph Data Structure

We consider data structures of trees and undirected graph in this chapter. Without loss of generality, we focus our study on *ordered and rooted* binary trees where it does not matter if a leaf is a right child or a left child and use the term binary tree to refer to this specific type of binary tree. The same techniques used for this type of tree can be extended to trees which has more than two children for each node. The number of the vertices $|V|$ and the number of edges $|E|$ in the binary tree are related such that $|V| = |E| + 1$. For a root binary tree structure $T = \langle V, E \rangle$ where V is a set of vertices and E is a set of edges each connecting two vertices, the standard representation of T is to use pointers to represent an edge $e \in E$ that connects two nodes v_1 to v_2 (v_2 then is a child of v_1)[26]. In other words, there is an ordered pair of pointers representing the two children associated with each node. Let n be the number of vertices in the binary tree T, the total number of bits needed to represent

T is therefore $n \log n$ where $\log n$ bits are used to label individual vertex.

An undirected graph $G = \langle V, E \rangle$ consists of a set of *vertices* V of size n, and a set of *edges* E of size m. Each edge is an unordered pair (v, w) of disjoint vertices v and w. The standard representation of G is to use an unordered linked list to represent the set of edges associated with each node. The total number of bits required to represent a given undirected graph is therefore also $O((n + m) \log n)$.

However, the representation described above is not optimal as it can be demonstrated in the case of the binary tree. Let T denote a binary tree described above. We can use the following encoding method to reduce the storage space for T to $O(n)$ where n is the number of vertices. Recall that depth first search (DFS) of T creates a unique traversal of vertices of T. We define an alphabet $\mathcal{A}$ containing three elements $\{0, 1, \#\}$. Each element represents a certain "move" in depth first search of T. Specifically, we let 0 represent a move going down to the left child of the current vertex, 1 denote a move going down to the right child of the current vertex, and $\#$ represent a move going upwards to the parent of the current vertex. We can encode T easily by following the traversal path of depth first search. Also we can reconstruct T just by simulating a depth first search if we are given an encoded sequence of T. Since the depth first search visits each vertex at most twice, the total size of the encoded sequence for T is $O(n)$ where n is the number of vertices in T. This encoding scheme extends to general trees as well. Despite the efficiency achieved by this encoding scheme, the compressed character sequence does not resemble the original binary tree structure and thus makes it difficult to analyze the properties of the original binary based on the compressed sequence. Therefore, for the coherence of our presentation, we still regard $O(n \log n)$ as the size of *standard* representation of a binary tree while $O((n + m) \log n)$ for a general undirected graph throughout this chapter, although it is clear that more efficient encoding is achievable.

The basic algorithmic idea we will use to compress the above data structures into similar structures with smaller size comes from the adaptive dictionary construction in LZW compression scheme. Recall from the second chapter, data compression algorithms are designed to encode a given input, normally a stream of characters, into a more compact form so that the compress form requires less space to store and less time to transmit. Lempel-Ziv compression (LZ) [113, 82] is a family of compression algorithms that use a pointers to encode the repeated phrases in the input text. LZ compression parses the original sequence into subsequences by longest prefix match and it is proven that LZ compression achieves optimal compression ratio when the size of the input goes to infinity. LZW compression is a popular version of LZ compression which uses dictionary encoding and has been implemented as system compression routines in practice.

The main data structure used in LZW is a trie which implements the dictionary. For simplicity purposes, we can view the trie as a multiway tree although the actual efficient implementation of the trie involves hash tables instead of the representation of a linked list.

The construction of the LZW tree is as follows. We start the dictionary as an empty set. At each step, we find the longest prefix match of the remaining input to the current LZW tree. We then add the new node into the dictionary which contains the longest prefix match plus the first character of the remaining input string. This partition of the original input creats a unique parsing where no two phrases are identical.

Recall from Chapter 2 that the trie used in LZW compression has statistical properties which are critical in achieving optimality of compression. All internal nodes in trie have indices in the LZW dictionary. The shape of the trie is an approximation of the actual probability distribution of the original input source. Any random string

has approximately equal probability to be indexed to any of the leaves in the current trie (see Chapter 2 for probability analysis). Intuitively, each time we update the LZW trie by adding a new leaf, every prefix of that leaf which are already stored in the dictionary appears one more time. And all the leaves in the current trie have just been visited once. The LZW compression achieves optimality (i.e. the compression ratio limits to the inverse of the entropy of the source) because it updates the current trie by the new phrase parsed from the input string and the trie grows correspondingly to the probability distribution of the input source. In the following tree and graph compression algorithms, we employ a parsing scheme similar to LZW parsing to build an adaptive dictionary containing unique subtrees by parsing the tree (and the DFS-tree in graph case) into subtrees.

We define a number of stochastic properties of a binary tree before we give the parsing scheme. For any given binary tree T, let V be the set of nodes (vertices) and E the set of edges contained in T. Also let n denote the size of T (the size of E is $n-1$).

We first consider the following way of generating a random binary tree with n leaves. We first expand the root node by adding two leaves. And then we expand the two leaves at random. At time k, we choose one of the $k-1$ leaves according to a uniform distribution and expand it until n leaves have been generated. We call the binary tree T_n we obtained a *randomly generated* tree. Note that every possible binary tree can be generated using this method. We can estimate the entropy $H(T_n)$ (the minimum number of bits per node needed to represent a n-node binary tree T_n using this expanding method) by writing out the recurrence relation (see page 73 [34]). The entropy $H(T_n)$ grows linearly with n. In other words, there exists a constant $c > 0$ such that the number of all possible randomly generated binary trees with n leaves is less than 2^{cn}. A randomly generated tree with n leaves has less than

n internal nodes. Therefore the number of all possible binary trees with n nodes (counting both internal nodes and leaves) is less $2^{cn/2}$.

In our tree compression algorithm, we do not use the assumption of uniform distribution. Instead, we assume that the binary tree is generated by a stationary and ergodic source. We then use breadth first search (BFS) trees to parse the original tree into subtrees. Let D_n denote the set of all BFS trees each containing n nodes. Note that D_n is only a subset of all trees with n nodes. Let $c_1 = 2^{c/2}$, we have the following lemma:

Lemma 6.1 *The number of distinct binary trees containing n nodes is less than* $c_1 2^n$ *where* c_1 *is a constant.*

The randomly generated tree we just described is an example of trees generated by a stochastic process. In general, we use $\mathcal{X} = \{X_1, X_2, ...\}$ to denote a stochastic process which generates binary trees in a breadth first search order according to a certain probability distribution. The probability space of $\mathcal{X}$ is defined as $(x_1, x_2, ..., x_n)$ for $n = 1, 2...$ where $(x_1, x_2, ..., x_i) \in D_i$ and D_i is the set of all distinct BFS (breadth first search) trees with i nodes. Recall that we say $\mathcal{X}$ is a *stationary* process if the joint probability distribution of any subset of the sequence is invariant with respect to shifts in the time index, i.e.,

$$\begin{aligned} Pr\{ \quad & (X_1, X_2, ..., X_n) = (x_1, x_2, ..., x_n)\} \\ & = Pr\{(X_{1+k}, X_{2+k}, ..., X_{n+k}) = (x_1, x_2, ..., x_n)\} \end{aligned} \tag{6.1}$$

for every time shift k and for all $(x_1, x_2, ..., x_n) \in \mathcal{X}$. However, in the binary tree case, the probability space is the set of all possible binary trees.

In the case of tree generation, at each time index k, the stationary source $\mathcal{X}$ generates new trees under a leaf according to the same probability distribution (in terms of breadth first search) as it starts from the root. Specifically, by the stationarity of

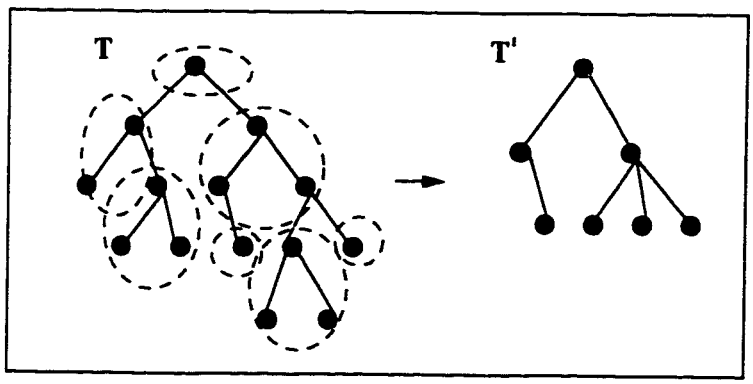

Figure 6.1: Parsing a tree via breadth first search.

the source, when we cut any edge in a binary tree generated by $\mathcal{X}$, the subtree we obtain below this edge has the same probability distribution as the original tree.

We define *entropy* $H(\mathcal{X})$ of a stochastic process $\mathcal{X}$ as the follows:

$$H(\mathcal{X}) = \lim_{n\to\infty} \frac{1}{n} H(X_1, X_2, ..., X_n) \tag{6.2}$$

For a stationary stochastic process $\mathcal{X}$, let ρ_{opt} denote the optimal compression ratio for any sequence it generates, the following holds [34]:

$$H(\mathcal{X}) = \frac{1}{\rho_{opt}} \tag{6.3}$$

6.2 Tree-Compress Algorithm

Before we present the tree compression algorithm, we first define some terminology used in this algorithm. Let T be a binary tree. We call a binary tree T_x is a *prefix tree* (in terms of breadth first search) of T if T_x can be obtained by traversing a certain number of nodes in breadth first search order on binary tree T.

We illustrate our tree-compress algorithm in the case of binary trees. The algorithm is readily extendible to trees with more than two children per node. The

tree compression algorithm utilizes a parsing scheme similar to the parsing of LZW data compression algorithm. The main difference in the data structure is that the dictionary D now contains indices each pointing to a BFS (breadth first search) tree instead of a binary phrase.

Initially D only contains a single index which points to a BFS tree consisting only one node (the root). We proceed by traversing the input tree T by a breadth first search. In addition, we use a queue S to keep track of the traversal. Specifically, S contains a queue of leaves of the current traversed tree. If any leaf x gets expanded to a certain subtree under x, the leaves of the newly created subtree will be added to the end of queue S and x will be deleted from the top of the queue.

Similar to the unique parsing in LZ compression, we first parse T into distinct BFS subtrees and construct a dictionary D which contains indices to distinct BFS subtrees. Each time we visit a new node $\alpha = x_i$, we find the maximum match of the subtree under α to the BFS trees already stored in D. Let subtree $match(\alpha)$ be the maximum match. We have two cases to be considered. If the whole subtree under α is already stored in D, we skip the subtree and continue parsing in breadth-first order. If $match(\alpha)$ is not equal to the whole subtree of α, we add one more index to D which points to a new BFS tree containing the maximum match plus the next node in breadth-first search order x_{i+1}. Let B_x be the newly constructed subtree. We update the counters of visits of every BFS subtree which is a prefix tree of B_x in D. We also update S by adding all the leaves in B_x in breadth first order to S and remove x_i from the top of S. Then we repeat the process in a breadth first search manner, i.e., we find the first node in S and parse the subtree below it and then work on the second node in S and so on until the whole tree is parsed.

After we finish parsing T, the dictionary D contains a set of parsed subtrees. We retain only a subset of BFS subtrees in D to compress the tree which ensures

a distinct parsing of T. This step differs from the original LZW compression that we do not use the whole dictionary but a portion of the dictionary which has a size proportional to the size of the binary tree. This is accomplished by removing all the BFS trees with size bigger than $k = \sqrt{n}$ from D and identify all the BFS trees which is not a prefix tree of any other in D. We calculate the probability of each BFS tree remaining in D using the counter associated with it. Specifically, let x be the sum of all the counters associated with trees (or the total number of parses made) in D and c_v be the number of visits of BFS subtree v in D, the probability p_v of v being visited is calculated as c_v/x. We then assign Huffman codes ([34] and Chapter 2) to each subtree according to the probability of these subtrees. Using this set of subtrees, we parse T again into a group of BFS subtrees (this is possible since the subtrees now ensures a distinct parsing) and construct a tree T' by replacing the subtree with a node containing the Huffman code for the corresponding subtree in the trimmed dictionary D.

It remains to describe how to compress the edges. Given the final trimmed dictionary D', the edges in the original tree T can be classified into two kinds: *internal edges* and *bridging edges*. Internal edges are those edges in T which are contained in one of the parsed subtrees of D'. These internal edges are readily represented by the subtree which it belongs to so the only edges we are concerned are the bridging edges each connect two different parsed subtrees in the compressed tree T'. Let e be such a bridging edge which connects two nodes v_1 and v_2 in T. Also let T_1 and T_2 denote the subtrees that v_1 and v_2 belong to, respectively. We encode e by the binary encodings of v_1 and v_2. Without loss of generality, we assume that v_1 is an ancestor of v_2 in T. Note that v_2 must be the root of the subtree T_2 it belongs to so we only need to give the encoding string of this subtree in order to represent v_2. To encode v_1, we first find the subtree T_1 that v_1 belongs to. Let i be the index of that subtree

in D. We then find the position j of v_1 in the left-to-right order of the leaves of this subtree. We then encode v_1 as $i\#j$ where $\#$ is a separator.

The following is a summary of the tree-compress algorithm.

Tree-Compress (T; D, T')
Input a tree T generated by a stationary ergodic stochastic process;

Step 1(Initialization)
Initialize D to contain a single index representing a BFS tree
containing only one node;
Initialize a queue S to contain roots of subtrees traversed in BFS;
Step 2(Parsing)
Traverse T in breadth first search order
While (search not completed) do
Let α be the current node in T we are visiting
Starting from α; find the maximum match $match(\alpha)$
in D for the subtree below α ($subtree(\alpha)$);
If $match(\alpha) = subtree(\alpha)$,
update the counters for $subtree(\alpha)$ in D;
insert all roots of the subtrees below $subtree(\alpha)$ into S;
let α be the next root of a subtree from S;
Else
form a BFS tree by adding the next node in breadth first search of T to $match(\alpha)$;
add this new BFS tree into D;
update the counters for all the prefix trees of this new BFS tree in D;
insert all roots of the subtrees into S after cutting the new BFS tree from T;
pop a root of a subtree from S and let it be α;
Step 3(Encoding)
Remove all trees in D with size bigger than $k = \sqrt{n}$;
Retain only those trees in D which are not prefix tree of any other tree;
Calculate the probability of occurrences for each tree in D and
assign corresponding Huffman code to it;
Reparse T using D and encode the subtrees by assigning dictionary code;
Identify and encode bridging edges;

Output a dictionary D containing a set of distinct BFS trees and a compressed tree T'.

6.3 Analysis of Tree Compression

In this section we show that the tree compression algorithm compresses a binary tree T efficiently into a compressed form T' with smaller size. Recall that we originally represent each node of T by $\log n$ bits.

Assume that e is an edge in T. If e is an internal edge, e is contained in a parsed subtree of T. The node in T' which represents this subtree already encodes e so no extra space is needed for encoding e in T'.

If e is a bridging edge in T, e is encoded by a concatenation of two binary numbers. Assume that v_1 and v_2 are the two nodes in G that e connects and v_2 is below v_1. Let T_1 be the parsed subtree that v_1 belongs to and T_2 be the parsed subtree that v_2 belongs to. Note that v_1 is a leave of T_1 and v_2 is the root of T_2. The edge e' in T' contains the subtree number in the dictionary for T_2 and the position of the v_1 in the left-to-right order of the leaves of T_1. The node number for T_2 takes at most $1/2 \log n$ bits since the total number of the nodes in T_2 is bounded by $\sqrt{n}$. The total number of leaves in T_1 is also bounded by $k = \sqrt{n}$. Thus the size of e' is bounded by $1/2 \log n + 1/2 \log n = \log n$ which is the size of e. Thus the encoded form of the bridging edge is of the same length as the original form.

Next we prove that our tree compression algorithm achieves optimal compression in the sense that the dictionary used to parse the tree is optimally selected so that the compression ratio $\rho = 1/H(\mathcal{X})$ where $H(\mathcal{X})$ is the entropy of a stationary ergodic source $\mathcal{X}$ from which the binary tree is generated.

Let $\mathcal{X} = (X_1, X_2, ...)$ be a stationary ergodic process that generates the binary trees in depth first search order. Let $P(x_1, x_2, ..., x_n)$ be the underlying probability mass function for $\mathcal{X}$. For a fixed integer k, we define the kth order Markov approximation to P as

$$Q_k(x_1, x_2..., x_n) = \prod_{j=1}^{n} P(x_j | x_{j-k}^{j-1}), \tag{6.4}$$

where $x_i^j = (x_i, x_{i+1}, ..., x_j), i \leq j$. We will show that the algorithm is optimal when $n \to \infty$.

Theorem 6.1 *Assume that the tree T is generated by a stationary and ergodic source, the tree compression algorithm is optimal when $n \to \infty$ where n is the size of the tree T.*

Proof: We approximate the stationary ergodic source $\mathcal{X}$ by k-th order Markov source Q_k where the probability of the next state is only dependent on the previous k states. When $k \to \infty$, $Q_k \to \mathcal{X}$ in the limit. In the construction of the final dictionary D' in our compression algorithm, we estimate the probability distribution of the leaves at k-th level of the binary tree. Let v be an arbitrary leaf in the dictionary and p_v be the probability that v appears in T. Let e_v be our estimation of the probability of v in Q_k. We show that for any given $\epsilon > 0$, $Prob(|p_v - e_v| \leq \epsilon) \to 1$ when the size of the tree $n \to \infty$. Also as $n \to \infty$, $k = \sqrt{n} \to \infty$ and $Q_k \to \mathcal{X}$ in the limit.

Recall that each subtree in T has the same probability distribution from the property of stationarity of the generating source. Let n' be the number of occurrences of v after v appears in the dictionary. Assume that v has a probability p_v in Q_k, the number of occurrences U of v in n' parses forms a binomial distribution with probability p_v.

Applying Chernoff bounds (see introduction) to the binomial distribution, we obtain the following,

$$Prob(U \geq (1+\epsilon)p_v n') \leq \frac{e^{-\epsilon^2 p_v n'}}{2} = \frac{1}{n'^{\Omega(1)}} \tag{6.5}$$

and

$$Prob(U \leq (1-\epsilon)p_v n') \leq \frac{e^{-\epsilon^2 p_v n'}}{3} = \frac{1}{n'^{\Omega(1)}} \tag{6.6}$$

for any given $\epsilon > 0$.

As $e_v = U/n'$, we have $Prob(|e_v - p_v| \leq \epsilon) \to 1$ as $n' \to \infty$.

As the Huffman coding based on the probability distribution of the leaves at k-th level is optimal for encoding the source Q_k, our dictionary achieves optimal compression in the limit when the size of the input tree T grows to infinity. □

The optimality we achieved above is not the same as the term optimality used in normal data compression algorithms. The reason is that the optimal compression defined with respect to the entropy of the input source can not extend to the binary tree case easily. It is not clear how to define the entropy of the source that generates the binary tree. Furthermore, the standard representation of binary tree, which uses $n \log n$ bits, is apparently not efficient, considering the coding scheme we provided earlier, even before any compression is performed. This situation may happen to other complex data structure as well. Therefore, our approach is to use a known optimal compression scheme, for example the Huffman coding in our case, at certain stage of the compression algorithm.

In the above algorithm, the nodes in the original binary tree are not distinct from one another. In other words, there is no specific information contained in each node. In the more complicated case where each node of the original tree contains certain information, we can still compress the tree using the above algorithm while using

another list to store information in each individual node. Note that the position of each node in the original tree can still be represented by $\log n$ bits after compression. However, the only saving in space achieved in this case is accomplished by the compression of the internal edges.

6.4 Compression of Graphs

The tree-compress algorithm can be extended to undirected graph as well. The basic idea is to compress the corresponding breadth first search (BFS) tree of the graph first and then compress the remaining edges accordingly.

Our algorithm for compressing a given undirect connected graph G works in two stages. First we traverse G using breadth first search to obtain a BFS tree T and then apply the tree-compress algorithm to construct a dictionary D and a compressed form T' of T. We then traverse G again to identify all back edges (see definitions in [26]) and encode them using the similar technique for encoding bridging edges in tree compression algorithm. The edges of G can be classified as *internal edges*, *bridging edges* and *back edges*. The first two kinds of edges are contained in the BFS tree of G and are compressed in the tree compression algorithm. The back edges are compressed in the second traversal using the same technique of compressing bridging edges in the tree compression algorithm. However, the size of compressed form of a back edge may exceed the size of the original encoding. More specifically, if e is a back edge connecting v_1 and v_2 in the original graph, let T_1 and T_2 be the subtrees in the dictionary which contains v_1 and v_2 respectively. The encoded form e' of e is stored in the node in G' representing T_1 and e' consists binary numbers representing T_2, v_1 and v_2 (the pre-order for v_1 in T_1 and v_2 in T_2). The number of bits used to specify the back edge e may be as large as $2 \log n$ since we need to specify the positions of v_1 and v_2 respectively. We expect that our algorithm is efficient when

the number of back edges is small comparing to the total number of internal and bridging edges thus the possible loss in compressing back edges is well compensated by the gain in compressing the other edges.

The following is a summary of our compression algorithm for undirected graph.

Graph-Compress (G; D, G')
Input an undirected graph G;

Step 1
Traverse G using breadth first search and construct BFS tree T for G;
Step 2
Apply Tree-Compress Algorithm on T and achieve dictionary D and compressed tree T';
Step 3
Traverse G again using breadth first search;
Compress back edges using similar strategy of compressing
bridging edges in Tree-Compress Algorithm;
Combine T' and compressed back edges to form compressed graph G';
Output a dictionary D containing a set of distinct trees and a compressed graph G'.

For a directed acyclic connected graphs (DAG), our algorithm can be applied similarly except that the depth first search which now follows the direction of the edge may not reach every node in one traverse. In other words, it is possible we obtain a forest of BFS trees instead of a single one. Also there is a new kind of edges, *cross edges* (see [26] for definition) that we need to encode using similar method of encoding back edges in the case of compressing an undirected graph which may need more space to represent than in the original graph. Note the compressed edges in the case of directed graph are also directed edges. Our method of compressing undirected graphs and directed acyclic graphs may not always yield a compressed form with a smaller size since the back edges and cross edges may not be compressed efficiently. However, if the back and cross edges can be identified easily (e.g., when the graph is generated by randomly adding edges to an existing tree) and the number of back edges and cross edges is only a small portion of the total number of edges in the original graph, we expect that our compression scheme to perform well. Furthermore, if the

graph can be parsed into a series of trees (e.g., in the case of parallel-series graph), we can apply our Tree-Compress algorithm to the parsed trees and achieve efficient compression.

In programming languages such as LISP, it requires a large database organized according to some data structure (tree, undirected graph, DAG, etc). To preserve the space for storing such a database, it is desirable to have a compression scheme which compresses the original data structure into one with smaller size. Our algorithms presented in this chapter can be applied to compress the database while keeping a structure similar to the original form. However, in most cases, the information contained in each node of the data structure may vary widely thus our compression scheme has only limited applicability in those kinds of actual databases. In other words, there is limited applicability of our algorithms in this case because there is not much redundancy or repeated patterns to explore. Nonetheless, if the actual data structure does have many repeated patterns (e.g. the identical subtrees in a tree structure) and the data contained in each node varies infrequently, our algorithm may be efficient. It still remains an interesting question how to design compression algorithm for practical data structures where the nodes of the data structure contain independent information and we want to distinguish the nodes which contain different information in the compressed form.

Another possible application of the compression ideas to more complex data structures is to represent individual node in the data structure in a more compact way. For example, in the design of compiler, the individual node containing different information is not only stored in the database from which it is generated but also dynamically referred to in other places of the system. Therefore, a succinct representation of individual node may save storage space further in the case where many copies of the same node exists. One efficient way to achieve this is to use pointers

instead of actual copy of the node. The tradeoff of this method is the time spent in checking whether these pointers are still valid in a dynamic system environment.

6.5 Summary

In this chapter, we extended our study to more complex input data types such as trees and undirected graphs. We introduced a new algorithm for compressing trees using a similar idea of adaptive dictionary construction in LZ compression. We also extended the algorithm to compress undirect graphs.

This chapter concludes our case studies of applying data compression data structures and techniques in other algorithmic areas. These case studies emphasized on different aspects of the use of data compression ideas while they also share the same methodology in design of new algorithm based on known techniques. One key difference among the algorithm problems studied is the assumption on input data. The input in the sorting problem is a set of binary strings. In the string matching problem, the input is assumed to be a continuous stream of characters. Finally, in the tree and graph compression problem, the input is a more complex type than binary string. The different assumptions on the input require the algorithm to apply techniques according to specific nature of the input. We have shown that the data structures and techniques in data compression can aid the design of new algorithms in a variety of ways.

In next chapter we will summarize this dissertation by proposing a theoretical framework of applying data compression techniques to aid algorithm design. We will also give a survey on our ongoing research and describe some possible algorithmic problems as future work.

Chapter 7

Summary and Future Work

This thesis has been focused on the idea of identifying algorithmic problems whose performance is related with certain statistical properties (i.e. bounded compressibility, stationarity, ergodicity, etc) of the input data and applying known techniques and data structures in data compression area to design new algorithms with improved performance. We specifically study the fundamental problem of sorting, string matching and tree compression. We summarizes our algorithms as follows:

- A novel sorting algorithm based on data compression techniques for entropy-bounded inputs. Also a parallel version of the sorting algorithm is developed on PRAM. We also extend the new sorting algorithm to solve problems such as priority queue design and convex hull. (see Chapter 3)

- A fast string matching algorithm for bounded-entropy inputs. We also extend the algorithm to two dimensional textual compression and give parallel algorithms for the problem. (see Chapter 4)

- A novel idea of reducing computational cost by performing operations directly in compressed transform domain and an efficient volume rendering algorithm based on this idea. (see Chapter 5)

- A new algorithm that compresses trees and undirected graphs using a parsing scheme similar to LZ compression. We demonstrated the use of dictionary encoding in compressing complex data type such as binary trees. (see Chapter 6)

Problems	*Assumptions on inputs*	*Techniques used*
Sorting	binary keys from stationary and ergodic source	dictionary construction
String matching	bounded entropy text random pattern	theorem proving complexity analysis
Volume rendering	3-D volume data samples	lossy compression
Graph compression	trees and graphs	unique parsing dictionary construction

Table 7.1: Summary of results.

From the above case studies, we showed the general procedures and noted problems involved in designing new algorithms using data compression techniques and data structure. In next section, we propose a general framework for identifying new algorithms that can be improved using certain data compression techniques.

7.1 Framework: Problems and Solutions

In previous chapters, we concentrated on designing new algorithms for fundamental problems such as sorting and string matching based on statistical properties of input data. In this chapter, we try to outline a general framework for identifying the class of algorithmic problems to which the idea of using statistical properties of input data can be applied and the possible techniques and data structures that can be utilized to improve performance.

The class of algorithmic problems that we have considered in this thesis which can be improved using data compression techniques have the following characterizations:

1. The **input** of the specific problem can be regarded as a continuous flow of data generated by unknown statistical source. For example, input data of the sorting problem can be modeled as a binary stream. The input data is a stream of *on-line* data in the sense that the input data is received by the algorithm solving the problem over a time period. Therefore, the knowledge of the statistical properties of the data

is not available at the beginning of the algorithm. The algorithm has to scan through the input data to learn the underlying properties of the input. As we demonstrated in the new algorithms, the trie data structure used for constructing dictionary in data compression algorithms is an excellent tool to estimate the probability distribution of the input data.

2. The **design** of the new algorithm utilizes data compression data structures and associated techniques such as dictionary building, prefix matching, trie searching or a combination of the above. These techniques proved to be good tools in predicting future data (e.g., incrementally constructed trie) or in string processing (e.g., efficient prefix matching for binary strings).

3. The **performance** of the algorithm designed for solving the problem in this class is determined by certain statistical properties of the inputs. In the sorting and string matching problem, we have shown that the complexity of our algorithms depends on the compressibility of the inputs. There are also other possible assumptions on the input data. For example, the input can be assumed to be generated by certain statistical model, such as n-state Markov source, where the performance of the algorithm is tied to the specific model used.

We summarize some observations about the possible areas in which the data compression techniques can be useful with examples from this thesis as follows:

- **Statistical assumptions**

  Statistical assumptions on the input data are widely used in design and analysis of efficient data compression algorithms. For example, the analysis of the LZ compression is based on the assumption that the input data is generated from a stationary and ergodic source. The LZ compression can be proved to be optimal for input data with this assumption. By making statistical assumptions on practical inputs, the algorithm can adapt the procedures to the assumed statis-

tical properties of the inputs and thus reduce computation time. For example, in the study of the sorting problem we assumed that the entropy of the input keys can be bounded by a constant. Similarly in the string matching problem we assumed that the entropy of the text string is bounded. Furthermore, we assumed that the input source is stationary and ergodic in both studies.

It remains an interesting topic to find other statistical assumptions for the input data which can be proved to be valid for most practical inputs and useful in improving efficiency of new algorithm. For example, in the areas of electrical engineering or physics, the input source can sometimes be modeled with specific probability distribution. The algorithms designed for solving problems concerning this type of input are expected to explore this property to reduce computation.

- **Algorithmic procedure**

  Data compression algorithms use different procedures to remove the redundancy of the input data and transform it into most compact form. Some of the procedures, such as dictionary building, string processing and unique parsing, may be used in solving other problems where a succinct representation of the input is more desirable. The new algorithms for these problems may or may not explicitly compress the input before operating on it. In the study of compression of trees and undirected graphs in Chapter 5, we implicitly used the same idea of dictionary building in LZ compression scheme with modifications to suit the new data type.

  In some cases the knowledge of certain algorithmic procedures in data compression algorithms can lead to efficient algorithm for the compressed form of the input data. For example, the algorithms for efficient string matching on compressed data are designed by implicitly utilizing the properties of the specific

compression scheme used for the compression of the input without decompressing the input data. The performance of this type of algorithms is directly linked to the specific algorithmic procedures used in the compression scheme.

- **Data structure**

  The data structures used in data compression algorithms can also be used in solving algorithmic problem in other areas. The trie structure, for example, is initially designed to construct a dictionary of phrases in LZ compression scheme. Some general statistical properties of the trie make it applicable in many algorithmic problems, especially those dealing with strings (see Chapter 2). In Chapter 3, we demonstrated the use of the trie as a tool to construct an approximation of the probability distribution of input source and to partition the input keys into buckets of even size. In Chapter 5, we extended the phrases contained in the dictionary from simple binary strings to more complex data type such as subtrees and also used the extended dictionary to learn the probability distribution of the input source.

- **Time complexity and optimality analysis**

  The data compression algorithms analyze the time complexity based on the statistical assumptions of the input data. Most data compression schemes assume that the input source is a stationary and ergodic source and the running time of the algorithm depends on the specific data structures and compression techniques used for this type of inputs. Similarly, in our study of string matching problem in Chapter 4, we analyze the expected running time of our algorithm based on the assumption of stationary and ergodic source with bounded entropy. The complexity analysis for restricted input source provides a easier way to calculate the time spent in each step of the algorithm and the overall result is

a closer indication of the practical performance given that the restriction holds widely for real time inputs.

Another important aspect in analysis of a data compression algorithm is whether the algorithm achieves optimality for large enough inputs, i.e. whether the compression ratio becomes optimal when the size of the inputs goes to infinity. For example, the LZ compression scheme achieves this optimal ratio and the proof of this optimality is mainly derived from the properties of the unique parsing technique the algorithm uses. We expect that the similar analysis can be applied to show the optimality of new algorithms employing the same unique parsing scheme.

- **Theorem proving**

  The data compression algorithms include a large number of proof techniques that may find useful in other areas related to compression. The notion of compressibility or entropy itself may provide a new way of proving new theorems, especially those concerning probabilities. We demonstrated the use of compressibility in theorem proving in the study of string matching problem and proved one important lemma by giving a counterexample of compression scheme that contradicts to the assumed optimal compressibility. Also the property of stationarity and ergodicity of input source is widely used in proving theorems in analysis of new algorithms. Statistical properties of certain data structure (e.g. trie) are also of fundamental use in complexity analysis in our sequential sorting algorithm in Chapter 3.

As summarized above, the data structures and techniques used in data compression area are powerful tools in designing efficient algorithms for a large variety of problems. They are especially attractive for certain problems in other areas such as

algorithmic applications in physics and electrical engineering. In these problems, we can assume that the input data satisfies certain known probability distribution (e.g. uniform, Gaussian, binomial, etc). We need to show that the specific assumption on the probability distribution holds for most practical inputs that the new algorithm encounters before we can use the assumption to design new algorithm.

Furthermore, because the new algorithm tailors solution based on statistical property of the input data, it reduces computation time if the input data satisfies certain statistical conditions. Some of the known statistical techniques can be applied in analysis of algorithms, such as the analysis of general statistical properties of suffix tree.

In this thesis, we tried to outline a general framework of design and analysis of algorithms in areas other than data compression by utilizing compression techniques. We hope the methodology combined with the concrete studies will provide a novel starting point to solve algorithmic problems in related areas. Many such applications provide insights in the statistical property of practical data and may as well provide new ideas in the design of data compression algorithms itself.

By no means we claim to have covered all possible ways of utilizing known techniques in data compression. Some interesting and promising topics, such as extending our idea to multi-dimensional data and applying lossy data compression techniques, have not been mentioned. However, the basic methodology studied in details in this thesis provides a guideline in studying these topics as our ongoing or future work. We will show some of these possible extensions in next section.

7.2 Future Work

In this section, we give a brief discussion on some of our ongoing research problems and future work. We try to extend our basic idea to apply data compression

techniques to input sources that are not necessarily stationary. Also we exploit the possibility of applying lossy data compression techniques to achieve approximation algorithms. Furthermore, the two or higher dimensional compression techniques may find useful in areas such as image processing.

7.2.1 Extension to Non-stationary sources

This thesis has studied algorithmic problem where the input data to the problem can be modeled as being generated from a stationary and ergodic source. The underlying probability mass function of the source does not change with the time index. This underlying probability mass function can therefore be learned by adaptively adjusting the current compression model. It is well known that the Lempel-Ziv adaptive dictionary compression algorithm is able to achieve optimality as the length of the input goes to infinity, in the hope that the compression model will converge to the one which captures full statistical properties of the input source.

However, many practical input sources can not be regarded as stationary. Although adaptive compression algorithms which compress stationary well can be used to compress non-stationary sources by periodically starting over again and again, the loss of compression efficiency is intolerable.

In this section, we give some considerations on the problem of building efficient data compression models for compressing nonstationary sources. Specifically, we introduce general models for evaluating the efficiency of various data compression algorithms for nonstationary sources. The first model assumes that the input is generated by a k-state dynamic Markov model (DMM) where k is a preset constant. We design a simple data compression algorithm based on this model and analyze its efficiency. The second model, which we call a *LZ-tree* model, assumes that the input is generated from a dynamic probability tree which is similar to the LZW tree used

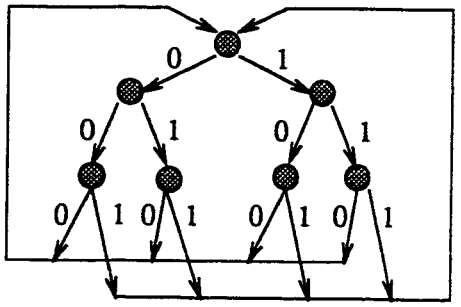

Figure 7.1: Initial model of a binary tree.

in analysis of LZW adaptive compression algorithm for stationary sources.

Dynamic Markov Model

We describe our Dynamic Markov model (DMM) in detail in this subsection. Note that there is a similar technique used in adaptive compression for a stationary source - Dynamic Markov Coding (DMC). DMC starts with a small initial model and grows it by adding new states when needed. It builds a complex state model that fits a particular stationary source by adaptively growing the model. The basic idea of DMC is to maintain frequency counts for each transition in the current finite-state model, and to "clone" a state when a related transition becomes sufficiently popular (i.e. the frequency count exceeds a preset threshold).

Our DMM is a variation of the DMC model. We assume the input binary string is generated by DMM and the initial model of DMM is known to the compressor. Instead of adding more states to the initial model, DMM only updates the frequency counters and select best codings according to the current model. There are various choices for the initial model (chain model, binary tree model, braid model and etc [7]). We choose the binary tree model for illustration of DMM since other models rarely outperforms simple binary tree model.

The DMM starts with the initial model which is a k-node binary tree with equal probability distributions where k is a known constant. The model then generates

phrases each of length k according the probability associated with edges in the model. Upon traveling one edge in the model, the probability of the edge is perturbed in a random fashion.

The compression algorithm starts by assuming the same initial model as the DMM. However, the changing of the probabilities between states are implicit to the compression algorithm. Hence the compression algorithm needs to maintain its own copy of the DMM model by learning the changes from the inputs. However, the compression would not be optimal until the compressor has learned the change from the inputs. The amount of inefficient compression is determined by the frequency and amount of probability changes in DMM.

LZ-Tree Model for Nonstationary Source

We propose a model for describing nonstationary source. The model is closely related to the finite-window model used in analyzing data compression algorithms for stationary sources. We assume that the non-stationary sources we study in this paper is generated by this model. We maintain a probability trie (PT) similar to the one used in LZW compression. After generating a phrase according to the probability of each node in the tree, the model randomly delete existing nodes or insert new nodes with a certain probability after a fixed window of input.

To illustrate how the PT model works, we first review the main data structure used in LZW compression. Each node x in the trie represents a phrase which corresponds to the path from the root of the trie to x. The trie is designed to answer the following query: given a sequence X, what is the longest prefix of X that is already in the trie. For each node x in the trie, we associate with it a counter $c(x)$ which keeps the frequency of x being visited. We parse the input by the same method used in LZW compression, i.e., the next phrase is the longest prefix of the input that already

exists in the trie. Let node x be such a node that corresponds to this longest prefix. We then update $c(x)$ accordingly. We update the trie along with the counters after reading every n characters from the input.

The updating of the trie is accomplished periodically after n characters of the input have been read. We search through the whole trie using standard traversal method (e.g. depth first search). We set the average visiting frequency for each node as the number of leaves in the trie divided by the number of phrases parsed in this period. At each node x, we check if the visiting frequency is below average. If so, we delete the node with a certain probability c $(0 < c < 1)$. We will show that after each interval n, the probability tree we obtain after updating is a close representation of the current probability mass function of the source within a certain error bound.

The probability tree we use has an important property that optimal compression is achieved if the probability tree the compression is based on exactly represents the underlying probability mass function of the source. In the case of LZW compression of the stationary source, the LZ tree (one version of the general probability tree) is updated adaptively through the parsing of the input text. Therefore the LZ tree limits to the exact representation when the length of the input text goes to infinity and the LZW compression based on the LZ tree is optimal in the limit.

The LZ-tree model can be used to monitor the change of probability distribution in the input source. The updating of the tree captures the probability change in each node of the trie by unique parsing of the input data stream. However, this requires the assumption that the probability distribution of the input source does not change too quickly over time. In other words, we need the assumption that the difference between current probability distribution of the source and what is reflected by current LZ-tree can always be bounded to ensure that the LZ-tree is a good approximation of current source. For input source that changes its probability distribution frequently,

other models need to be developed. However, in the case that the source appears to be chaotic, it is difficult to design algorithms that take into account of all possible probability distributions. Fortunately, in practice most input data sources appears to demonstrate a great deal of stationarity for certain statistical model to be applicable.

7.2.2 Lossy compression and multi-dimensional data

It will be a interesting challenge to apply lossy compression techniques to design approximation algorithms. In Chapter 5, we proposed a novel idea of performing operations directly in compressed transform domain. The compression method used in compressing the vectors or arrays is similar to standard lossy compression method in image compression, such as JPEG. It is a promising task to extend this idea further to use lossy compression techniques to reduce computational cost because it is well known that input data, such as those in the area of image and speech processing, can be lossily compressed well without losing vital information.

Unlike lossless compression algorithms, the lossy compression algorithms try to gain more speed by giving up accuracy in compressed data. In other words, some information is lost after lossy compression. Lossy compression has been found especially useful in compressing two or higher dimensional data, such as two dimensional images, where the loss in the compressed form is not essential [77, 78, 111, 17, 53]. A well known technique in image compression is to divide the image into small blocks and match each block against a pre-constructed set of image samples of the same size. Each image block is then represented in the compressed image by the index of the closestly matched image sample in the sample set. The extreme case where the image block is only of size of a single pixel gives no compression while there is no loss of information. The bigger the size of the image block, the more information is lost after compression. Therefore the quality of the compressed image depends on the

resolution of the partitioning of the image. This technique can be improved by adaptively constructing the image samples based on the actual image being compressed and thus the compression scheme achieves optimal encoding for the specific image. Also there is a recursive encoding method where the blocks are subdivided into sub-blocks and the sub-blocks are recursively encoded. These methods have been proven to be very efficient in image compression and some of the best image compressors such as JPEG are based on similar ideas.

These methods for lossy compression may find use in solving other problems in image processing. For example. it will be an interesting task to solve the problem of efficient pattern matching in a compressed image without decompressing the image. In string matching algorithm in Chapter 4, there is a similar problem in one dimensional case where the basic methodology remains the same. To solve this problem, it is essential to study the compression scheme and utilize the properties of this specific compression to speed up pattern searching. Based on the compression used, the matching algorithm can take advantage of the specific encoding method used in searching for possible pattern matches. It is noteworthy that for compression algorithms that are non-adaptive (i.e. with fixed encoding scheme), the corresponding pattern matching algorithms are relatively easy to design, with the basic idea of compressing the pattern using the same compression scheme and then performing a normal pattern matching algorithm on the compressed image using the compressed pattern. However, in case that the compression algorithms are adaptive, it requires novel procedures to search for pattern match in the compressed image.

Bibliography

[1] A. Amir and G. Benson. Efficient two dimensional compressed matching. *Proc. of Data Compression Conference, Snow Bird, Utah*, pages 279-288, Mar 1992.

[2] A. Amir and G.Benson and M.Farach. Witness-free dueling: The perfect crime! (or how to match in a compressed two-dimensional array). submitted for publication, 1993.

[3] A. Amir and G. Benson and M. Farach. Let sleeping files lie: pattern matching in Z-compressed files. *Symposium on Discrete Algorithms*, 1994.

[4] A. Andersson and S. Nilson. A new efficient radix sort. *35th Symposium on Foundations of Computer Science*, 714–721, 1994.

[5] S. Azhar and G. Badros and A. Glodjo and M.Y. Kao and J.H. Reif. Data Compression Techniques for Stock Market Prediction. *Data Compression Conference*, 1994.

[6] A.V. Aho and M.J. Corasick. Efficient string matching: and aid to bibliographic search. *Communications of the ACM*, Vol 18, 6:333-340, 1975.

[7] T.C. Bell, J.G. Cleary and I.H. Witten, *Text Compression,* Prentice Hall Publisher, 1990.

[8] P. Beam and J. Hastad, Optimal bounds for decision problems on the CRCW PRAM, *J.ACM,* 36:643-670.

[9] P. Billingsley, *Ergodic Theory and Information.* John Wiley & Sons, New York 1965.

[10] P. Van Emade Boas, R. Kaas and E. Zulstra, Design and implementation of an efficient priority queue, *Math. Systems. Theory,* 10:99-127.

[11] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith and M. Zagha, A comparison of sorting algorithms for the Connection Machine CM-2, *3rd Annual ACM Symposium on Parallel Algorithms and Architecture,* July 21-24, 1991.

[12] A. Bookstein and S.T. Klein and T. Raita and I.K. Ravichandra Rao and M.D. Patil, Can Random Fluctuation Be Exploited In Data Compression, Data Compression Conference, 1993.

[13] R.S. Boyer and J.S. Moore, A fast string searching algorithm, *Communications of the ACM,* 20(10):762–772, 1977.

[14] G. Bilardi and A. Nicolau, Bitonic sorting with $O(N \log N)$ comparsions, *Proc.20th Ann. Conf. on Information Science and Systems,* Princeton, NJ(1986)309-319.

[15] D. Bino and V. Pan. *Polynomial and matrix computations*, Vol 1, Birkhauser.

[16] R. Board and L. Pitt, On the Necessity of Occam Algorithms. *Proceedings of the 22nd Annual ACM Symposium on Theory of Computation*, May 1990, pp 54-63.

[17] A. Borodin and P. Tiwari. On the Decidability of Sparse Univariate Polynomial Interpolation, *Proc. 22nd Ann. ACM Symp. on Theory of Computing*, 535–545, ACM Press, New York, 1990.

[18] R.L. Baker and Y.T. Tse. Compression of high spectral resolution imagery. In *Proceedings of SPIE: Applications of Digital Image Processing*, pages 255–264, August 1988.

[19] P. Cignoni and L.D. Floriani and C. Montani and E. Puppo and R. Scopigno, Multiresolution modeling and visualization of volume data based on simplicial complexes, *Proceedings of Symposium on Volume Visualization*, 1994.

[20] R.J. Clarke, *Transform coding of images*, Academic Press, 1985.

[21] K.L. Clarkson, Applications of random sampling in computational geometry ii. *Proc of the 4th Annual ACM Symp on Computational Geometry*, pages 1–11, 1988.

[22] A. Czumaj and Z. Galil and L. Gasieniec and K. Park and W. Plandowski, Work-Time-Optimal Parallel Algorithms for String Problems, *Syposium on Theoretical Computing, 1995.*

[23] R. Cole and R. Hariharan. Tighter bounds on the exact complexity of string matching. *33th Symposium on Foundations of Computer Science*, 600–609, 1992.

[24] K. Curewitz,P. Krishnan and J.S. Vitter. Practical Prefetching via Data Compression, *Proceedings of the 1993 SIGMOD International Conference on Management of Data (SIGMOD '93)*, Washington, D.C, May 1993, 257–266.

[25] M. Crochemore and L. Gasieniec and W. Rytter. Two-dimensional pattern matching by sampling. *Information Processing Letters*, 46:159–162, 1993.

[26] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to algorithms*, McGraw-Hill Book Company, 1990.

[27] R. Cole, Parallel merge sort, *SIAM J.Comput*, 17(1988)770-785.

[28] S. Chen and J. H. Reif. Using difficulty of prediction to decrease computation: Fast sort, priority queue and computational geometry for bounded-entropy inputs. *34th Symposium on Foundations of Computer Science*, 104–112, 1993.

[29] S. Chen and J. H. Reif. Fast pattern matching for entropy-bounded inputs. *Data Compression Conference*, 282–291, Snowbird, April, 1995.

[30] S. Chen and J.H. Reif. Survey on predictive computing. Duke Univ. Technical Report No. CS–1995–14, October, 1995.

[31] S. Chen and J.H. Reif. Efficient lossless compression for trees and graphs. *Data Compression Conference*, Snowbird, 1996.

[32] S. Chen and J.H. Reif. Adaptive compression models for non-stationary sources. Manuscript, 1996.

[33] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annuals of Mathematical Statistics*, 23:493–507, 1952.

[34] T.M. Cover and J.A. Thomas, *Elements of information theory*, John Wiley & Sons, 1990.

[35] J.G. Cleary and I.H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Tran. on Communications*, 32:396–402, 1984.

[36] V. Cuperman. *Efficient waveform coding of speech using vector quantization.* PhD thesis, University of California, Santa Barbara, Feb 1983.

[37] J. Danskin and P. Hanrahan. Fast algorithms for volume ray tracing. *Proceedings of the 1992 Workshop on Volume Visualization*, pages 91–106, Boston, 1992.

[38] D. Dudgeon and R.Mersereau. *Multidimensional Signal Processing.* Prentice-Hall, N.J., 1984, pp.81, 82, 363–383

[39] C. Derman, *Finite state Markov decision processes*, Academic Press, New York, 1970.

[40] P. Van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, *Information Processing Letters,* volume 3, No.3, 1977.

[41] M. Farach and M.'Noordewier and S. Savari and L. Shepp. On the Entropy of DNA: Algorithms and Measurements based on Memory and Rapid Convergence. *SODA*, 1995.

[42] T.R. Fischer and D.J. Tinnen. Quantized control with differential pulse code modulation. In *Conference Proccedings: 21st Conference on Decision and Control*, pages 1222-1227, December 1982.

[43] T.R. Fischer and D.J. Tinnen. Quantized control using differential encoding. *Optimal Control Applications and Methods*, pages 69-83, 1984.

[44] T.R. Fischer and D.J. Tinnen. Quantized control with data compression constraints. *Optimal Control Applications and Methods*, pages 593-596, May 1982.

[45] Z. Galil. A constant-time optimal parallel string-matching algorithm. *24th Symposium on Theory of Computation*, 69-76. 1992.

[46] R.G. Gallager. Variations on a theme by Huffman. *IEEE Trans. on Information Theory*, 24:668–674, 1978.

[47] J. Gil and Y. Matias, Fast hashing on a PRAM - Designing by expectation, *Proc. 2nd. Ann. ACM Symp. on Discrete Algorithms.*

[48] Z. Galil and Kunsoo Park. An improved algorithm for approximate string matching. *SIAM J. Comput.*, Vol 19, 6:989-999, December 1990.

[49] Z. Galil and J. Seiferas, Time-space-optimal string matching. *Journal of computer and System Sciences*, 26(3):280-294, 1983.

[50] R.M. Gray. Vector Quantization. *IEEE ASSP Magazine*, 1:4-29, April 1984.

[51] R.M. Gray. *Entropy and information theory.* Sringer-Verlag, 1990.

[52] R.M. Gray. *Source coding theory.* Kluwer Academic Publishers, 1990.

[53] S. Gupta and A. Gersho. Feature predictive vector quantization of multispectral images, *Trans. on Geoscience & remote sensing*, 30(3):491-501, May 1992.

[54] T. Hagerup, Towards optimal parallel bucket sorting, *Inform. and Comput.*, 75(1987)39-51.

[55] T. Hagerup and C.RÜB, A guided tour of Chernoff bounds, *Information Processing Letter* 33(1989/90)305-308, North-Hollend.

[56] M.C. Harrison. Implementation of the substring test by hashing. *Communications of the ACM*, 14:777-779, 1971.

[57] G. Held. *Data compression*, John Wiley & Sons, 1991.

[58] K.H. Holm. Graph matching in operational semantics and typing. In *Proceedings of Colloquium on Trees in Algebra and Programming*, pages 191–205, 1990.

[59] J.S. Huang and Y.C. Chow, Parallel sorting and data partitioning by sampling, *Proceedings of the IEEE Computer Society's Seventh International Computer Software and Applications Conference*, 627-631, November 1983.

[60] W.L. Hightower, J.F. Prins and J.H. Reif, Implementations of randomized sorting on large parallel machines, *Symposium on Parallel Algorithm and Architecture, 1992.*

[61] D.A. Huffman, A method for the construction of minimum-redundancy codes, *Proc. Institute of Electrical and Radio Engineering, 40* (9),1098-1101, September.

[62] J.JáJá, *An introduction to parallel algorithm*, Addison-Wesley, 1992.

[63] R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. Technical Report TR-31-81, Aiken Computation Laboratory, Harvard University, 1981.

[64] D.E. Knuth, *The art of computer programming*, Volume 3, Sorting and searching, Addison-Wesley, Reading, 1973.

[65] E. Karatsuba and Y.N. Lakshman and J.M. Wiley, Modular Rational Sparse Multivariate Polynomial Interpolation, *Proc. ACM-SIGSAM Int. Symp. on Symb. and Alg. Comp. (ISSAC '90)*, 135–139, ACM Press, New York, 1990.

[66] D.E. Knuth and J.H. Morris and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comput*, 8:323-350, 1977.

[67] A.R. Karlin, S.J. Philips and P. Raghavan, Markov paging, *Thirty-third Annual Symposium on Foundations of Computer Science*, Pittsburgh, Pennsylvania,1992.

[68] P. Krishnan and J.S. Vitter, Optimal Prefetching in the worst case, *Proceedings of the 5th Annual SIAM/ACM Symposium on Discrete Algorithms*, Alexandria, VA, January 1994.

[69] M. Levoy. Display of surface from volume data, *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[70] J.S. Lim. *Two Dimensional Image and Signal Processing* Prentice-Hall, 1990.

[71] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *Computer Graphics*, 28(4):451–458, 1994.

[72] T. Linder, G. Lugosi, and K. Zeger. Universality and rates of convergence in lossy source coding. *preliminary draft*, 1992.

[73] T. Malzbender. Fourier volume rendering. *ACM Transactions on Graphics*, 12(3):233–250, July 1993.

[74] M.R. Nelson, LZW data compression, *Dr.Dobb's Journal*, October 1989.

[75] H. Mannila, Measures of Presortedness and Optimal Sorting Algorithms, *IEEE Trans. Computers*, 318-325, 1985

[76] S. Muthukrishnan and K. Palem. Highly efficient dictionary matching in parallel. *extended abstract*, 7:51–75, 1992.

[77] T. Markas and J. Reif. Multispectral image compression algorithms, In *Proceedings of 3rd Annual Data Compression Conference*, pages 391–400, April 1993.

[78] T. Markas and J. Reif. Image compression methods with distortion control capabilities. In *Proceedings of 1st Annual Data Compression Conference*, pages 121–130, April 1991.

[79] P.D. MacKenzie and Q.F. Stout, Ultra-Fast Expected Time Parallel Algorithms, *SODA Conference*, 1991.

[80] Y. Matias and U. Vishkin, On parallel hashing and integer sorting, *Proc. of 17th ICALP, Springer LNCS 443*, 729-743, 1990.

[81] Y. Matias and J.S. Vitter and W.C. Ni. Dynamic Generation of Discrete Random Variates, *Proceedings of the 4th Annual SIAM/ACM Symposium on Discrete Algorithms*, January 1993.

[82] V.S. Miller and M.N. Wegman, Variations on a theme by Ziv and Lempel, *Combinatorial algorithms on words,* edited by A.Apostolico and Z.Galil, 131-140, NATO ASI Series, Vol. F12., Springer-Verlag, Berlin.

[83] M. Palmer and S. Zdonik, "Fido: A Cache that Learns to Fetch," *Proceedings of the 1991 International Conference on Very Large Databases*, September 1991.

[84] B. Pittel, Asymptotical growth of a class of random trees, *The Annals of Probability,* 1985, Vol 13, No.2. 414-427.

[85] W.B. Pennebaker and J.L. Mitchell. *JPEG still image data compression standard.* Van Nostrand Reinhold, 1992.

[86] P. McIlroy, Optimistic Sorting and Information Theoretic Complexity, *SPAA 93,* 467-474, 1993.

[87] P. Raghavan, A statistical adversary for on-line algorithms, DIMACS Series on Discrete Mathematics and Theoretical Computer Science, Vol 7, 1992.

[88] M.O. Rabin, Probabilistic algorithms, *in J.F. Traub, ed. Algorithms and Complexity,* pages 21–36, 1976.

[89] M. Rodeh, V.R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *J. of ACM*, 28:1:16–24, 1981.

[90] S. Rajasekaran and J.H. Reif, Optimal and sublogarithmic time randomized parallel sorting algorithms, *SIAM J.Comput.,* 18:594-607, 1989.

[91] S. Rajasekaran and S. Sen, On parallel integer sorting, *Acta Informatica,* 29, 1-15(1992).

[92] T. Raita and J. Teuhola. Predictive text compression by hashing. New Orleans, LA.

[93] J. Reif, An optimal parallel algorithm for integer sorting, *Proc.26th Ann. IEEE Symp. on Foundations of Computer Science* (1985)496-504.

[94] J.H. Reif and S. Sen, Optimal randomized parallel algorithms for computational geometry. *Proc. of the 16th International conference on Parallel Processing,* 1987.

[95] J.H. Reif and L.G. Valiant, A logarithmic time sort for linear size networks, *Proc. 21st Ann. ACM Symp.on Theory of Computing* (1989)264-273.

[96] R. Reischuk, A fast probabilistic sorting algorithm, *SIAM J.Comput.*, 14(1985)396-409.

[97] M. Rodeh and V.R. Pratt and S. Even, Linear algorithm for data compression via string matching, *Journal of the Association for Computing Machinery* Vol.28, No.1, January 1981, pp. 16–24.

[98] F. Rubin. Experiments in text file compression. *Communication of the ACM*, 19:11:617–623, 1976.

[99] P. Sabella. A rendering algorithm for 3d scalar fields. *Computer Graphics*, 22(4):51–58, 1988.

[100] S. Sen, *Random sampling techniques for efficient parallel algorithms in computational geometry*, Ph.D thesis, Duke University, 1989.

[101] R. Solovay and V.Strassen. A fast monte-carlo test for primality, *SIAM Journal of Computing*, 84–85, 1977.

[102] J.A. Storer, *Data compression: methods and theory*, Computer Science Press, Rockvill, Maryland, 1988.

[103] D.D. Sleator and R.E. Tarjan. Amortized Efficiency of List Update and Paging Rules, *Communications of the ACM*, Vol 28, No.2, pp. 202-208, February 1985.

[104] W. Szpankowski, A typical behavior of some data compression schemes, *Data Compression Conference*, 1991.

[105] W. Szpankowski, (Un)expected behavior of typical suffix trees, *Data Compression Conference*, 1991.

[106] Y. Shiloach and U. Vishkin, Finding the maximum, merging, and sorting in a parallel computation model, *Journal of Algorithms* 2:88-102(1981).

[107] C. Upson and M. Keeler. V-buffers: Visible volume rendering. *Computer Graphics*, 22(4):59–64, 1988.

[108] J.S. Vitter and P. Krishnan, Optimal prefetching via data compression, In *Journal of ACM*, 43(5), September, 1996. A shortened version appeears in *Thirty-second Annual IEEE Symposium on Foundations of Computer Science*, 1991.

[109] L.A. Westover. Splatting: a parallel, feed-forward volume rendering algorithm, Ph.D thesis, University of North Carolina, Department of Computer Science, 1991.

[110] L. Westover, Footprint evaluation for volume rendering, *Computer Graphics*,, Volume 24, Number 4, August 1990.

[111] I.H. Witten and T.C. Bell and H. Emberson and S. Inglis and A. Moffat, Textual image compression: two-stage lossy/lossless encoding of textual images, *Proceedings of the IEEE*, No.6, 1994.

[112] J. Wilhelms and A.V. Gelder. Muti-dimensional trees for controlled volume rendering and compression, *IEEE Proceedings of Symposium on volume visualization*, 1994.

[113] J. Ziv and A. Lempel, A universal algorithm for sequential data compression, *IEEE Trans. Information Theory*, 23, 3, 337-343(1977).

[114] J. Ziv. Coding theorems for individual sequences, *IEEE Trans. Information Theory*, 24, 405-412(1978).

[115] A.C.C. Yao. The complexity of pattern matching for a random string. *SIAM J.Comput*, 8:3:368–387, 1979.

Biography

Shenfeng Chen was born in Shanghai, China. He received his B.S. degree in Computer Science and Engineering from Shanghai JiaoTong University in July 1991. His research interests include data compression, predictive computing, parallel and randomized algorithms. His major publications include: "Using difficulty of prediction to decrease computation: Fast sort, priority queue and computational geometry for bounded-entropy input" (appeared in 34th Symposium on Foundations of Computer Science, 104–112, 1993), "Fast pattern matching for entropy-bounded inputs", (appeared in Data Compression Conference, 282–291, Snowbird, April, 1995), and "Efficient lossless compression for trees and graphs" (appeared in Data Compression Conference, Snowbird, 1996).