Efficient Algorithms for New Computational Models

by

Jan Matthias Ruhl

Dipl.-Math., Albert-Ludwigs-Universität Freiburg, Germany (1997)
Dipl.-Inf., Albert-Ludwigs-Universität Freiburg, Germany (1998)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2003

Author . . . .
Department of Electrical Engineering and Computer Science
August 29, 2003

Certified by . . . .
David R. Karger
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Efficient Algorithms for New Computational Models

by

Jan Matthias Ruhl

Submitted to the Department of Electrical Engineering and Computer Science
on August 29, 2003, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Advances in hardware design and manufacturing often lead to new ways in which problems can be solved computationally. In this thesis we explore fundamental problems in three computational models that are based on such recent advances.

The first model is based on new chip architectures, where multiple independent processing units are placed on one chip, allowing for an unprecedented parallelism in hardware. We provide new scheduling algorithms for this computational model.

The second model is motivated by peer-to-peer networks, where countless (often inexpensive) computing devices cooperate in distributed applications without any central control. We state and analyze new algorithms for load balancing and for locality-aware distributed data storage in peer-to-peer networks.

The last model is based on extensions of the streaming model. It is an attempt to capture the class of problems that can be efficiently solved on massive data sets. We give a number of algorithms for this model, and compare it to other models that have been proposed for massive data set computations.

Our algorithms and complexity results for these computational models follow the central thesis that it is an important part of theoretical computer science to model real-world computational structures, and that such effort is richly rewarded by a plethora of interesting and challenging problems.

Thesis Supervisor: David R. Karger
Title: Associate Professor

Acknowledgments

Many people helped me directly or indirectly during the writing of this thesis and the research leading up to it, or generally by making my five years at MIT so enjoyable. To all of them, I owe many thanks.

First of all, I want to thank my advisor David Karger. David's enthusiasm for research was a constant source of motivation for me; for every solved problem, he would think of two new problems – just solving all the further research problems he suggested based on this thesis would take me several years. His emphasis on intuition over technical details when presenting a result has certainly left its influence on me; I wish I had more time to let this thesis live up to his standard of writing. Last, but not least, I am very grateful for all his advice on "life after grad school".

One of the main reasons why research is so much fun, especially at a place like MIT, is the ability to work on research problems together with other people. I owe thanks to the many friends and colleagues with whom I have worked on research problems during my grad school time, sometimes producing publishable results, sometimes not, but always creating an enjoyable research experience. I have learned more from working with these people than I ever could have learned from books or classes.

So I want to thank Yevgeniy Dodis, Dan Engels, Jon Feldman, Venkat Guruswami, David Karger, Alantha Newman, and Zully Ramzan, who worked with me during my years at MIT. I also want to thank Gagan Aggarwal, Marshall Bern, Krishna Bharat, Bay-Wei Chang, Mayur Datar, Ron Fagin, David Goldberg, Monika Henzinger, Sridhar Rajagopalan, and An Zhu, with whom I spent many hours working (and, in some cases, goofing off) during my summer internships.

Much of this thesis resulted from joint work with these people, and our results have been previously published in [EFKR01, KR02, ADRR03, RADR03]. So I particularly want to thank my co-authors on these papers:

Jon Feldman was a constant source of enthusiasm, and a master of the "looking at a problem a different way" methodology. Our work together on the scheduling problem in Chapter 2 of this thesis, the Steiner network problem [FR99], Lemma 6.20 of this thesis which had long eluded me, and many other problems, were a high-point of my graduate studies.

Dan Engels' daily queries of "You still haven't solved it yet?!?" were very motivating during our work on the scheduling problem. David Karger collaborated with me on the scheduling problem and the peer-to-peer results in Chapters 3-5.

The work in streaming on Chapter 6 was begun in the summer of 2001 based on discussions with Sridhar Rajagopalan at IBM Almaden, and continued the next year with fellow summer interns Gagan Aggarwal and Mayur Datar of Stanford. All of them deserve many thanks.

Writing and defending this thesis occupied me during the last four months. Many people made this job much easier than it would have been for me alone. I want to thank my thesis committee: David Karger, Piotr Indyk and Robert Morris, whose comments improved the presentation and the content of this thesis considerably.

Alantha Newman was a tremendous help in all aspects of writing this thesis. Her unfaltering moral support was an invaluable resource for me during these months. She also gave me lots of helpful advice on my thesis defense.

Maria Minkoff, who was also writing her thesis at the same time, also was an invaluable resource for me during this time.

Frank Dabek provided me with much data on the structure of the Internet.

Many hours of writing this thesis were spent in the Someday Cafe in Somerville, MA, whose mochas are the best I have had on this side of the Atlantic, and in The Prolific Oven in Palo Alto, CA, whose mochas are also quite good. I want to thank them for keeping up a continuing (and, in the case of the Someday Cafe, often free) supply of caffeine.

I also want to thank Anna Lysyanskaya, Kyle Rose, and everybody else in the theory group at MIT. In particular I thank Be Blackburn for always giving cheerful advice and cookies.

Finally, I want to thank my family – my sisters Carolin, Kathrin, Kirsten, my Mum and Dad – for their support during the last five years. It's been a great time!

Matthias Ruhl
Palo Alto, CA
August 29, 2003

Contents

Chapter 1

Introduction

1.1 Computational Models

Since computers were invented 60 years ago, they have undergone many changes. In particular, there have been tremendous improvements in their size and speed that continue to happen even today. But during all this time, the underlying architecture of computers has stayed essentially the same. An off-the-shelf PC today, just like a computer 50 years ago, consists of a single CPU connected via a bus to main memory and external storage media.

The capabilities of this computing architecture are captured by the *RAM model* (short for Random Access Machine model) in theoretical computer science. It assumes that the machine has a (potentially) unlimited main memory, and the execution of all basic instructions (such as reading from and writing to memory, arithmetic and logical operations, and so on) take the same amount of time.

Having a mathematical model of a computing system allows one to reason formally about its capabilities. We can show whether a problem can or cannot be solved in the model, and in the former case, decide whether it can be solved *efficiently* in the model. For example, in the RAM model, a problem is considered efficiently solvable if it can be solved by an algorithm that runs in time polynomial in the input size. In general, the hardness of a problem depends on the computational model used. This fact is often forgotten simply because the RAM model is such a common, even implicit, assumption when reasoning about computation.

A Brief History of Computational Models

While the RAM model is clearly the most important computational model, it is definitely not the only such model studied in theoretical computer science. We will give a brief and by no means exhaustive list of such models.

Even before the invention of the first computers, mathematicians studied the notion of "what can be systematically computed" in the 1920s and 1930s. This led to the definition of such models as the λ-calculus [Chu41] and Turing-machines [Tur36]. While these models are equivalent in terms of what they can compute [vEB90], neither is practical and they do not correspond to any actual machines.

As already mentioned, the first programmable computers invented in the 1940s followed the RAM model, as has been true for most computers since then.

Multi-processor computers were introduced in the 1950s and 1960s for computationally intensive application areas. With multiple processors, the notion of efficiency changes from that in the RAM model. For a multi-processor system, an algorithm is efficient if it successfully exploits the parallelism offered by having several CPUs. Important complexity classes studied in this setting include PRAM classes, **NC** and **RNC** (see e.g. [KR90]).

In the 1970s and 1980s, the introduction of networking technology led to a continuing interest in distributed computations. Here, computational tasks are split onto several computers joined in a network. The communication between nodes is costly in terms of computation time, so the notion of efficiency includes reducing the communication and synchronization necessary for the computation.

In the 1990s, computers finally became part of everyday life. Most homes own a computer, and chips can be found in an increasing number of household appliances. This commoditization has led to a spurt in the development of new computational systems. To study these systems formally, we need to define new computational models that capture the capabilities of these system accurately.

New Computational Models

In this thesis, we consider three different areas of computer systems that have (re)attained practical importance in the past 10 years, and therefore have been the subject of intense study in the computer systems community. These areas are "processors with multiple instruction units", "peer-to-peer networks" and "massive data set computations". In all three areas, there are still many open problems, and the research done on them in theoretical computer science is by no means exhaustive.

We will develop a number of algorithms that are efficient for these computational models, and solve basic and important problems in them. Some of our results have applications even outside of our study of computational models (see Section 1.5).

In the remainder of this introduction, we will discuss these three models in more detail, and summarize our results in a non-technical manner. For more details, and more discussion of related work, we refer the reader to the chapters that follow.

1.2 Processors with Multiple Instruction Units

Traditionally, multi-processor systems have been used mostly in high-end computers. From a software perspective, one of their main characteristics is the fact that transferring data between different processors is much more time-consuming than executing single instructions. Therefore, devising efficient algorithms requires reducing the need to transfer data.

The reduction in size, and improvement of chip design has blurred the distinction between single and multi-processor systems. Now, modern chips have multiple instruction-processing units on a single chip that can perform operations in parallel. Mass-market examples of such an architecture are so-called VLIW processors, like the 64-bit Intel Itanium processor. In these chips the transfer of data between processing units and the execution of operations take a comparable amount of time. In this sense it differs from previously existing multi-processor computational models, and creating efficient algorithms or compilers for these architectures requires new ideas for exploiting the available kind of parallelism.

Our Results

In Chapter 2, we study the problem of compiling programs for this kind of architecture. Compilation is the most basic problem for any computing system. It is the task of translating a high-level program into a set of basic instructions and schedule these instructions to be executed on the actual hardware. Clearly, we are interested in producing schedules that are as short as possible, since they enable faster execution of programs.

The problem we consider in this thesis is the following. We are given as input a set of basic instructions, and a precedence graph among them. This graph describes the fact that one instruction might depend on the result of another instruction, so the latter always has to be executed before the former. In addition, for each pair of dependent instructions, we have a description of the delay that has to elapse after executing the first instruction before the second instruction can be executed. This delay describes the time to compute the first instruction and to transfer the result of the operation to the instruction unit computing the second instruction.

Given such a description of a program to be executed, we want to generate an execution schedule, i.e. an assignment of the individual operations to instruction units and time slots, obeying the precedence constraints and necessary delays. Among all possible assignments, we are interested in one that minimizes the total length of the schedule, i.e. one that minimizes the so-called makespan.

We give a polynomial time algorithm that generates such optimal schedules, provided that the precedence graph forms a forest (a collection of trees), and the number m of instruction units and the maximal delay D are constants. Our problem can be described in the (somewhat cryptic) scheduling notation as

$$Pm \mid tree; p_j = 1; \ell_{i,j,a,b} \in \{0, 1, \ldots, D\} \mid C_{\max}.$$

In English, this means that we generate a program for m identical processors ("Pm"), the precedence constraints are a collection of trees, the instructions take unit-time ("$p_j = 1$") due to pipelining, and the delays $\ell_{i,j,a,b}$ between instructions a and b running on instruction units i and j are bounded by a number D. For such an instance, we have to generate a schedule minimizing the total completion time $C_{\max}$.

Our algorithm is based on two ideas. First, we use dynamic programming to generate an optimal schedule by brute force, as long as the set of available tasks is small. When too many tasks are available at the same time, we prune this set by keeping only those tasks that precede the largest number of other tasks. After generating a schedule in this manner, we apply a "merging algorithm" that incorporates the pruned tasks back into the schedule in an optimal way.

It is the latter merging technique that ought to be the most useful in a practical application of our algorithm. First of all, it runs in linear time, and is therefore very efficient, as opposed to the dynamic program, whose execution time is exponential in m and D. Secondly, it shows how the general scheduling problem can be reduced to the problem of scheduling only a few independent tasks at a time. For example, for a precedence graph consisting of many independent dags (directed acyclic graphs), a schedule for only the largest dags has to be created. To design fast heuristics for practical compilations, one can therefore restrict one's attention to this special case.

1.3 Peer-to-Peer Networks

In the past decade, large scale networks have become a fact of our daily lives. Examples are the Internet and cellular phone networks, both used by millions of people every day.

Large networked collections of (potentially low power) computing devices open new possibilities for distributed applications. One example are peer-to-peer applications, where all devices in a network are equivalent in the sense that there is no central entity controlling them. Moreover, peer-to-peer applications allow the dynamic joining and leaving of nodes in the network. This computational model was not studied much until a few years ago, because its large scale deployment was economically infeasible, and it was not clear what applications would benefit from the peer-to-peer approach.

This was all changed in 1999 with the Napster file-sharing system. Napster allowed any user connected to the Internet to take part in the (often illegal) sharing of files, mostly media files such as music MP3s or compressed movie files. The peer-to-peer approach was suited to this application since it involved many equal users, most of whom were not online all the time, implying that network membership was very dynamic. Although shut down after legal action, Napster was followed by many other file sharing systems, such as Gnutella and Kazaa.

The design of these systems was often ad-hoc, without a complete understanding of issues like scalability and fault-tolerance. Computer science research is now trying to catch up with reality, and many research groups have proposed improved protocols for data storage applications in peer-to-peer networks.

Our Results

We give several new protocols for peer-to-peer based storage applications. After a more thorough introduction to research on peer-to-peer networks in Chapter 3, we study load balancing issues in Chapter 4 and consider location-aware data storage schemes in Chapter 5. All our results are stated as improvements over the "Chord" protocol [SMK+01] developed at MIT, but most likely also apply to other similar peer-to-peer systems.

Load balancing. Load balancing is an important issue in any peer-to-peer application. Since all nodes are considered "equal" in the system, any computational, storage or communication load should be balanced fairly in the system. A number of protocols addressing this issue have been proposed and studied in the literature.

Our first result is an improvement over a load assignment scheme called "consistent hashing" [KLL+97], reducing the traffic overhead for maintaining the network by a logarithmic factor (Section 4.2). We then give the first formal analysis of a protocol that balances load by moving items from overloaded to underloaded nodes (Section 4.3). A variant of this protocol can also be used to implement efficient searching on ordered data in peer-to-peer networks (Section 4.4). The latter is an important feature, since data stored in a peer-to-peer network often has a natural ordering, e.g. the lexicographic ordering of filenames.

All these load balancing protocols are simple, and therefore should be easy to implement in practice. They follow the basic design principle that all decisions are made at a local level, i.e. no node has a complete overview of the network, and each node uses only information about itself and a few other nodes to execute the protocol.

Nearest neighbor queries in peer-to-peer networks. After giving protocols for load balancing, we shift our attention to designing protocols for peer-to-peer networks that take into account the physical location of nodes in the network. It is well-known that the access time to other nodes in a network depends on the relative location of the nodes. For example, two computers in the same building are likely to have a faster connection than two computers at opposite ends of the world. This has considerable consequences for the operation of data storage applications in peer-to-peer networks. In peer-to-peer data storage systems, copies of data items are often stored on several nodes. The reasons for this are twofold. On the one hand, it improves fault-tolerance by not losing access to an item once the only node storing it disappears from the network. On the other hand, it achieves load-balancing by spreading the accesses to popular items on several nodes. However, when requesting an item, there is usually no guarantee on which copy is returned by the system. For performance reasons, it would clearly be desirable to obtain a copy that is "close" to the requestor.

We provide a protocol that enhances peer-to-peer systems with the capability of finding the closest copy of a redundantly stored data item. This protocol increases the time requirements for maintaining the network, and for finding an item only by a constant factor over the basic data storage protocol.

The protocol can also be used to perform range searches, i.e. finding all nodes (or copies of an item) within a certain distance of a node, or finding the K closest nodes (copies) to a given node. The first variant (by distance) is more efficient to implement than the second (by number K), and therefore is more likely to be useful in practical applications.

An offline variant of our protocol has applications even beyond peer-to-peer networks (see Section 1.5).

1.4 Computations on Massive Data Sets

Computing on massive data sets has recently become an area of intense research in theoretical computer science and data mining. A "massive data set" is a data set that is far bigger than the memory of the computer processing it; today, they are easily of the order of several terabytes. This means that the data has to be kept on external storage media, which makes access to the data time-consuming, especially if done in a random access fashion.

In some sense, this is not a new area in computing, since the very early computers of the 1940s and 1950s had to deal with the same problem. Their internal memory, in the order of several kilobytes, was far smaller than many potential inputs. This lead to a significant amount of research on external memory algorithms, such as tape-based algorithms [Knu98].

In the 1970s and 1980s, the problem became less important, since memory sizes grew rapidly, but inputs still were mostly created by humans, so their size was limited and not too much bigger than available memory sizes.

With the use of computers in everyday life, however, things have changed for the worse again. Many applications of computers lead to the (digital) creation of massive data sets, such as phone company call logs, multimedia streams or customer click-logs on Internet web sites. Dealing with these data sets has again led to an interest in algorithms that use external storage to keep their data. Because of the relatively large cost of accessing the data, or restrictions on how the data can be accessed, this leads to computational models different from traditional computers.

Our Results

In Chapter 6, we consider the question of "what can be computed efficiently on massive data sets?". Since the answer depends on the computational model used, the first step towards answering this question is the definition of an appropriate model. It turns out that for massive data sets computations, there is no single universally accepted model, but instead a number of different models have been proposed over the years. In this thesis, we consider several of the most influential of these models, and prove theorems about their relative power. We also introduce a new model, the weakest known model to allow for the solution of many practical problems. This model also seems to capture well the computational power of modern computing hardware.

The goals of this chapter are twofold. On the one hand, we study several computational models in terms of their computational power, establishing a hierarchy among various complexity classes defined using these models. On the other hand, from a more practical perspective, we give several algorithms for a simple computational model, which might be useful in practice.

In more detail, we consider the following computational models (in order of increasing computational power).

1. *Streaming.* In the "streaming model" of computation, one assumes that the input is presented as a sequence of data items, that can be read once (or only a few times) from beginning to end (without being able to "go back"), by a machine whose memory can hold only a small fraction of the input [MP80, AMS99, HRR99]. Most problems can only be solved approximately in this model. It is generally considered too weak as a general model for computations on massive data sets, but is appropriate in cases where the input data is only available as a stream without the possibility of storing it, or where approximate answers are acceptable.

2. *Streaming+Sorting.* This model is introduced in this thesis. It extends the streaming model by allowing the writing and reading of intermediate streams, and the sorting of a stream according to simple comparison functions. This model is considerably more powerful than the streaming model, and allows for the solution of many basic problems on graphs, strings, and in geometry (Sections 6.7 and 6.8).

3. *Tape computations.* We consider the much-studied model [Knu98] of computation where tapes are used to read and write data. In this model one is allowed to do a few passes of sequential reading and writing on tapes. It turns out that by being able to read two or more tapes concurrently, the computational power is increased over the streaming+sorting model. The somewhat intricate proof of separation is given in Section 6.5.

4. *Linear External Memory Algorithms.* This is a restriction of "External Memory Algorithms" (see below) to mostly sequential accesses. This model has been implicitly defined and used by several researchers in the recent past [FFM98, FCFM00, BCDFC02]. It actually turns out to be equivalent within logarithmic factors to the tape computation model.

5. *External Memory Algorithms.* This model was introduced by Aggarwal and Vitter [AV88] to model the fact that most computers have access to fast main memory and slow, block-based external storage memory, e.g. hard-disks. In this model, one tries to minimize the number of reads and writes to the external storage medium.

1.5 Other Applications

Some of our results have applications outside of the computational models we consider. In particular this is true for the nearest neighbor search protocols developed for peer-to-peer networks in Chapter 5.

Our protocol directly leads to a dynamic, off-line data structure for nearest neighbor searching in "growth-restricted metrics". A metric (M, d) is called growth-restricted if for any $r > 0$ and $p \in M$, the number of points within distance r of p is a constant fraction of the number of points within distance $2r$ of p. So in some sense, this means that points come into view gradually when we increase the size of a ball around p.

Growth-restricted metrics appear in many application areas. For example, a random point set in a low-dimensional Euclidean space will be growth-restricted. The same is true for a random point set in a low-dimensional manifold of bounded curvature. Recent work on machine learning [TdSL00, Ten98] postulates that the feature vectors representing data points that are being analyzed form such a low-dimensional manifold in high-dimensional space. Since there is no explicit characterization of the manifold, traditional low-dimensional nearest neighbor algorithms cannot be applied. However, our algorithm can be used to resolve nearest neighbor queries efficiently on such point sets, allowing one to solve typical machine-learning tasks such as clustering.

1.6 Notation

In this thesis, by "with high probability in n" (whp in n) we mean a probability of the form $1 - n^{-c}$ for some constant $c \geq 1$. If n is clear from context, we omit an explicit reference to it.

Logarithms are always to base 2, unless stated otherwise.

Chapter 2

Multi-Processor Scheduling with Delays

2.1 Introduction

Multi-processor systems might have been the first well-studied computational model that differed from the standard single processor RAM-model. Following the introduction of multi-processor parallel computers, a considerable amount of work has been done on both the design of effective hardware architectures and the implementation of efficient algorithms for such architectures.

Two main issues constrain the design of parallel algorithms. First, the memory access of different processors has to be orchestrated such that there are no race conditions where different processors write different values to the same memory location, leading to a non-deterministic behavior of the algorithm, depending on who gets to write first.

Second, whenever data is transferred between processors (or to and from memory), one must consider the fact that this transfer takes several orders of magnitude more time than the execution of commands. Thus, an efficient parallel algorithm will restrict the inter-processor communication to a minimum.

Several Instruction Units on a Chip

Advances in chip design and manufacture have led to a tremendous decrease in the size of modern micro-processors. In fact, only about 5% of the surface of current processors is taken up by the part of the chip that executes instructions, the rest is devoted to the on-chip cache memory.

This has led to the creation of processors that contain several independent functional units. Each functional unit is capable of executing commands, such as arithmetic or logical operations. Frequently, not all functional units on a chip are able to perform the same set of operations. But the operations can be executed in parallel, making the single chip as powerful as a multi-processor system.

One commercially successful type of such processors are so-called *VLIW processors*. A VLIW processor is controlled by *meta-instructions* that combine the instructions for the individual functional units into one single instruction word, hence the name VLIW, which stands for "Very Long Instruction Word".

VLIW architectures have recently begun to appear in a variety of commercial processor and embedded system designs. The VLIW architecture is the basis for Intel's Itanium chip that was released commercially in 2001. It uses a new instruction set named IA-64 [Int00] that was developed jointly by Intel and Hewlett-Packard, and is based on EPIC (Explicitly Parallel Instruction Computing) – Intel's adaptation of VLIW. VLIW architectures have also been used in state-of-the-art Digital Signal Processor (DSP) designs, such as the popular Texas Instruments TMS320C6x series [Tex00].

One limitation of VLIW processors is that the functional units receive their instructions in a single word, which synchronizes the processors to each other, and limits scalability. The RAW project at MIT [TKM+02] tries to circumvent these problems by developing a processor architecture with multiple functional units, where each functional unit has its own instruction stream. They also allow the infrastructure for data transfer between functional units to be modified, leading to a processor architecture that is adaptive and scalable.

The role of the compiler is much more crucial for these architectures than it is for traditional processors. To exploit the inherent hardware parallelism, the compiler must combine basic operations into meta-instructions in an efficient way. When doing so, it has to observe the data dependencies between the operations and the time it takes to transfer data from one functional unit to another. Since hardware based acceleration schemes such as branch prediction or speculative execution become less powerful on these implicitly parallel architectures, it is the compiler that really determines the quality of the resulting code. This quality is especially important in embedded system design, where the code is only compiled once (making even lengthy compilation times acceptable), but an optimal performance is required of the resulting system.

In this chapter, we will give algorithms for constructing optimal execution schedules on these architectures for a large class of programs. Although they reside on a single chip, one can view the functional units as independent processors, and apply the previous research on multi-processor scheduling to this compilation problem. There are two crucial differences to traditional multi-processor scheduling, however. First, the number of processors in multi-processors systems is often large, while the number of functional units on a chip tends to be small, say less than 10. Second, for traditional multi-processor systems communication times between processors are magnitudes greater that execution times of single commands, but this is not true for our setting where all functional units reside on the same chip. Thus, when creating optimal execution schedules both kinds of delays (communication and execution time) have to be taken into account.

Multi-Processor Scheduling with Delays

More concretely, we will consider the problem of scheduling unit-length jobs on m identical parallel machines to minimize the makespan in the presence of *precedence constraints*, *precedence delays* and *communication delays*. Precedence constraints model dependencies between the tasks; if job j depends on job i, then job j must be executed after job i. Precedence delays $l_{i,j}$ impose relative timing constraints; job j cannot begin execution until at least $l_{i,j}$ time steps after job i completes. Communication delays $c_{i,j}$ impose delays across machines; if jobs i and j run on different machines, job j cannot begin execution until at least $c_{i,j}$ time steps after job i completes.

Previous algorithms for scheduling jobs on parallel machines consider either communication delays or precedence delays, but not both. In our work, we generalize both types of delays to a single *separation delay* $\ell_{i,j,a,b}$, where job j running on machine b cannot begin execution until at least $\ell_{i,j,a,b}$ time units after job i completes on machine a. Moreover, we overcome the restriction of previous algorithms where delays could only be either 0 or 1.

Note that this scheduling model exactly fits the problem of compiling for chip architectures with multiple functional units. Each meta-instruction can be thought of as a slice of time, and the functional units correspond to machines. Pipelining allows all jobs to have unit execution time. Precedence constraints encode the data dependencies, and delays encode the latencies: variable pipeline lengths and limited bypassing create variable precedence delays, and data movement between functional units creates communication delays. Since all the functional units are part of the same processor, precedence delays and communication delays are on the same order of magnitude, and should be considered together. Furthermore, fixing the number of machines and imposing a bound on the delays makes sense in this context; these quantities are a function of the physical characteristics of the chip, and are usually small. For example, Intel's Itanium chip has six functional units, and Texas Instruments' TMS320C6x has eight.

We give a polynomial algorithm for the case where the precedence graph between the jobs is a forest (i.e. a collection of in-trees and out-trees) and the delays are bounded by a constant D. Forest precedence constraints often occur in practice, for example, when processing expression trees or divide-and-conquer algorithms.

Our scheduling algorithm is based on dynamic programming, and uses a structural theorem, the so-called "Merge Theorem", shown in Section 2.4. The Merge Theorem applies to the scheduling of arbitrary precedence constraints (not just forests). If the precedence graph is a collection of independent dags, then the Merge Theorem states that any schedule S for the largest dags can be converted, in linear time, into a complete schedule that is either optimal or has the same makespan as S.

Organization

The remainder of this chapter is organized as follows. In Section 2.2 we formally state our scheduling problem and discuss related work. We then first concentrate on the simpler case of scheduling where the precedence graph is a collection of chains (Section 2.3). This algorithm relies on the previously mentioned Merge Theorem, which we prove in Section 2.4. The chain-scheduling algorithm is then generalized in Section 2.5 to an algorithm that allows the scheduling of tree precedence constraints. We briefly discuss the issue of using our algorithm in a practical compiler (Section 2.6), and conclude in Section 2.7 by stating some further research directions.

2.2 The Scheduling Problem

2.2.1 Problem Statement

The formal statement of the scheduling problem we will consider is the following. We are given a set of n jobs and m machines on which to execute the jobs, where m is a constant. Each job has unit processing time. There exists a directed acyclic precedence graph $G = (V, E)$ on the jobs V. With each precedence-constrained job pair $(i, j) \in E$, and pair of machines (a, b), there is an associated non-negative delay $\ell_{i,j,a,b}$ bounded by a constant D. The output is a schedule assigning a job to each processor and time slot. A schedule is legal iff it includes all jobs, and for all precedence-constrained job pairs $(i, j) \in E$, if job j runs on machine b at time t, then job i must be scheduled on some machine a *before* time $t - \ell_{i,j,a,b}$ (i.e., there must be $\ell_{i,j,a,b}$ time units *between* them).

We denote the completion time of job j as C_j. We are concerned with minimizing the makespan, $C_{\max} = \max_j C_j$. Let $C^*_{\max}$ be the optimal value of $C_{\max}$. Extending the notation introduced by Graham et al. [GLLR79], we can denote the problem we consider as $Pm \mid prec; p_j = 1; \ell_{i,j,a,b} \in \{0, 1, \ldots, D\} \mid C_{\max}$.

We can also allow multiple instances of the same job to be scheduled on different machines; this is called *job duplication*. Allowing job duplication can make a difference in the makespan of a schedule when computing the same value twice is more efficient than transferring the value across machines (see Section 2.5.1).

2.2.2 Our Results

We give a polynomial-time algorithm for the problem where the precedence graph G is a forest: $Pm \mid tree; p_j = 1; \ell_{i,j,a,b} \in \{0, 1, \ldots, D\} \mid C_{\max}$. The algorithm works with or without job duplication allowed on a job-by-job basis.

Another important contribution of this work is the Merge Theorem:

Theorem 2.1 (Merge Theorem)
Consider an instance of $Pm \mid prec; p_j = 1; \ell_{i,j,a,b} \in \{0, 1, \ldots, D\} \mid C_{\max}$ where the precedence graph G contains at least $2m(D+1)-1$ independent dags. Given a schedule with makespan T for only the jobs from the largest $2m(D+1)-1$ dags, one can construct in linear time a schedule for all jobs with makespan $\max\{\lceil \frac{n}{m} \rceil, T\}$.

Since this theorem holds for *any* dag, not just trees, it shows that any heuristic or approximation algorithm for scheduling only the jobs from large dags can be extended into an algorithm for scheduling all jobs. The theorem might also be applied to single dags after they have been broken into independent pieces. Furthermore, since a schedule of length $\lceil \frac{n}{m} \rceil$ is clearly optimal, the new algorithm will have the same performance guarantee as the original algorithm with only a linear time additive cost in running time.

2.2.3 Related Work

Our result is more general than previous known polynomial algorithms in both the precedence delay and the communication delay communities for optimally scheduling trees on a fixed number of processors, since previous results assumed at most unit time delays.

Polynomial Algorithms: Precedence Delays

Precedence delays have been used to model single-processor latencies that arise due to pipelined architectures. Bernstein and Gertner [BG89] use a modification of the Coffman-Graham algorithm [CG72] to solve $1 \mid prec; p_j = 1; l_{i,j} \in \{0, 1\} \mid C_{\max}$. Finta and Liu [FL96] give a polynomial time algorithm for the more general $1 \mid prec; p_j; l_{i,j} \in \{0, 1\} \mid C_{\max}$. Both of these algorithms crucially depend on assuming unit-delays between jobs.

Polynomial Algorithms: Communication Delays

In the classical models of parallel computation, communication delays are orders of magnitude larger than precedence delays, so algorithms for scheduling on parallel machines have generally ignored precedence delays. A survey by Chrétienne and Picouleau [CP95] gives an overview of the work in this area.

All previous polynomial-time algorithms for a bounded number of machines work only for the special case of unit communication delays. Varvarigou, Roychowdhury and Kailath [VRKL96] show that $Pm \mid tree; p_j = 1; c_{ij} = 1 \mid C_{\max}$ is solvable in time $O(n^{2m})$ by converting the tree into one without delays. This conversion relies heavily on the fact that the delays are unit-length. The special case $m = 2$ was shown to be solvable in $O(n^2)$ time by Picouleau [Pic92], and was later improved to linear time by Lenstra, Veldhorst and Veltman [LVV96], using a type of list scheduling.

Finta and Liu [FLMB96] give a quadratic algorithm for $P2 \mid SP1; p_j = 1; c_{ij} = 1 \mid C_{\max}$, where $SP1$ are *series-parallel-1* graphs, a subclass of series-parallel graphs. There has also been some work on approximation algorithms for an arbitrary number of machines. Möhring and Schäffter [MS95] give a good overview of this area.

Several authors (e.g. [JKS89, PY88]) have considered related problems where the number of processors is unbounded, i.e. the schedule can use as many processors as desired. However, that model is fundamentally different from the one we study, since optimal schedules usually make extensive use of the unlimited parallelism.

Hardness Results

Even without any delays, the problem is NP-hard if the precedence relation is arbitrary and the number of machines is part of the input. This is the classic result of Ullman [Ull75], showing NP-hardness of $P \mid prec; p_j = 1 \mid C_{\max}$. Lenstra, Veldhorst and Veltman [LVV96] show that the problem is still NP-hard when the precedence graph is a tree and there are unit communication delays ($P \mid tree; p_j = 1; c_{ij} = 1 \mid C_{\max}$).

Engels [Eng00] proves NP-hardness for the single-machine case when the precedence constraints form chains, and the delays are restricted to be either zero or a single input value, i.e., he shows $1 \mid chain; p_j = 1; l_{i,j} \in \{0, d\} \mid C_{\max}$ to be strongly NP-hard, where d is an input to the problem.

When the processing times are not unit, the problem is also NP-hard. Engels [Eng00] shows that scheduling chains with job processing times of either one or two and constant precedence delays, i.e., $1 \mid chain; p_j \in \{1, 2\}; l_{i,j} = D \geq 2 \mid C_{\max}$, is strongly NP-hard.

Thus the only natural gap between our result and NP-hard problems is the generalization to arbitrary precedence structures on a fixed number of machines, i.e., the problem $Pm \mid prec; p_j = 1; \ell_{i,j,a,b} \in \{0, 1, \ldots, D\} \mid C_{\max}$. However, this gap comes as no surprise, since the famous 3-processor scheduling problem ([GJ79], problem [OPEN8]) is a special case. It turns out that even an algorithm for the one-processor version where all delays are equal to three ($1 \mid prec; p_j = 1; \ell_{i,j} = 3 \mid C_{\max}$) could be used to solve instances of 3-processor scheduling ($P3 \mid prec; p_j = 1 \mid C_{\max}$). The reduction is straightforward.

2.3 Scheduling Chains

We will now state a simple version of our algorithm for the case where G is a collection of chains, and there is only one processor ($m = 1$). Later, we give a more general version that works for trees on parallel processors. The algorithm given here is slightly less efficient than what we can achieve; it runs in time $O(n^{3D+1})$. We will describe how to improve this to $O(n^{2D+1})$ at the end of the section. We give this slightly less efficient algorithm because it establishes some of the machinery used for the the general case.

The Merge Theorem (to be proved in the next section) shows how to construct an optimal schedule, assuming we know how to optimally schedule the $2m(D + 1) - 1$ largest chains in the precedence graph. This immediately suggests an algorithm:

1. **Dynamic Program.** Use a dynamic program to optimally schedule the $2m(D+1)-1$ largest chains in the input, setting aside the other chains for the moment.

2. **Merging.** Apply the Merge Theorem to schedule the chains we set aside during the dynamic program.

The dynamic program we will use can be thought of as finding the shortest path through a state space, where state transitions correspond to the scheduling of jobs for a single time step. Every state encodes "where we are" in the current schedule; it records the jobs available to be scheduled on the upcoming time step, as well as a recent history of the current schedule, which we will use to determine when jobs become available in the future. More precisely, states have the form $\langle A, P \rangle$, where

- A is a set we call the *active set*. This is the set of currently *active* jobs, i.e. the jobs which can be scheduled in the next time step.

- P is a vector of length D, whose entries either contain jobs or are empty. These are the *past jobs*, the jobs that have been scheduled within the last D time steps. Essentially, P is a "window" into the last D time steps of the schedule.

The following operations define both the legal transitions between states and the scheduling/status updates done by the dynamic program when passing through the transition:

1. Schedule a job j in A. Shift the window P one time step forward, yielding P_{new}, whose last entry is j. It is also possible to not schedule any job (this is the only possibility if the active set is empty). In that case, P_{new} will have an empty last entry.

2. Use the information in P to determine the set B of jobs that become available for this new time step (the delays from their parents have just elapsed). Since the delays are bounded by D, the information in P is sufficient to make this determination.

3. Set A_{new} equal to the new set of active jobs, $(A \setminus \{j\}) \cup B$. The new state is $\langle A_{new}, P_{new} \rangle$.

Creating an optimal schedule now corresponds to finding a shortest path from the *start state* $\langle A, P \rangle$ (where A consists of the roots of the $2D+1$ largest chains, and P is an empty vector), to an *end state* (one where A is empty, and all jobs in P have no children that are not also in P).

The above dynamic program is enough to schedule chains on a single processor ($m = 1$) in polynomial time. This is because we can bound the size of the active set A. The set A can contain at most one job per chain, since no two jobs from the same chain can be active at the same time. The size of A is therefore limited to $2D+1$, so that there are only $O(n^{2D+1})$ possible values for A. Since there are $O(n^D)$ possible values for P, the number of states is bounded by $O(n^{3D+1})$. This bound is polynomial, and therefore we can find the optimal schedule for the largest $2D+1$ chains in polynomial time.

The second step of our algorithm for chains (the Merging step) is quite simple. Suppose the resulting schedule for the largest $2D+1$ chains has length T. We then apply the Merge Theorem to construct a schedule of length $\max\{n, T\}$ that contains all jobs. Since T was a lower-bound on the optimal solution for the whole problem, the resulting schedule must be optimal.

As a side note, we can reduce the size of the state space for chains and one processor to $O(n^{2D+1})$. Each state stores, for every chain, the last job executed and how long ago it was executed. This is enough information to determine A and P as above.

2.4 Proof of the Merge Theorem

We begin by proving the Merge Theorem for the case where we have chain precedence constraints and only one processor (G is a collection of k independent paths, $m = 1$). This proof establishes all of the techniques used for the general case and is less obscured by details. We then describe the natural generalization to dags, parallel processors and general separation delays.

The key technique used in the proof of the Merge Theorem is a round-robin scheduling of a set of $D + 1$ or more independent chains. By this we mean that we cycle through the chains in some fixed order, and always execute the current first job in each of them. This creates a schedule without holes (i.e. an optimal schedule) if all chains initially have the same length. Unfortunately, this will usually not be the case for actual inputs. In the first half of the proof below, we will therefore modify the schedule to get us into a position where we can apply the round-robin technique.

2.4.1 Special Case: Chain Precedence Constraints, One Processor

Our goal is the following: Given a schedule of the $2D+1$ largest chains that finishes at time T, we want to construct a complete schedule for all k chains that finishes at time $\max\{n, T\}$. As a running example consider the instance shown in Figure 2-1. This example consists of 7 chains with a total of $n = 21$ jobs. The maximum precedence delay is $D = 2$. Figure 2-1a shows a feasible schedule for the $2D + 1 = 5$ largest chains with makespan $C^*_{\max} = n$. We will construct the new schedule in four steps.

Step 1: Truncating. Let n_i be the number of jobs in the i-th largest chain. We begin by removing the last n_{2D+1} jobs in each of the scheduled chains from the current schedule (as in Figure 2-1b, where $2D + 1 = 5$ and $n_5 = 2$). We call these deleted sub-chains the *tails*. Note that we have removed $2D + 1$ tails with exactly n_{2D+1} jobs in each tail.

Step 2: Shifting operations. Next, we modify the schedule with the tails removed by shifting jobs so that they are executed as early as possible. Beginning at the first time slot, we traverse the schedule through time T. Whenever we encounter a hole in the schedule (a time slot that does not have a job scheduled in it), we try to fill that hole by moving a job earlier in the schedule (as in Figures 2-1c-g).

We can always fill a hole with a job that is currently scheduled later if, at the position of the hole, at least $D+1$ of the chains are *active*, i.e., they have not yet been scheduled up to the point at which they were truncated. To see why this is possible, note that if $D + 1$ chains are still active, at least one of these chains has not been executed during the last D time steps before the hole. Therefore, if we move the next job of that chain into the current hole, it will be executed at least D time units after its predecessor. The precedence delay is satisfied after this move since the delay is at most D.

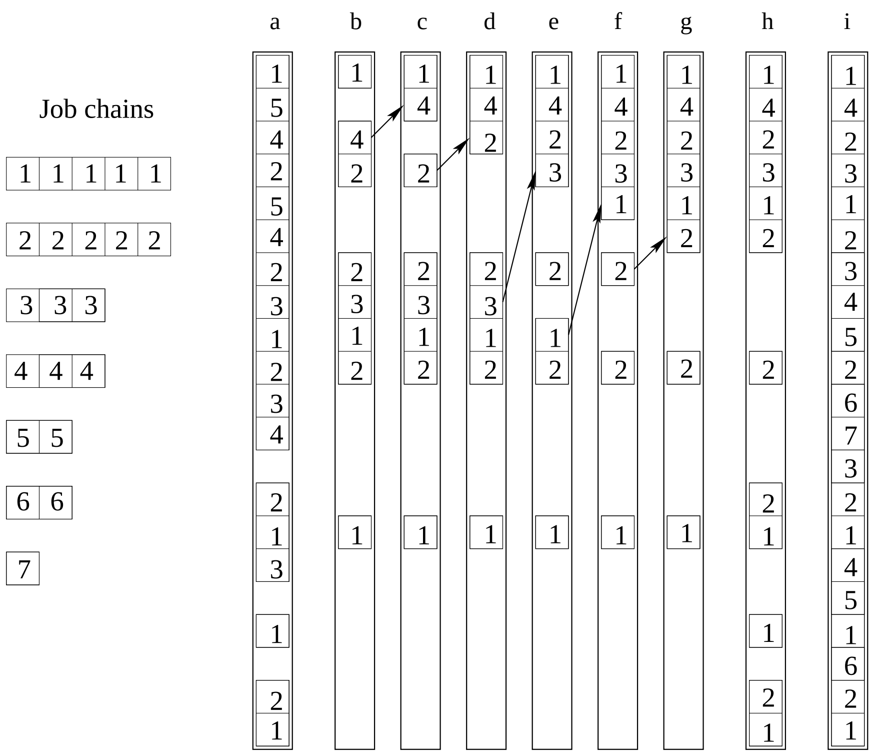


Figure 2-1: Problem instance (on left) and construction of an optimal schedule (on right), for $D = 2$. The instance is composed of two chains with five jobs, two chains with three jobs, two chains with two jobs and one chain with one job. All delays between consecutive jobs in a chain are 2. The schedule is constructed as follows: **a)** A schedule for the large chains. **b)** Step 1, deleting the tails of the large chains. **c-g)** Step 2, shifting jobs earlier in the schedule until at most D chains remain active. **h)** Step 3, putting the tails of the active chains back into the schedule. **i)** An optimal schedule after inserting the remaining jobs using the round-robin of Step 4.

After repeatedly moving jobs to fill holes, we will either finish shifting all of the truncated chains or reach the first hole that we cannot fill without violating a delay constraint (as in Figure 2-1g). The resulting schedule is tight before that hole (i.e. there are no holes before it), and there are at most D of the truncated chains active at that position (recall that we can always move a job if more than D chains are still active). In the example (Figure 2-1g), chains 1 and 2 are still active at the first hole.

Step 3: Re-inserting some of the tails. We now reinsert the tails of the (at most D) chains that are still active at the first hole (as in Figure 2-1h). We reinsert these jobs at their positions given by the original schedule. These positions in the schedule are still unoccupied, since jobs were only moved to time slots earlier than the first hole. Moreover, the makespan of the total schedule is still at most T.

Step 4: A Round-Robin for Scheduling Tails and Short Chains. We are now left with the tails of at least $D+1$ chains, each containing exactly n_{2D+1} jobs, whose truncated versions finished before the first hole (call these tails the **blue** chains) and $k-(2D+1)$ short chains, each containing at most n_{2D+1} jobs (call these the **red** chains). The red chains are the ones that were not among the $2D+1$ largest. In the example, the sub-chains consisting of the last $n_{2D+1} = 2$ jobs in chains 3, 4 and 5 are blue, and chains 6 and 7 are red.

Completing the schedule is done by filling holes with the remaining jobs in a round-robin fashion, i.e., we cycle through the chains (both the red and blue chains) in some fixed order, inserting the next job of each one, until they are all scheduled.

We have to be a bit careful about the first D holes we fill in this process, since the blue chains cannot start too close to their predecessors from their original chain.

This problem can be solved by systematically choosing the order we cycle through the chains. Since there are at least $D+1$ blue chains, one of their predecessors has not been executed during the last D steps, so we can safely schedule that chain first. Among the remaining blue chains, one has not been executed in the last $D-1$ steps, and therefore it can be scheduled second, and so on. We fix this order of the blue chains (in the example, we let this order be 3,4,5), and then follow it with any order of the red chains (6,7 in the example).

Since all blue chains have the same length, they all finish on the same round. Furthermore, the red chains finish on or before this round, since they are no longer than the blue chains. Therefore, every round consists of at least $D+1$ different chains, and we can fill every hole until the round-robin ends.

Thus, we have scheduled all jobs, obeying the chain precedence constraints and the precedence delays (as in Figure 2-1i). If step 4 did not fill all the holes that existed after step 3, then we know that our schedule still has makespan at most T. Otherwise, the new schedule has no idle time, and has makespan n.

Also, the running time of this construction can be made linear in the number of jobs. Determining n_{2D+1} can be done using a $O(n)$ time rank-computation, or by a linear pass through the n_i, maintaining the $2D+1$ largest values (noting that $2D+1$ is a constant). For step 1, a linear pass through the schedule suffices, maintaining a counter for each chain to record how many jobs of the chain have already been seen, and removing a job if it is among the last n_{2D+1} jobs of a chain. Step 2 can also be accomplished with a linear pass through the schedule resulting from step 1; we maintain a list of all not-yet-completed chains, and also a list of the D chains scheduled in the last D time steps. This information is sufficient to be able to determine a chain to shift a job from in time $O(1)$. When there are only D active chains left, we have a list of these chains, allowing us to complete step 3 in linear time as well. For step 4, it takes a linear pass to determine the order in which the blue chains should be scheduled (which is the order in which their last jobs appear before the first hole). The round-robin can then also be carried out in linear time.

2.4.2 Dags, Multiple Processors, and General Separation Delays

There is a natural generalization of the above construction to dags, multiple processors and general separation delays.

Given a schedule with makespan T for the largest $2m(D+1)-1$ dags, we must construct a schedule for all the dags with makespan $\max\{\lceil \frac{n}{m} \rceil, T\}$. We follow the same four basic steps as before.

Previously, for chains, the first step of the construction removed the last n_{2D+1} jobs from the large scheduled chains. Now, in the general case we remove the $n_{2m(D+1)-1}$ jobs from each dag that are *scheduled* last (ties are broken arbitrarily). In step 2 of the chains case, we shifted jobs to earlier in the schedule as long as at least $D+1$ of the chains were still active. To be able to shift jobs in the general case, we now need $m(D+1)$ dags active (see step 4 below). Step 3 is identical; we reinsert the jobs from the dags that are still active at the first hole we cannot fill.

At step 4 in the general case, there are at least $m(D+1)$ **blue** dags, each containing the same number of jobs, and several smaller **red** dags (the ones that were not in the initial schedule). In step 4 of the chains case (the round-robin fill-in step), notice that we made no assumptions about the delays between the jobs in the red and blue chains other than that they were bounded by D. So for dags, we first topologically sort the dags in an arbitrary way, making them chains. Then we perform the round-robin as before. The red chains finish first, the blue chains all finish on the same round, and we have either finished before time T, or filled every hole. The running time of each step is still linear in the number of jobs. □

2.5 Scheduling Trees

In this section we give a polynomial time algorithm for scheduling jobs with tree precedence constraints, separation delays, and possible duplication of jobs. We first assume that the precedence graph G forms a collection of out-trees. By reversing the time-line, the algorithm can also be used to schedule a collection of in-trees. Later, we extend the algorithm to support scheduling a collection of in- and out-trees.

2.5.1 Job Duplication

Before we turn to the actual algorithm, we will briefly discuss job duplication. When scheduling jobs under separation delay constraints, it sometimes pays to execute a job multiple times on different processors. This is especially true if many other jobs depend on this one job, and it is time-consuming to move data from one processor to another.

The simplest example is an out-tree consisting of three nodes: a root with two children. The delay between the root and its children is 0 if they run on the same processor, and 10 otherwise. Suppose we want to schedule this instance on two processors. Clearly, without duplication, the shortest solution uses three time steps (schedule all three jobs on one

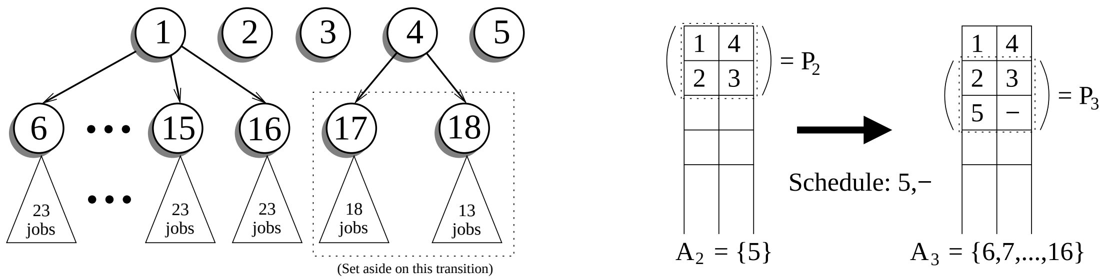


Figure 2-2: Example of an input tree and a state transition from $\langle A_2, P_2\rangle$ to $\langle A_3, P_3\rangle$. The maximum delay D is 2 and all delays $\ell_{i,j,a,b}$ are equal to 2. There are two machines ($m = 2$). The active set A_2 consists of only job 5, as it is the only one available. The transition schedules job 5 on the first machine, and nothing on the second machine. Jobs 6 through 18 all become available, but only $2m(D+1)-1 = 11$ can be in A_3, so jobs 17 and 18, the ones with the fewest number of jobs in their subtrees, are set aside, along with the jobs in their subtrees. The new active set A_3 is $\{6, 7, \ldots, 16\}$.

processor). However, if we execute the root on both processors, we can execute both children in the next time step, resulting in a schedule of length two.

While duplication is clearly useful, it does not appear in completely arbitrary ways in an optimal schedule. In fact, there always exists an optimal schedule in which no two copies of a job are executed more than D time steps apart. To see this, consider a job that is executed twice, where the second execution occurs more than D time steps after the first. In that case we can just delete the second execution, since all its children were already available at the time the second copy was executed.

2.5.2 Overview of the Algorithm

We now turn to the scheduling algorithm for trees. The algorithm consists of the same two phases as the algorithm for chains given in Section 2.3: a dynamic program and a merging step. The states in our dynamic program will be similar to the ones in Section 2.3. They are of the form $\langle A, P\rangle$, where A contains jobs available on all processors and P contains a 'window' into the past D time steps of the schedule.

The transitions given in Section 2.3 are not general enough to schedule trees, since the number of concurrently active jobs in A may grow without bound, e.g. if a job has many children that all become available at the same time. If the size of A is not bounded, the size of our state space will not be polynomial. To overcome this problem, we limit the maximum number of jobs in A to be $2m(D+1)-1$. Whenever a transition increases the number of active jobs above that number, we *set aside* the jobs from all but the largest $2m(D+1)-1$ trees rooted at these potentially active jobs. In the Merging step we will include the jobs from these set aside trees into the schedule.

Note that in our dynamic programming algorithm, the set aside trees will actually depend on the *path* taken through the state space. So when solving our dynamic program it is important to remember this path, since the states of the dynamic program by themselves do not contain enough information to perform the Merging step of our algorithm. This differs from many dynamic programs where the states contain enough information to reconstruct the solution.

To simplify the following presentation, we introduce the notion of the *status* of a job. This status is not explicitly stored in the state, but is useful when we think about how the dynamic program creates a schedule. We say a job is:

- ***active***, if it can be scheduled right away on *all* processors, since all delays from its predecessor have elapsed,
- ***waiting***, if it has not been scheduled, and there is a processor on which it cannot run yet (because its predecessor has not been executed yet, or not long enough ago),
- ***scheduled***, if it has already been scheduled on some processor, or
- ***set aside***, if the dynamic program has decided to ignore it, and it will only be scheduled later in the Merging step.

2.5.3 A New Dynamic Program

The state space contains all pairs $\langle A, P \rangle$, where A is the active set, limited to $2m(D+1)-1$ jobs, and P is an $m \times D$ *matrix* recording the last D time steps of the schedule. This means that we have $O(n^{3mD+2m-1})$ states in the dynamic program, making it possible to find a shortest path in polynomial time.

The state transitions are more complex than in the algorithm Section 2.3. An example state transition can be found in Figure 2-2. If we are at a state $\langle A, P \rangle$, we can go to a new state $\langle A_{new}, P_{new} \rangle$, as follows:

1. Choose jobs $j_1, j_2, \ldots, j_m$ to be executed on the m processors. Set their status to *scheduled*. Each job j_i can be one of the following:
   - nothing (no job scheduled)
   - any job in the set A
   - any job in the matrix P that is executable on processor i at the current time step (job duplication)
   - any child of a job in matrix P that is executable on processor i (but not all processors) at the current time step (partially available job)
2. The new matrix P_{new} is P shifted forward by one row, with $(j_1, j_2, \ldots, j_m)$ being the new last row. All jobs that were in the first row of P (the one that got shifted out) that are still in P_{new} (due to job duplication) are removed from P_{new}.
3. Using the information in P, determine the set of jobs B that on *this* step become available on *all* processors, and have not been executed before, and set A_{new} to $(A \setminus \{j_1, j_2, \ldots, j_m\}) \cup B$.
4. If A_{new} has more than $2m(D+1)-1$ elements, remove all but the $2m(D+1)-1$ "largest" jobs from the set, where "largest" is measured in terms of the size of the sub-tree rooted at the job. These removed jobs, along with all the jobs contained in their sub-trees, are *set aside*. They will be dealt with in the Merging phase.

The *start state* of the dynamic program is $\langle A_0, P_0 \rangle$, where A_0 consists of the roots of the $2m(D+1)-1$ largest trees, and P is the empty matrix. The *end states* have the form

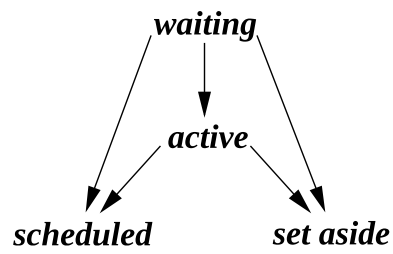


Figure 2-3: The life of a job.

$\langle A, P\rangle$ where A is empty (i.e. no jobs are active), and all jobs in P either have no children, or their children are also in P (i.e. no jobs are waiting).

As we traverse the path from a start state to an end state, the status of each job evolves as in Figure 2-3. It follows that at the end of the path, every job is classified as either *scheduled* or *set aside*.

2.5.4 Merging and Correctness

A path of length T from a start state to an end state in the state-space defined above gives a schedule of length T for part of the forest we are scheduling. We need to show how the jobs *set aside* by the path can be merged back into the schedule. In the remainder of this section, we will show two lemmas. The first lemma will establish that we can find a path in the state space that can be converted into an optimal schedule via a Merging step. The second lemma will show how to perform this Merging step.

Before stating the lemmas, we need three definitions. First, we define the set U_q for a state q, which contains all the jobs which *must* appear after state q in any legal schedule (these are the jobs which are *available* or *waiting* at that state). This set is completely determined by the information contained in $\langle A, P\rangle$.

Definition 2.2 (Dependent Jobs)
For a state $q = \langle A, P\rangle$, let U_q contain all jobs in A, all descendants of jobs in A, and all descendants of jobs in P that are not yet available on all processors, and that are not in P themselves. □

Now we define the deadline of state q as an upper bound on q's position on a path in state space such that all dependent jobs of q can still fit into the schedule without making it longer than $C^*_{\max}$.

Definition 2.3 (Deadline)
Let $q = \langle A, P\rangle$ be a state. The deadline *of q is the value* $\left\lfloor C^*_{\max} - \frac{|U_q|}{m} \right\rfloor$. □

In any path in state space that corresponds to an optimal schedule, every state must appear before its deadline. We formalize this in a definition.

Definition 2.4 (Admissible Path)
A path in the state space from the start state to an end state is called admissible *iff for all x from 0 to $C^*_{\max}$, the x-th state on the path has a deadline of at least x.* □

We will now show that an admissible path always exists, that it can be found in polynomial time, and how to convert it into an optimal schedule.

Lemma 2.5 (Dynamic Program Correctness)
There always exists an admissible path that can be found in polynomial time.

Proof: An admissible path, if it exists, can easily be found by breadth-first search through the state space of the dynamic program we constructed earlier. The deadline of each state can be determined beforehand[1]. At depth x of the search, we extend the search only to states with a deadline of at least $x + 1$.

Now we show that such a path always exists. We show this by constructing an admissible path $(q_0, q_1, ..., q_{C^*_{\max}})$ using an optimal schedule S as a template. We assume that S has no unnecessary job duplications (jobs whose removal from the schedule would maintain feasibility).

We will proceed along the schedule, and at the x-th step take the state transition from $q_{x-1} = \langle A, P\rangle$ to q_x that corresponds to executing the jobs in the x-th time slice of S that are in $P \cup U_{q_{x-1}}$. There must be such a transition, because for every job in $P \cup U_{q_{x-1}}$ that is executed in S at that time slice, it is either in A, or its parent appears in P at the same position as it appears in S (easily shown by induction).

It remains to show that the path thus constructed is admissible. Note that when we are at state q_x along the path, all jobs in U_{q_x} have to appear after time slot x in the schedule S. Because we are executing "down" the trees, and we never add to a set U_{q_x} to obtain $U_{q_{x+1}}$, so we always have $U_{q_x} \subseteq U_{q_y}$ if $x > y$. So, if $x > y$, and a job in $U_{q_{y-1}}$ appears in S at time step y (and so is not in U_{q_y} by construction), it will not be in U_{q_x}. This means that none of the jobs in U_{q_x} can appear at or before the x-th time step in S, and therefore all appear after it. But this implies $\lceil |U_{q_x}|/m\rceil \leq C^*_{\max} - x$, which shows that the path is admissible. □

Now that we have a schedule for part of the tree, we need to merge the jobs that we set aside back into the schedule. Here is where we use the Merge Theorem.

Lemma 2.6 (Merging)
Given an admissible path, an optimal schedule can be constructed in time $O(n^2)$.

Proof: An admissible path can be directly converted into a schedule S of the same length that contains all but the jobs which were *set aside*. We now show how to incorporate the set aside jobs into the schedule, without making it longer than the optimal schedule.

We do this by traversing the path from its end to its beginning. When we reach a state q_x at which jobs were set aside, we include them into the schedule as follows. Since trees were set aside at that state, there must be $2m(D + 1) - 1$ larger trees rooted at the jobs in q_x's active set. The jobs in these 'active' trees are already in the schedule, since either they were scheduled by the admissible path, or they were set aside later, in which case we already merged them into the schedule (recall we are traversing it backwards).

This means we can apply the Merge Theorem to merge the set aside trees into the schedule. Since we started with an admissible path, we know that the number of jobs not yet *scheduled* at q_x does not exceed $m \cdot (C^*_{\max} - x)$, the available room in the schedule. Therefore, merging the set aside trees does not make the schedule longer than the optimal schedule. We repeat this procedure for all states and obtain an optimal schedule.

Since applying the Merge Theorem for every state costs linear time, and there might be up to n states on the path, the total time for the merging operation is $O(n^2)$. □

[1] Note that we have to know $C^*_{\max}$ to compute the deadline. But since $C^*_{\max} \leq nD$, we can find the value using binary search with a multiplicative increase of $O(\log n)$ in running time.

2.5.5 Scheduling Arbitrary Forests

We now show how to adapt the algorithm so that it optimally schedules arbitrary forests, i.e. collections of both in- and out-trees. Observe that in the dynamic program in the above algorithm, our path through state-space determines which jobs are "set aside". Our algorithm constructed the schedule from beginning to end, i.e. by finding a shortest path from the start state to an end state. But we equally well could have constructed the schedule by searching from an end state to the start state – the path would still have determined the set aside jobs.

For scheduling both in-trees and out-trees at the same time, we are going to run a combination of two dynamic programs. The schedule for the out-trees is going to be constructed by going forward in the state space, while the schedule for the in-trees is constructed going backwards. The new state space is thus the product of the state spaces for scheduling just the in-trees and just the out-trees. That start states are products of the start state of the out-tree dynamic program and the end states of the in-tree dynamic program, and the end states are similar products with the roles of in- and out-trees reversed.

In constructing a schedule, we will only traverse states that do not schedule jobs from both in-trees and out-trees on the same processor in the same time slot. In other words, the two schedule do not intersect each other.

Any path through this new state space defines "set aside" jobs for both the in- and out-trees, and we have to apply the Merge Theorem twice to incorporate these jobs into the schedule. This causes a slight problem: since the schedule for in-trees was constructed "backwards", the Merge Theorem will insert jobs forward into the schedule, i.e. by prepending them to the schedule. To make sure that these jobs do not spill over the beginning of our schedule, we have to make sure that there is enough room (i.e. unassigned processor/time slots) at the beginning of the schedule to insert all these jobs. This can be accomplished by augmenting the states by a number that keeps track of the empty slots in the schedule so far, and making sure that the number of set aside jobs in the in-trees part of the schedule never exceeds this number of empty slots.

In summary, the number of states in the new program is $O(n^{6mD+4m-1})$, and thus finding an optimal schedule is still possible in polynomial time.

2.6 Practical Considerations

The part of this work that seems most useful for an actual compiler implementation is the algorithm for the Merge Theorem in Section 2.4. It can be used to combine schedules for a set of independent dags of precedence constraints in linear time. Given an optimal schedule for the largest $2m(D+1)-1$ among a set of independent dags, it creates a schedule for all dags in the set that is no longer or optimal.

Using our tree-scheduling algorithm in an compiler is difficult for two reasons. First, program dependency graphs often are not trees (even though parts of them can be very close to trees). So one would have to apply some mechanism or heuristic to cut the dags into trees, which will likely lead to sub-optimal schedules. Second, the dynamic programming portion of our algorithm is quite slow if m and D are of non-trivial size. So in practice one would likely use heuristics or approximation algorithms to generate a schedule of the largest trees. Only if an optimal program is desired for performance reasons, and the execution time savings are worth the considerable compilation time, should one consider using the full algorithm as stated in this chapter.

2.7 Open Problems

We have given the first polynomial-time multi-processor scheduling algorithm for tree-based precedence constraints that impose precedence and communication delays. As opposed to previous results, separation delays $\ell_{i,j,a,b}$ can depend on jobs and machines, and can have values other than 0 and 1, as long as they are bounded by a constant D. That makes our algorithm more general and applicable to the instruction scheduling for processor architectures with multiple functional units. The potentially long running time of the algorithm is acceptable to embedded system designers since the software is compiled only once and an optimal performance is required of the resulting system.

The algorithm for trees uses an unconventional dynamic program, where partial paths in state space do not correspond to partial schedules, but rather have to be transformed into a solution during the Merging phase.

Good Scheduling Heuristics: The running time of our algorithm depends exponentially on the number of processors m and maximum delay D, making it impractical for large values of these constants. However, it is the dynamic programming part of the algorithm that incurs this runtime; the merging step only takes $O(n^2)$ time.

This suggests using an heuristic instead of the optimal dynamic program to produce a path through the state space. The Merge Theorem can then be used to incorporate the remaining trees into the schedule. Finding good heuristics, from both a theoretical and an experimental point of view, is a very interesting open problem. Some work in this direction has been performed by Amarasinghe et al [AKLM02].

Different Functional Units: Another important extension would be to generalize our algorithm to capture different job *types*. In real processors, the functional units are usually not identical – they can perform different subsets of all operations. For example, one functional unit might be able to execute arithmetic operations, while another can perform flow-control operations. So while we assumed, for simplicity, that all functional units are able to execute all jobs, this assumption is not true in practice. It would be interesting to see whether our algorithms can be extended to that setting.

Scheduling Dags: A final intriguing question is whether our techniques can be extended to the case where G is an arbitrary dag. The Merge Theorem still holds for these inputs. But our dynamic program critically uses the fact that once a branch occurs, the subtrees are completely independent. A more complicated dynamic program might get around this problem without a large increase in the size of the state space. As already mentioned in the introduction, this is very likely a hard problem, since an algorithm for just the single-processor case with $D = 3$ can be used to solve the famous open 3-processor scheduling problem.

Chapter 3

Peer-to-Peer Systems

3.1 Introduction

One of the most active research areas in computer systems during the past decade is networking. The last ten years have seen networking technology transformed from niche product to ubiquitous commodity hardware. A prime example is the proliferation of the Internet in our daily lives. Ten years ago, the Internet was only used by a small number of academics, but the invention of the World Wide Web, followed by the Internet Boom, has led to a tremendous growth in audience; hardly a computer is sold today without the ability to connect to the Internet.

Another example is wireless networking, as used by cellular phones and wireless computers. In 1993 the providence of a self-selected few who were willing to tolerate cost, low battery life and weight, mobile phones have now become omni-present. The technology has matured so much that wireless infrastructures are more readily available than physical telephone wires in some underdeveloped countries [Rom00, Lop00]. Wireless Ethernet has become a standard feature for modern laptops, with access points becoming virtually omnipresent, such that Internet access anywhere anytime seems likely to become a reality soon.

From a sociological perspective, the rise of the Internet and widespread networking has revolutionized and democratized the distribution of information in a way unparalleled in human history. From a technological point of view, the readily available underlying networking infrastructure yields many opportunities and challenges for distributed applications.

Distributed computing has long been a subject of study in computer science, from both a theoretical and a practical perspective. Initially, distributed computing was seen as a means to parallelize computations, differing from "parallel computations" in that each machine had a separate memory, and data transfers between machines were time-consuming. Such computations usually involved only a fixed, not too large set of machines, so that every machine was aware of what every other machine was doing, allowing one to orchestrate centrally controlled computations. These restrictions were reasonable when computations were carried out at the behest of a single party responsible for both networking infrastructure and computers.

Recently however, there has been a surging interest in so-called "peer-to-peer systems", abbreviated to "P2P systems". A P2P system is a set of networked computational devices (nodes) that cooperate in the execution of distributed applications. What makes P2P systems special among distributed applications is the fact that there is *no hierarchy* among the nodes (as, e.g., there would be in a client-server system); all nodes perform similar functions and any node might contact any other node as an equal "peer". Also, the membership in a P2P system is usually *dynamic* – nodes can join and leave the system at arbitrary points of time, either voluntarily, or due to failures (of nodes or their network connections). Protocols have to ensure the continued operation of the system under such circumstances. And finally, nodes only have a *local view* of the network, so that they are, for example, unaware of the total number n of nodes in the network at any given time.

While P2P systems have existed almost as long as the Internet – the introduction of Usenet in 1979 being an example – the growing interest in them during the past few years is mostly due to file sharing applications for individual users over the Internet. Examples of such systems are Napster (`www.napster.com`), Gnutella (`www.gnutella.com`) and Kazaa (`www.kazaa.com`). In these applications, participating users can make files available on their machine to other users of the network. The systems provide a lookup mechanism that, given a filename, determines whether there is any machine in the network storing a file of that name, and if so, returns the file to the querying user. The systems are designed so that users can join and leave the network at any time.

This application of "distributed data storage", where a set of nodes cooperate in storing data items distributively, while allowing efficient lookup given a key to an item, is clearly very useful. Its implementation in the above mentioned protocols is however fraught with problems. First, there are legal problems resulting from the fact that in practice, users often share copyrighted multimedia files, such as music (MP3s) or video. More relevant for this thesis however, is the poor design of the above protocols, which illustrates the difficulties in creating good P2P systems, and is therefore worth discussing in some detail.

In Napster, the system maintains a global index of all stored files at a central location. This departs from the basic principle that there should not be any centralized control in a P2P network. More importantly, it creates a single point of failure, which is also the reason why Napster was so quickly shut down after legal challenges brought by the music industry.

Gnutella does not keep a centralized index, in fact, it keeps no index at all. So to find an item, every node in the network has to be contacted simultaneously using a flooding protocol. This creates a tremendous load on the system, in fact it has been reported [Gut00, Man02] that the system might be unusable on a low bandwidth (e.g. dialup) connection, since the down-link is instantly saturated with requests by other users.

Both these examples show the failures of "ad-hoc" approaches to the problem of distributed data storage in a P2P network. These observations have led to a significant amount of research aimed at determining what can be done with P2P systems in practice, while remaining scalable, efficient, fault-tolerant and balancing load equally. Examples are the Chord project [SMK⁺01] at MIT, Pastry [RD01] at Rice, Oceanstore [KBC⁺00] at Berkeley, Kademlia [MM02] at NYU and other projects all over the world [MNR02, KK03]. The main contributions of the these protocols are to provide a fault-tolerant routing infrastructure, and basic data storage facilities, both efficient in terms of space overhead and cost of resolving a query. We will describe these systems and other related work in more detail in Section 3.2.

Our Results

The functionality provided for distributed data storage by the above-mentioned systems is very limited. Essentially, they provide two functions: Store(Item,Key) stores an item in the system under the supplied key, and Retrieve(Key) returns the item associated with the given key (provided it has previously been stored in the network). We improve these systems in several ways. All our results are stated in terms of the Chord system, but the protocols can most likely be adapted to other P2P systems as well.

First, in Chapter 4, we give improved protocols for *load balancing* in P2P systems. In most of the P2P schemes mentioned above, items are assigned to nodes for storage. For fairness and efficiency reasons, it is desirable that the number of items stored is roughly the same for all nodes.

Chord uses a technique called "consistent hashing" to achieve load balance, which unfortunately increases the network traffic to maintain the P2P system. We improve on this protocol and provide a new load balancing scheme based on consistent hashing that reduces the traffic by a factor of $\Theta(\log n)$ (Section 4.2.2).

We then turn to load balancing schemes that move items or nodes in the system to achieve load balance (Sections 4.3 and 4.4). An application of this is an efficient data structure for searching on ordered data in a P2P system, while maintaining an even load distribution among all nodes.

In Chapter 5, we turn to a different aspect of data storage in P2P systems. For reliability reasons, and to ensure load balance, items are often stored at more than one node in the network. When "retrieving" an item, the system then returns an arbitrary copy of the item. This behavior ignores the fact that the *location* of the returned item greatly influences the time it takes to download the item. In other words, it would be desirable to obtain a copy of the item that is close in terms of network distance to the querying node.

We develop a protocol that extends Chord to allow retrieving the closest copy of an item, while increasing the lookup cost only by a constant factor. Maintaining the data structure increases Chord's maintenance costs by only a constant factor as well.

Our protocol can also be used as an off-line data structure for nearest neighbor queries in so-called "growth-restricted metrics" (for a definition see Section 5.2). These occur in other applications such as machine learning [TdSL00, Ten98].

3.2 Previous Work

As mentioned in the introduction, there has been a significant amount of research on efficient P2P protocols that support routing and basic data storage functionality. Since our results improve upon these previously developed protocols, we will take some time to explain the protocols in sufficient detail to put our results into context. As our main example, we will concentrate on the Chord protocol developed at MIT [SMK$^+$01], since we will later state our protocols so that they easily integrate with the Chord protocols. Many of our results are also applicable to other popular P2P systems, since these systems tend to share many design principles. We will therefore briefly sketch the differences and similarities between Chord and a number of other protocols.

A key component in any P2P system is the assignment scheme used to allocate the data items to be stored or computations to be carried out to the individual nodes in the network. A particular paradigm for these allocations called *distributed hash tables (DHTs)* has become the de facto standard for recent research on P2P systems [SMK$^+$01, RFH$^+$01, MM02, KK03]. In a DHT-scheme, nodes are viewed as "buckets" for a hash function. Items are hashed to buckets to assign them to nodes. Compared to conventional hash functions, the system not only allows the insertion or deletion of items, but also the insertion and deletion of buckets. As machines enter and leave the network, the assignment of items to nodes gets updated to maintain the DHT hashing scheme.

In addition to the item assignment protocol, P2P systems support a routing protocol that allows one to quickly find the node/bucket responsible for an item, while only storing little information about the rest of the network at each node. Most protocols described below store information about $O(\log n)$ other nodes at each node to perform routing, and a routing operation takes $O(\log n)$ hops to get to the destination node.

As mentioned above, we assume that no node has an accurate knowledge of the number n of nodes in the network. We will assume, however, that every node is able to estimate $\log n$ within a constant factor. This assumption seems standard among virtually all proposed P2P systems.

3.2.1 Chord

We will now give a brief description of the P2P protocol Chord [SMK$^+$01], which will form the basis of our protocols in the following chapters. The description below omits technical details that are not relevant for our discussions.

Chord employs a particular version of DHTs called *consistent hashing* [KLL$^+$97], where nodes and data items are mapped into a circular address space. In this scheme, a hash function h (in practice, SHA-1 is used) maps unique identifiers of nodes (for example their IP-address) and items (the key used to refer to the item) to the address space $[0, 1)$. The address space is cyclic, such that addresses 0 and 1 are identified. In the following, we will refer to this hash value as the address of a node or an item, respectively.

For this and the next two chapters, n is always the number of nodes present in the network, while N is the number of items stored in the network.

Routing. It is impossible in practice to have every node in the network know about every other node. There are two main reasons for this: this would require $O(n)$ storage per node, which could be too high, and – even more problematic – the network traffic caused by the insertion or deletion of nodes would be prohibitively large.

In Chord, every node therefore keeps track of only a small number of nodes, $O(\log n)$ to be precise. This clearly makes it impossible to route to a destination within a constant number of hops (we will need $O(\log n)$ hops), but solves the space and traffic problems.

In essence, Chord maintains a skip list on the nodes, inheriting the performance guarantees of that data structure ($O(\log n)$ space per node, $O(\log n)$ steps to get from a node to any other node). Note that since the address space is regarded as cyclic, the skip list "wraps around" the address space as well. By "$succ(a)$" we will refer to the node with the smallest address greater than or equal to a (taking into account the cyclic nature of the address space, i.e. if there are no nodes with addresses between a and 1, then $succ(a)$ is the node with smallest address in the address space).

More precisely, the skip list pointers are the following. A node with address a maintains pointers (also called "fingers") to the nodes $f_k = succ(a + 2^{-k})$, for all $k \geq 1$.

It is not hard to show that with high probability only $O(\log n)$ of the f_k will actually be distinct (assuming a uniform distribution of node addresses), thus storing the skip list requires only $O(\log n)$ space per node.

To create a routing path from a source to a destination, we repeatedly follow the pointer to the node f_k with the largest address that is not larger than the destination address. This also takes $O(\log n)$ hops whp.

Data Storage. Chord supports a basic version of distributed data storage. Here, data items are "pseudo-randomly" assigned to nodes that then store the items. For this, items are hashed (via their key) to the same address space inhabited by the nodes. A node stores all items whose hashed addresses are between its own address (inclusively) and the address of the preceeding node (i.e. the node with the highest address below it).

Given the key of an item, it is easy to find the node responsible for its storage. Starting at any node in the network, we simply follow pointers as if searching for a node occupying the address given by the hash value of the item's key. The routing protocol will then move us to the node with the largest address below the hash value. The successor of that node is the node responsible for storing the item.

Note that this is actually the main use of the routing data structure. Node-to-node routing is not commonly used, since a node's address is a hash of its IP-address – but if one knows a node's IP-address, it can be contacted directly, avoiding the use of the routing skip list.

Data Replication. When an item is very popular, storing it on only one node in the network would result in poor performance, because that one node would quickly get overloaded by requests. Also, storing an item at only one point leads to a single point of failure – if that node fails, access to the item is lost. In order to avoid such problems, data items can be replicated on several nodes.

This is done in Chord by replicating an item "forwards" along the address space. We call the node whose address is smallest above the address of the item the "primary" node for the item. If ℓ copies of an item are to be made, then they are stored at the ℓ nodes with addresses smallest above the item's address, i.e. at the primary node and its $\ell - 1$ successors. To achieve load balance, every query returns a random item among the set.

Implementation Issues. Inserting nodes in Chord is simple. One first finds the node preceeding the new node's address, and adds the new node to its skip list fingers. The new node's fingers f_k can be computed by following the fingers of the nodes succeeding it in the address space.

In fact, the fingers of all nodes are kept up to date by repeatedly recomputing them from the nodes following them. This will cause the "knowledge" about new nodes to percolate backwards along the address space. It also causes the skip lists to be updated after nodes have become inactive, e.g. due to crashes, network failures or voluntary departures from the network. This makes it unnecessary for Chord to have an explicit deletion mechanism (the usefulness of such a mechanism would be doubtful anyway, since one cannot rely on disappearing nodes to invoke it in the first place).

3.2.2 Other P2P Systems

In the past few years, a number of different P2P systems have been introduced in the systems and theory communities. Due to the number of different protocols and the rapid pace of new developments, a comprehensive survey is beyond this thesis, but we will briefly mention several of the better known systems that have been proposed in the past. An overwhelming amount of current research actually consists of modifications of these schemes, the work in this thesis just being one example.

Plaxton, Rajaraman and Richa [PRR99] give a distributed P2P data storage protocol. As opposed to the following schemes, it takes the *locations* of nodes into account. This means that for routing in the network and for lookup of data items, the query has to travel only a constant factor further than the direct distance between source and destination nodes. This desirable property is not achieved by any of the other protocols described here.

The routing infrastructure on a high level is similar to Chord and the other protocols described below. Every node is assigned a $O(\log n)$-bit random label, and each node maintains $O(\log n)$ pointers to other nodes that have labels with "similar" binary representations. As opposed to the other schemes mentioned here, however, the choice of pointers to other nodes also depends on the nodes' physical locations. To route to a given destination, one performs hops to addresses that are successively closer to the destination address, by having more and more bits agree between the current and the destination address.

Unlike most of the other schemes described below, Plaxton et al's protocol is not DHT-based. Items are stored at arbitrary nodes, the system does not force any particular item-to-node assignments. This models the application of sharing music files, where users usually publish the files they own, but are not willing to store other users' data. Item-lookup is performed by storing additional information in the network for each item. That means that the network traffic necessary to maintain this lookup information increases linearly with the number of items stored in the network. This is the weak point of the protocol, which makes its scalability questionable.

In Section 5 we show how one aspect of Plaxton et al's protocol, finding a nearby copy of a redundantly stored item, can be integrated into Chord with a moderate overhead that does not depend on the number of items stored in the network.

The P2P-system "Pastry" developed by Rowstron and Druschel [RD01] is similar to Chord in that nodes and items are hashed to a circular address space, and items are stored on the nodes whose addresses are close to the item's hash value. Each node has a $O(\log n)$-size routing table that contains nodes whose address representation is similar to the address of the node storing the table. For nodes close in address space, Pastry also takes their physical location into account when constructing routing tables. The result is a $O(\log n)$-hop routing scheme. The scheme can be modified for data storage to return somewhat closeby copies of items, although no provable performance guarantees are stated by the authors.

"Tapestry" [ZHS$^+$03], like Pastry, bases its routing infrastructure design on Plaxton et al's protocol. It is also not a DHT scheme, in that items are not assigned by the system to specific nodes, but can be stored at arbitrary nodes. Tapestry improves on the load-balance of Plaxton et al's protocol, but still retains the problem that the maintenance overhead for the data lookup infrastructure grows linearly with the number of items.

"CAN" [RFH$^+$01] implements a DHT-like scheme, but its addresses are not in a linear space, but points in d-dimensional space, more precisely on a d-dimensional torus. Every node maintains pointers to its immediate neighbors in address space. Routing is done greedily, by advancing a packet to the neighbor closest to the destination. This results in $O(dn^{1/d})$ hops for routing. This matches the other protocols if $d = \Theta(\log n)$, but increasing or decreasing values of n might lead to a major restructuring of the network.

As mentioned previously, in the recent past, many variations of the above schemes have been considered, improving on various aspects. For example, Kademlia [MM02] improves on Chord's fault tolerance by providing a different routing scheme, and Viceroy [MNR02] and Koorde [KK03] have constant size routing tables with $O(\log n)$-hop query resolution – yielding an optimal routing table size, but raising issues about fault tolerance.

At the time of writing, however, no universally superior protocol for P2P data storage applications has been developed. It seems like the research community is first trying to independently optimize different aspects of the P2P protocols (such as routing table sizes, reliability, load-balancing, location-aware routing, and so on), before attempting to build a protocol that incorporates the best properties from all previously developed protocols.

Chapter 4

Load-Balancing in P2P Systems

4.1 Introduction

In this chapter we consider load-balancing in P2P systems. Balancing the number of items per node in a storage application, or distributing the computational load of a distributed application fairly is an important consideration when designing a P2P system. This is in particular true since all nodes in a P2P system are considered "equal", so it would be unfair to burden some nodes more than others. Therefore, in a data storage application, we would like every node to store a number of items as close to the average (i.e. N/n) as possible.

4.1.1 Load-Balancing in Chord

We are going to state our results in terms of improving the load-balancing of the Chord protocol described in Section 3.2.1. Recall that Chord uses a scheme called "consistent hashing" to distribute items to nodes. In this scheme both items and nodes are pseudo-randomly mapped to an address space $[0, 1)$, and each node stores the items whose address falls between that node's address (inclusively) and the address of the preceeding node.

Assuming a even distribution of keys in the address space, the fraction of items stored by a node is therefore proportional to the fraction of address space between its address and the address of its predecessor. While the expectation of that fraction of address space is $1/n$, it turns out that the largest fraction served by any node is expected to be $\Theta((\log n)/n)$. This would mean that some nodes store $\Theta(\log n)$ times as many items as the average node.

In Chord this problem is resolved as described in the original work on consistent hashing [KLL$^+$97]. Each node does not choose just one address from which to serve items, but $\Theta(\log n)$ such addresses. So every real node acts like $\Theta(\log n)$ "virtual nodes". Now every node serves the union of the address intervals preceeding its $\Theta(\log n)$ addresses, which one can show to be a $\Theta(1/n)$ fraction of the total address space whp.

While simple and elegant, this solution has the disadvantage of increasing the storage for the routing information slightly (to $O(\log^2 n)$ per node), and the network traffic considerably (by a factor of $\Omega(\log n)$).

4.1.2 Overview of Results

In the following, we will see several load balancing schemes that avoid the use of virtual nodes, and still achieve load balancing guarantees that are within a constant factor of optimal. We distinguish between protocols that assume a uniform distribution of items in the address space, and protocols that do not make this assumption.

1. *Protocols assuming uniform item distribution.* In this case, good load-balancing is achieved if the fraction of address space covered by every node is $O(1/n)$. We modify consistent hashing so that every node activates only one among $O(\log n)$ possible virtual addresses. By making it unlikely that two chosen addresses are close to each other, the protocol leads to an even split of the address space. Details and analysis of the protocol are given in Section 4.2.

2. *Protocols not assuming uniform item distribution.* We give two related mechanisms that achieve load-balance in this case.

   (a) Our first protocol achieves load-balance by moving items from overloaded to underloaded nodes. This protocol can be extended to the case where the cost of storing an item is node-dependent. Unfortunately, this scheme breaks the DHT-assignment of items to nodes, so that items cannot easily be found anymore. Its application is thus limited to situations where this is not an issue, e.g. when the items correspond to programs that do not have to be found because they transmit their results automatically. Details are given in Section 4.3.

   (b) The second protocol achieves load-balance by moving nodes in the address space. It does not have the problem of destroying the DHT-assignments, so items can easily be found. In its most general form, it has two drawbacks, however. First, we lose some security, because nodes can choose their addresses arbitrarily, which allows for Byzantine (i.e. inside) attacks on the network. Second, if item keys are very unevenly distributed, this will lead to an uneven distribution of nodes as well, increasing the routing times in Chord (which were only $O(\log n)$ assuming an even distribution of nodes in the address space). Details on the protocol are in Section 4.4.

   In that section, we also state an application of the protocol: the ability to support efficient searches on ordered data by assigning keys according to the order of the data items. In this setting, it is also possible to avoid the two previously mentioned drawbacks of this load balancing protocol.

We conclude the chapter with a list of some open problems in Section 4.5.

4.1.3 Related Work

While much research has been done on routing in P2P networks, work on efficient load balancing and complex queries in P2P networks is only in its beginning stages. Most structured P2P systems simply assume that items are uniformly distributed on nodes. As mentioned above, Chord [SMK$^+$01] achieves load balances within a constant factor from optimum by employing consistent hashing [KLL$^+$97] and using $O(\log n)$ virtual nodes per real network node, incurring a corresponding overhead in routing.

Two protocols that achieve near uniform partitioning of the address space without the use of virtual nodes have recently been given in [AHKV03, NW03]. Our first scheme, given in Section 4.2, improves upon them in two respects. First, in those protocols the address assigned to a node depends on the rest of the network, i.e. the address is *not* selected from a list of possible addresses that only depend on the node itself. This makes the protocols more vulnerable to malicious attacks. Second, in those protocols the address assignments depend on the construction history, making them harder to analyze, and in fact load-balancing guarantees are only shown for the "insertions only" case.

Work on load balancing by moving items was done by Rao et al [RLS$^+$03]. Their algorithm is very similar to the one given in Section 4.3, however it only works for the static case, where items and nodes are not added or removed from the network. They also give no provable performance guarantees, only experimental evaluations.

Complex queries such as range searches are another emerging research topic for P2P systems [HHH$^+$02]. An efficient range search data structure for ordered data was recently given by Aspnes and Shah [AS03]. However, they ignore load balancing as an issue, and make the simplifying assumption that each node stores only one item. In this setting, the lookup times are $O(\log N)$ in terms of the number of items N, and not in terms of the number of nodes n. Also, $O(\log N)$ storage is used per data item, meaning a total storage of $O(N \log N)$, which is typically much worse than $O(N + n \log n)$.

4.2 Improving Consistent Hashing

4.2.1 A Note on Security

In this section, we will describe a protocol that improves consistent hashing in that every node occupies just one address, but whp is still only responsible for a $O(1/n)$ fraction of the address space.

In the original Chord protocol, each node i operates "virtual nodes" at addresses $h(i,1), h(i,2), \ldots, h(i, c\log n)$, where h is some fixed pseudo-random hash-function, and $c > 2$ some fixed constant. (In Chord, i is usually the IP-address of a node, but it can be any unique identifier.)

From a security point of view it is beneficial that a node cannot freely choose its addresses, by making them hashes of the node's IP-address. Even if a node has some control over its IP-address, the fact that the hash-function is cryptographically one-way makes it hard to determine an IP-address mapping close to a given address – at least if it is an IPv6-address. If a node were entirely free in choosing its position in the Chord address space, then it could for example prevent some item from being found in the network. This is done by simply choosing the node's address to be the address of the item – the node then becomes responsible for storing the item, and can simply refuse to do so. Such insider attacks on a network are also called Byzantine attacks.

We want to maintain this beneficial property of a node not being totally free in choosing its address. Thus we keep the set of virtual positions $h(i,j)$ from Chord. But in our protocol, each node operates only one virtual node $h(i,j)$ at any given time (we call this the "active" virtual node). However, which virtual node is active for a given node might change over time due to the insertions and deletions of other nodes. In fact, each insertion or deletion causes $O(\log n)$ other nodes to change their addresses.

4.2.2 A Simple Protocol

Before we state our actual protocol, we will consider a more intuitive, easier to analyze protocol. It is impractical, however, because it relies on the knowledge of n.

For our protocol we fix an infinite list $a_1, a_2, \ldots$ of addresses $a_k \in [0,1)$. These addresses will have the property that the first $n/2$ of them will be roughly equally spaced in the address space $[0, 1)$. We will then activate virtual nodes such that for each $k = 1, \ldots, n/2$ an active node will be close to address a_k.

This leads to load balancing: since the a_k partition the address space almost uniformly, and there is an active node near each of them, the load of no node is more than $O(1/n)$.

To be more specific, suppose we set $a_k := k \cdot \frac{2}{n}$ for $1 \leq k \leq n/2$. The following is our load balancing protocol.

Local Improvement: Each node i occasionally considers for each of its virtual positions $h(i, j)$ the address a_{k_j} with minimal index k_j contained in the interval between $h(i, j)$ and the active node preceeding it in the address space.

Node i then makes active the position $h(i, j)$ for which k_j is minimal. In case of a tie, it makes active the position closest to this a_{k_j}.

In Section 4.2.4, we will describe a way to make this protocol more efficient by invoking it only for nodes that are likely to change their addresses according to the protocol.

For our analysis, it will be useful to consider a centralized version of the protocol. We will show below (Lemma 4.5) that both protocols converge to the same final state.

Global Assignment: Start with all nodes having no active virtual node. Then for $k = 1, 2, \ldots$ do the following. Consider all virtual nodes between a_k and the succeeding active node. If this set is not empty, make active the virtual node closest to a_k, and drop from consideration all other virtual nodes that belong to that node.

As mentioned above, for load balancing purposes, it suffices to show that there will be a node active close to each a_k for $1 \leq k \leq n/2$. This is shown in the following Lemma, which actually holds for any set of addresses a_k, not just our specific choice.

Lemma 4.1
Suppose $c > 2$, and $||a_k - a_{k'}|| \geq 1/n$ for all $1 \leq k < k' \leq n/2$. Then the above protocols will activate a node within distance $1/n$ above a_k for all $k = 1, \ldots, n/2$. The result holds even if the virtual addresses are chosen only with $\Omega(\log n)$-wise independence.

Proof: The proof idea is simple. Consider the intervals $I_k := [a_k, a_k + 1/n]$ for $k = 1, 2, \ldots, n/2$. By assumption, all these intervals are disjoint. Now consider the global assignment scheme. We will argue that when dealing with address a_k, the scheme will activate a node in the interval I_k.

For this we have to show two things. First, we have to show that no node in I_k is active yet. This simply follows by induction since the I_k are disjoint. It remains to show that I_k at that point contains virtual nodes not belonging to active nodes.

This is not hard to see: by a Chernoff bound whp each of the intervals I_k contains $(c - \varepsilon) \log n$ virtual nodes at the beginning of the assignment process. By the deferred decision principle, we can assume that the "node"-to-"virtual node" assignment is only decided when we activate a node. In other words, we always keep track of which virtual nodes do not belong to already activated nodes. When we activate a virtual node, we choose an additional $c \log n - 1$ virtual nodes for that node by picking them at random from the unassigned virtual nodes.

After assigning fewer than $n/2$ nodes, we therefore have randomly assigned at most half the virtual nodes. Since each interval started out with $(c - \varepsilon) \log n$ virtual nodes, it whp still contains $(\frac{1}{2}c - \varepsilon)\log n$ virtual nodes. All we need is that this number is non-zero, i.e. $\frac{1}{2}c > \varepsilon \Leftrightarrow c > 2\varepsilon$.

Just as with the original consistent hashing scheme [Lew98], our results continue to hold for $\Omega(\log n)$-wise independent choices of virtual addresses. This follows by a standard application of results in [SSS93]. In our proof, we used two Chernoff bounds, first to show that each interval $[a_k, a_k + 1/n]$ would whp receive $\Omega(\log n)$ virtual addresses, and second to show that after randomly removing half the virtual addresses, each interval would whp still contain at least one virtual address. Both can be rephrased as standard balls-and-bins experiments, to which the results in [SSS93] directly apply. □

Corollary 4.2
The above protocol with $a_k = 2k/n$ for $k = 1, 2, \ldots, n/2$, whp leads to maximum load of $3/n$ per node.

Proof: Since each pair of neighboring addresses a_k is $2/n$ apart, and there is an active node at most $1/n$ above each address, it is immediate that any pair of neighboring active nodes is at most $3/n$ apart. □

4.2.3 The General Protocol

Although the above protocol allowed for a relatively simple analysis, we would like the addresses a_k not to depend on n, because the number of nodes will generally not be known in a P2P system. This problem is overcome by a simple trick. We choose $a_1, a_2, \ldots$ to be a sequence of addresses that gives a successively finer partition of the address space.

We let the a_k be the sequence of addresses with finite binary expansion, ordered by length of the expansion, and in case of equal length, by magnitude. We will therefore speak of "shorter" and "longer" addresses to mean earlier or later in the ordering, respectively.

This definition leads to the following sequence of addresses:

$$0 = 1,\ \frac{1}{2},\ \frac{1}{4},\ \frac{3}{4},\ \frac{1}{8},\ \frac{3}{8},\ \frac{5}{8},\ \frac{7}{8},\ \frac{1}{16},\ \frac{3}{16},\ \frac{5}{16},\ \frac{7}{16},\ \frac{9}{16},\ \frac{11}{16},\ \frac{13}{16},\ \frac{15}{16},\ \frac{1}{32},\ \ldots$$

Or equivalently, but more cryptically, we have

$$a_k = \frac{2k - 2^{\lceil \log k \rceil} - 1}{2^{\lceil \log k \rceil}}.$$

As with Corollary 4.2, it is not hard to show that the global assignment yields a good load balance.

Lemma 4.3
For the above choice of the a_k, each node has a maximum load of $5/n$.

Proof: Among the first $n/2$ addresses, neighboring addresses are at least $1/n$, and at most $4/n$ apart. Thus, by applying Lemma 4.1, we obtain that each node has load at most $5/n$. □

Equivalence of Local Improvement and Global Assignment

So far we have only analyzed the global assignment scheme (which one would not use in practice), having claimed that it yields the same assignment as the local improvement scheme (which one *can* use in practice). We will now prove this equivalence.

Definition 4.4 (Node assignment)
A node assignment f *is a function mapping each node* i *to its active node* $h(i, j)$. $\square$

Lemma 4.5
The global assignment g *is the unique fixed point of the local improvement strategy.*

Proof: We will show that

(i) the global assignment g is stable under the local improvement strategy,

(ii) in any node assignment not equal to the global assignment, there is a possible local improvement, and

(iii) any local improvement decreases a node assignment's "distance" to the global assignment.

The combination of these claims implies the Lemma.

In the following, we will speak of a node being "assigned" to an address a_k by the global assignment g. By this we mean that when the global assignment scheme was considering address a_k, it activated that particular node in a location close to a_k.

Claim (i) can easily be shown by induction. By induction over k, we show that no node would want to move from its current position under g to serve a_k, except for the node already assigned to a_k. This is true for $k = 1$, because there are no virtual nodes closer to a_1 than the node already assigned to a_1. For any higher k, by definition of g, the only virtual nodes between a_k and the next active node belong to nodes already assigned to $a_{k'}$'s with $k' < k$. Thus, none of these nodes would want to move, either.

To show claims (ii) and (iii), we first need two definitions. For any assignment f and address a_k, let $F(k)$ either be equal to the active node serving a_k, if that node serves no shorter address, or $\emptyset$ otherwise. We call $F(k)$ – unless it is $\emptyset$ – the node serving the address a_k. Let $\mathcal{F}(k)$ be the distance between a_k and address of $F(k)$, if $F(k) \neq \emptyset$, and ∞ otherwise. Let $\mathcal{F}$ be the vector in $(\mathbb{R} \cup \{\infty\})^\infty$, whose components are the $\mathcal{F}(k)$.

For claim (iii), note that any local improvement will change the vector $\mathcal{F}$. If we consider the natural lexicographic ordering on the vectors $\mathcal{F}$, then every improvement is easily seen to reduce the vector $\mathcal{F}$ in this ordering. The vector $\mathcal{G}$ associated with the global assignment is also easily seen to be the minimal possible vector resulting from any assignment: one can view the global assignment as first minimizing $\mathcal{G}(1)$ (by activating the virtual node closest to a_1), then $\mathcal{G}(2)$, and so on.

In this sense, any local improvement brings an assignment f closer to g. Since there is only a finite number of assignments, this process cannot continue forever, and must eventually reach a local minimum.

We now show claim (ii), i.e. that the global assignment is the only local minimum in the local improvement process. Consider any assignment $f \neq g$. This implies $\mathcal{F} \neq \mathcal{G}$ (assuming that all virtual nodes are distinct, f can be reconstructed from $\mathcal{F}$). Consider the minimal k for which $\mathcal{F}(k) \neq \mathcal{G}(k)$. Since $\mathcal{G}$ is minimal, we must have $\mathcal{G}(k) < \mathcal{F}(k)$, and thus $\mathcal{G}(k) \neq \infty$. So there is a node i serving a_k in g.

Since f and g agree for all addresses $a_{k'}$ for $k' < k$, node i cannot be serving any of these addresses in f. Thus, the local improvement moving node i to the position occupied in g is possible for f. This shows the claim. $\square$

4.2.4 An Improved Implementation

So far, we have seen that our variant of consistent hashing leads to an almost uniform split of the address space when the system reaches a stable state. It remains to see how quickly that stable state is reached, and how often nodes should invoke the local improvement protocol.

Let us first make an observation on how many nodes have to change their addresses based on the insertion or deletion of a node (for the proof see Section 4.2.5).

Lemma 4.6
Suppose starting at the stable state, we insert or delete a node. Then whp $O(\log n)$ nodes have to change their addresses to return the system to the stable state. □

This has an immediate unhappy corollary, because each of the $O(\log n)$ moving nodes stores $O(1/n)$ of the items.

Corollary 4.7
The insertion or deletion of a node causes $O(\log n/n)$ items to be reassigned between nodes. □

However, this might not be as bad as one might imagine. Since each node has only $O(\log n)$ possible positions, its stored items are only from one of $O(\log n)$ segments of the address space. If we do not necessarily delete stored items when moving to a new address, we already store the required items should we ever move back to the old address.

Triggering Local Improvement Updates

We have seen that every time a node gets inserted or deleted, $O(\log n)$ nodes have to change their addresses, and therefore run the local improvement protocol. Of course, if we let all nodes invoke the protocol every once in a while, eventually the "right" nodes would notice that they have to change their address, but this could take a considerable amount of time. The protocol can be made more efficient by observing that during an insertion (and, by storing some additional information, during a deletion), we can easily determine which nodes have to change their addresses. We make two observations about this.

1. When a node i moves to a new position covering some address a_k, it might "displace" another node j that was previously covering address a_k. This might cause j to move to a new address.

   This suggests the following improvement to our protocol: Whenever a node i moves to a new address, the local improvement protocol gets invokes for i's new successor.

2. When a node i moves from a position covering some address a_k with $k \leq n/2$, then some node j might have to move to cover a_k. By Lemma 4.1, the position that node j will occupy is whp among the first $(c+\varepsilon)\log n$ virtual positions after a_k.

   This suggests the following improvement: Each node stores the $(c+\varepsilon)\log n$ first virtual positions following the address a_k it is covering (let us call this list the *contenders* for address a_k). When the node gets deleted or changes its active address, it calls the local improvement protocol on these $\Theta(\log n)$ contenders. (To guard against node failures, we might have to replicate the information about contenders at several nodes.)

To quantify how often the local improvement protocol has to be executed, we need the notion of the "half-life" of a system. Intuitively, the half-life gives a measure on how quickly a P2P system changes.

Definition 4.8 (Half-life)
The half-life *of a P2P system is the time it takes for half of the nodes in the system to change (by insertions or deletions).* □

Lemma 4.9
The following is true for our protocol with the two improvements mentioned above.

(i) Maintaining for every address a_k with $k \leq n/2$ the first $(c+\varepsilon)\log n$ virtual positions following a_k can be done whp by running the local improvement once per node per half-life of the system.

(ii) If in addition local improvements are triggered by insertions and deletions (in the case of failures, before a super-constant number of successive nodes fails), then the load for all nodes is $O(1/n)$ at all times, assuming we start in a load-balanced state.

(iii) An insertion causes $O(\log n)$ local improvement invocations, and a deletion causes $O(\log^2 n)$ local improvement invocations. □

Note that the above Lemma does not guarantee that the system remains in the stable state, but only that the $n/2$ first addresses have nodes assigned to them. As we saw in the proof of Lemma 4.3, this suffices for load-balance.

We now discuss the algorithmic part of the above Lemma (claim (i)), and defer the remaining proofs to the next section. Let us first see that the list of contenders is easy to maintain during insertions.

Whenever we invoke the local improvement protocol for a node i, we consider $c\log n$ virtual positions. For the active node preceeding each of these virtual positions, we insert node i into its list of contenders if necessary. And when a node moves to cover a new address a_k, it "inherits" the list of contenders from the node previously covering a_k (if there was such a node), and initializes an empty list of contenders otherwise (this whp only happens if $k > n/2$, so it is not problematic that we do not have an accurate list of contenders). All of this increases the cost of a local improvement update only by a constant factor.

While this approach suffices to maintain the contenders during insertions, we run into a problem with deletions. In that case, the list of contenders might become smaller than $(c+\varepsilon)\log n$, because some contenders get deleted. There is a simple fix for this problem. Suppose we store slightly more than twice as many contenders as necessary (say, $(2c+3\varepsilon)\log n$ nodes). Then even after removing $n/2$ nodes from the system, whp we still always have the $(c+\varepsilon)\log n$ first contenders for each address. By that time (i.e. by the time a half-life elapses), we would have executed the local improvement protocol for all nodes, and thus again obtained a list of the $(2c+3\varepsilon)\log n$ first contenders for every address.

4.2.5 Analysis

In the following, we prove the Lemmas stated in the previous section.

Proof (Lemma 4.6): We are going to note the changes necessary in the global assignment scheme. It suffices to analyze the deletion of a node, since the scheme is Markovian and therefore insertions and deletions are symmetric.

Consider what happens when a node assigned to some address a_{k_0} gets deleted from the network. Then some node previously assigned to an address a_{k_1} ($k_1 > k_0$) gets assigned to a_{k_0}, causing some node previously assigned to a_{k_2} ($k_2 > k_1$) to move to a_{k_1}, and so on. This results in a linear sequence of changes, until finally no node moves to a vacated address. Since nodes only move to addresses with lower indices, the number of movements is clearly finite. We will now show that this number is $O(\log n)$ whp.

Let n_i be the number of nodes assigned to addresses a_k with $k > k_i$. Note that the node moving to address a_{k_i} is uniformly at random among these n_i nodes, thus $n_{i+1} \leq n_i/2$ with probability at least $1/2$. As these "decrease by $1/2$" events are independent, whp $O(\log n)$ movements suffice to reduce n_i to zero, showing that $O(\log n)$ nodes have to change their addresses upon the deletion or insertion of a node. $\square$

Proof (Lemma 4.9): Claim (i) is immediate from the discussion following the statement of Lemma 4.9 in the previous section.

For claim (ii), it suffices to observe that when restricting attention to addresses a_k with $k \leq n/2$, then the triggered local improvements will always contact the node assigned to these addresses under the stable assignment. This is because this node will always be among the $(c + \varepsilon)\log n$ first contenders for the address. So for an insertion, all nodes that are displaced will be notified, and for an deletion any node moving to the vacated address will be notified.

Claim (iii) follows from Lemma 4.6: there are $O(\log n)$ total movements. If they are due to an insertion, and each moved node causes at most one other node (its new successor) to be displaced, so $O(\log n)$ local improvements are triggered in sequence. For deletions, whenever an address is vacated, the $(c + \varepsilon)\log n$ contenders are triggered, one of which might move, triggering an additional $O(\log n)$ nodes, and so on, for a total of $O(\log^2 n)$ triggers. $\square$

4.3 Load-balancing: Moving Items

Until now, we have concentrated on balancing the fraction of address space served by each node. As long as the items are (almost) uniformly distributed in the address space, this amounts to *indirectly* balancing the number of items stored at the nodes.

In this and the next section, we give protocols that *directly* balance the load among nodes by actually moving items between nodes. The advantage of this is that we can give load balancing guarantees even if the items are not uniformly distributed in the address space. The disadvantage is that we have to modify Chord's assignment of items to nodes to achieve this. That is, we either have to allow items or nodes to move freely within the address space.

To see why this is necessary, suppose every node is restricted to a small number of ℓ virtual locations in the address space $[0, 1)$ (for example, $\ell = c \log n$ addresses as in the previous section). Then we cannot guarantee to handle item distributions where an address interval of length p has more than a $\omega(p\ell)$ fraction of the load. This is easily shown by a counting argument: consider a set of $1/p$ non-overlapping intervals of length p. Then not all of them can contain more than $np\ell$ virtual locations, in fact one of these intervals will contain at most npk virtual locations. Thus, if we limit each node's load to $O(1/n)$, we can only handle a load of $O(p\ell)$ in that interval.

In this section, we will consider the case where items can be moved to arbitrary nodes. Obviously this makes it impossible to use Chord's lookup mechanism to locate an item.

This can be fixed by maintaining for every item a pointer at the primary node to the actual location of the item. This additional level of indirection is, while only changing the cost of operations by a factor of two, cumbersome in practice, and makes the system more error-prone.

The load balancing scheme is useful, however, for P2P applications where there is no need to "find" an item, for example if the items correspond to computational tasks, and we simply want to spread the computational load equally among the nodes. This problem will be solved by the protocol given in this section.

Another instance where destroying Chord's item-to-node mapping is not harmful is when one maintains enough structure in the load-balanced data to find items without resorting to Chord's built-in item location mechanism. We will show an example of this in the following section when dealing with ordered data.

4.3.1 The Protocol

The protocol used for item-based load balancing is simple. Every node occasionally contacts a random other node, and if one of the two nodes has much larger load than the other, then items are moved from the node with more load to the node with less load.

In the following, we will denote

- by a_i the number of items stored at node i,
- by c_i the cost of storing an item at node i (so a smaller c_i could mean that the node has a higher storage capacity) , and
- let $\ell_i := c_i \cdot a_i$ be the weighted load on node i.

Ideally, we would like to have $\ell_i = \ell_j$ for all nodes i,j, i.e. the load should be balanced equally among all nodes. Note that if $N = \sum a_i$ items are stored in a network, and $L = \ell_i$ holds for all nodes i, then we have

$$L = a_i c_i \implies a_i = L \cdot \frac{1}{c_i} \implies N = L \cdot \sum \frac{1}{c_i} \implies L = \frac{N}{\sum \frac{1}{c_i}}.$$

We will use $L := N / \sum \frac{1}{c_i}$ in the remainder of this and the next section.

If it is executed frequently enough, the following simple protocol will maintain $\ell_i = O(L)$ for all nodes i at all times with high probability. Here, $\varepsilon \in [0, \frac{1}{2})$ is some constant.

Load exchange: Every node i occasionally contacts another node j at random. If $\ell_i < \varepsilon \ell_j$ or $\ell_j < \varepsilon \ell_i$, then the node with higher load transfers items to the other node until their loads are equal.

Note that this protocol has the useful feature that we never actually have to determine L, the protocol relies only on local information.

As with most P2P maintenance protocols, the quality of the load-balancing achieved depends on the rate at which the protocol is run, i.e. it depends on what exactly "occasionally" means in the description of the protocol. We will make explicit this relationship between update frequency (or protocol overhead, depending on your point of view) and the resulting load-balancing guarantees.

We will also analyze the number of items that have to be moved in order to achieve load balance. Naturally, we would like this number to be as small as possible. There are simple lower bounds. Whenever an item is inserted or deleted, that item itself is "moved", so $\Omega(1)$ is a lower bound on the total number of items moved to achieve load balance. Upon insertion or deletion of a node, on average N/n items have to be moved to or from that node to achieve load balance, thus $\Omega(N/n)$ is a lower bound in that case. We will show that the cost of the above protocol stays within a constant factor of these lower bounds.

4.3.2 Analysis: The Unweighted Case

We will first analyze the load exchange protocol for the unweighted case where $c_i = 1$ for all nodes i. In this case, the average load per node is $L = N/n$.

For a simpler analysis, we will first assume that nodes j are chosen *uniformly* at random in the protocol. Later, we will discuss the difficulties with doing this in practice, and show how the protocol can be modified to work in an actual system, and how our analysis has to be changed in that case.

Update Rate

Let us first consider the update rate, i.e. how often nodes have to contact random other nodes in order to keep their load within a constant factor of the target load $L = N/n$. Note that by the Markov's inequality, half the nodes have load at most $2L$. So if a node with load $\geq \frac{2}{\varepsilon}L$ contacts a random node, then with probability $1/2$ it will enter a load exchange. If the update rate is sufficiently high, we will therefore expect the load not to increase too much beyond $\frac{2}{\varepsilon}L$. This is expressed by the following Lemma, which relates the update rate to the "item-half-life" of the system.

Definition 4.10 (Item-Half-Life)
The item-half-life *of a P2P system is the time it takes for half of the nodes or half the items in the system to change (by insertions or deletions).* $\square$

Lemma 4.11
The load exchange protocol limits the load of all nodes to at most $\frac{16}{\varepsilon}L$ if each node performs $\Omega(\log n)$ executions of the protocol, while each of the following events occur:

- *(i) the number of items stored at the node itself doubles, or*
- *(ii) the number of items stored in the whole system halves, or*
- *(iii) the number of nodes in the system doubles.*

In other words, there have to be $\Omega(\log n)$ executions per node per item-half-life of the system, or before the node's load doubles.

Proof: Consider a node with load ℓ. Suppose it enters a load exchange with a node of load $\ell' \leq \varepsilon\ell$ or less, then the node's load reduces by a factor of at least

$$\frac{\ell}{\frac{1}{2}(\ell + \ell')} \geq \frac{\ell}{\frac{1}{2}(\ell + \varepsilon\ell)} = \frac{2}{1+\varepsilon} =: \beta.$$

To keep the load of all nodes below $\frac{16}{\varepsilon}L$, it suffices if every node whose load grows over $\frac{2}{\varepsilon}L$ has $\log_\beta \frac{16/\varepsilon}{2/\varepsilon} = \log_\beta 8 = \Theta(1)$ successful load exchanges before its load increases to $\frac{16}{\varepsilon}L$. As mentioned above, for nodes with load at least $\frac{2}{\varepsilon}L$, $\Theta(\log n)$ invocations of the protocol will whp lead to a load exchange. It therefore suffices to invoke the protocol $O(\log n)$ times before the load increases from $\frac{2}{\varepsilon}L$ to $\frac{16}{\varepsilon}L$.

What can cause the load of a node to increase? First, the number of items stored at the node can increase. Second, the value of $L = N/n$ can drop globally, either by a decrease of the number of items in the system, or an increase of the number of nodes in the system. The effective change in relative load of a node is the product of these three effects. Thus, for the relative node to change by a factor of 8 from $\frac{2}{\varepsilon}L$ to $\frac{16}{\varepsilon}L$, at least one of the values has to change by a factor of 2. So the update rate stated in the Lemma is sufficient to maintain the claimed load-balance. □

The traffic caused by the update queries of our protocol (to contact random nodes, ignoring the movement of items) is sufficiently small such that it can be carried out within the maintenance traffic necessary to keep the P2P network alive. Chord uses $\Omega(\log n)$ messages per node per half-life to maintain its routing infrastructure. If we assume that $N = \Omega(n \log n)$ (as is not too unreasonable in practice), then the above Lemma states that on average per item added or deleted in the system, we have to perform $O(1)$ executions of the protocol. Since the messages used to contact random nodes are small, they will tend to cause less traffic that the movement of the item itself. If $N = o(n \log n)$, then the traffic overhead of our protocol becomes more significant. In that case, however, perfect load-balancing might not be so important, since there are not that many items in the system.

Lower-bounding the load. In some circumstances, it might be desirable that every node also stores $\Omega(L)$ items, i.e. the load is also *at least* a constant fraction of the average load. Our protocol also achieves this goal. Note that given a sufficient update rate whp the protocol limits the maximal load to $\frac{16}{\varepsilon}L$. This implies that a constant fraction of all nodes has load at least $L/2$: let α be the fraction of nodes with load $L/2$ or more, then we must have $\alpha\left(\frac{16}{\varepsilon} - \frac{1}{2}\right) \geq \frac{1}{2} \implies \alpha \geq 1/\left(\frac{32}{\varepsilon} - 1\right)$. Thus, when contacting $\Omega(\log n)$ nodes at random, whp at least one of them will have load $L/2$ or more.

So if we make that many contacts with random nodes while a node's load drops from $\frac{\varepsilon}{4}L$ to $\frac{\varepsilon}{8}L$, then we can guarantee that the load of a node remains above $\frac{\varepsilon}{8}L$ whp. Upon inserting a node, we also have to make sure that it receives some items, for example by contacting a random node, and taking half of that node's items.

Update Cost

After showing the correctness of our protocol, it is now time to analyze its cost, in terms of the number of items that have be moved to achieve load balance. We are going to analyze this cost in an amortized sense. It turns out that the number of items to be moved is optimal within constants, if all loads stay within $\Theta(L)$.

Lemma 4.12
If all nodes have load $\Theta(L)$, then the amortized number of items moved by the load exchange protocol are

- $O(1)$ *per item insertion or deletion,*
- $O(L)$ *per node insertion or deletion.*

For arbitrary loads, the amortized number of items moved by the load exchange protocol are

- *$O(|\log \ell - \log L| + 1)$ per item insertion or deletion to or from a node of load ℓ,*
- *$O(L)$ per insertion of a node, and*
- *$O(\ell + \ell' - L)$ for the deletion of a node with load ℓ, whose load gets transferred to a node of load ℓ'.*

Proof: We are going to analyze the amortized cost using a potential function. We want to choose the potential function such that the change in potential function "pays" for all load balancing operations. To achieve this, an item has to pay upon insertion for all the load balancing exchanges that it will later be part of. In the proof of Lemma 4.11, we saw that the load of the higher loaded node reduces by a factor of $\beta = 2/(1+\varepsilon)$ or more in every load exchange. Given that the load reduces by a constant factor in every load exchange, an item on a node with load ℓ should be involved in roughly $O(\log \frac{\ell}{L})$ load exchanges until its load gets reduced to L. This motivates us to set the potential of a node with ℓ items to $\ell \log \frac{\ell}{L} = \ell \log \ell - \ell \log L$. The global potential function therefore is

$$\Phi(\bar{\ell}) := \delta \left(\sum_{i=1}^{n} \ell_i \log \ell_i - N \log L \right),$$

where δ is a sufficiently large constant, e.g. $\delta = 5$. Recall that ℓ_i is the load of node i, i.e. the number of items stored at the node (since we are considering the unweighted case). Our potential function is related to the entropy of the item distribution. More precisely, up to additive terms independent of the item distribution, the potential function is exactly the negative of the entropy. Thus, our function gets minimized when all nodes have the same load.

The amortized cost of an operation (insertion, deletion, or load exchange) will be its actual cost plus the resulting change in potential function, i.e. "amortized cost" = "actual cost" + $\Phi_{\text{after}} - \Phi_{\text{before}}$.

Item insertion: The actual cost of inserting an item is 1, since the affected item has to be handled. So the amortized cost of inserting an item at a node j is

$$\begin{aligned}
&1 + \delta \left(\sum_{i \neq j} \ell_i \log \ell_i + (\ell_j + 1) \log(\ell_j + 1) - (N+1) \log \frac{N+1}{n} - \sum_i \ell_i \log \ell_i + N \log \frac{N}{n} \right) \\
&= 1 + \delta \left((\ell_j + 1) \log(\ell_j + 1) - \ell_j \log \ell_j + N \log \frac{N}{n} - (N+1) \log \frac{N+1}{n} \right) \\
&= 1 + \delta \left(\log(\ell_j + 1) + \log \left(\frac{\ell_j + 1}{\ell_j} \right)^{\ell_j} - \log \frac{N}{n} + \log \left(\frac{N/n}{(N+1)/n} \right)^{N+1} \right) \\
&= 1 + \delta \left(\log(\ell_j + 1) + \log \left(1 + \frac{1}{\ell_j} \right)^{\ell_j} - \log \frac{N}{n} + \log \left(1 - \frac{1}{N+1} \right)^{N+1} \right) \\
&= 1 + \delta \left(\log(\ell_j + 1) - \log L + \Theta(1) \right) \\
&= \Theta \left(1 + \log(\ell_j + 1) - \log L \right).
\end{aligned}$$

If $\ell_j = O(L)$, the cost reduces to $O(1)$.

Item deletion: The actual cost of a deletion is 0, since no item has to be moved. The change in potential is the negative of an item insertion, and thus $\Theta(1-\log \ell_j + \log L)$. This cost is $O(1)$, if $\ell_j = \Omega(L)$.

Node insertion: The actual cost of adding a new node is zero, as no items are moved. The change in potential function (and therefore the amortized cost) is

$$-\delta N \log \frac{N}{n+1} + \delta N \log \frac{N}{n} = \delta \log \left(\frac{n+1}{n}\right)^N = \delta \log \left(1+\frac{1}{n}\right)^{Ln} = \Theta(\log e^L) = \Theta(L).$$

Node deletion: When deleting node j, and moving its items to node k, we incur an actual cost of ℓ_j items. The amortized cost of deleting a node therefore is:

$$\begin{aligned}
&\ell_j + \delta \left((\ell_j+\ell_k)\log(\ell_j+\ell_k) - \ell_j \log \ell_j - \ell_k \log \ell_k - \log \left(1+\frac{1}{n}\right)^{Ln}\right) \\
&= \ell_j + \delta \log \left(\left(1+\frac{\ell_k}{\ell_j}\right)^{\ell_j} \left(1+\frac{\ell_j}{\ell_k}\right)^{\ell_k} \left(1+\frac{1}{n}\right)^{-Ln}\right) \\
&= \ell_j + \Theta\left(\log\left(e^{\ell_k} e^{\ell_j} e^{-L}\right)\right) = \Theta(\ell_j + \ell_k - L).
\end{aligned}$$

If $\ell_j, \ell_k = O(L)$, the amortized cost is $O(L)$.

Load balancing operation: When moving items from node j to node k, the initial loads on those nodes are ℓ_j and ℓ_k, while both nodes will end up with a load of $(\ell_j+\ell_k)/2$. Thus, $(\ell_j - \ell_k)/2$ items have to be moved, which is the *actual* cost of the load exchange. The amortized cost therefore comes out to

$$\frac{\ell_j - \ell_k}{2} + \delta \left(2\frac{\ell_j+\ell_k}{2} \log \frac{\ell_j+\ell_k}{2} - \ell_j \log \ell_j - \ell_k \log \ell_k\right).$$

We have to show that this quantity is at most 0. For notational simplicity, let $\eta := \frac{\ell_k}{\ell_j} \le \varepsilon$. Then we have the cost

$$\begin{aligned}
&\ell_j \frac{1-\eta}{2} + \delta\left(\ell_j(1+\eta)\left(\log \ell_j + \log(1+\eta) - 1\right) - \ell_j \log \ell_j - \eta \ell_j \log(\eta \ell_j)\right) \\
&= \ell_j \left(\frac{1-\eta}{2} + \delta\left((1+\eta)\log \ell_j + (1+\eta)\log(1+\eta) - (1+\eta) - \log \ell_j - \eta \log \eta - \eta \log \ell_j\right)\right) \\
&= \ell_j \left(\frac{1-\eta}{2} + \delta\left((1+\eta)\log(1+\eta) - \eta \log \eta - (1+\eta)\right)\right) \\
&= \ell_j \left(\frac{1-\eta}{2} + \delta\left(\log(1+\eta) + \log\left(1+\frac{1}{\eta}\right)^{\eta} - (1+\eta)\right)\right) \\
&\le \ell_j \left(\frac{1-\varepsilon}{2} + \delta\left((\eta+0.087) + \log\left(1+\frac{1}{1/2}\right)^{1/2} - (1+\eta)\right)\right) \\
&\le \ell_j \left(0.5 - 0.12\delta\right),
\end{aligned}$$

using $\varepsilon < 1/2$. Thus, for $\delta > 4.17$, we obtain that load exchanges are paid for by the drop in potential function. $\square$

Random Choices

So far, we have assumed that the nodes involved in an update are chosen uniformly at random. However, in an actual network, choosing nodes truly at random is a very hard, if not impossible task. So how do our protocol and analysis change if do not choose nodes uniformly at random? And how do we actually implement such a "random choice" in a P2P system, for example in Chord?

In Chord, we can use the following simple scheme: pick an address uniformly at random from the address space, and choose the node that serves that address. The probability of returning a node is then equal to the fraction of address space served by the node. Does our protocol still work in this case?

We used the fact that nodes are selected at random only in the proof of Lemma 4.11. There we used the fact that in selecting a random node j, with probability at least 1/2, the load of that node would be at most $2L$. Suppose that in our new probability distribution, half the probability mass is on nodes with load at most αL. Then a straightforward change of the analysis shows that the update rate given in Lemma 4.11 guarantees a maximal load of $\alpha \cdot \frac{8}{\varepsilon} L$. Thus, in order for the protocol to still "work", we must show that with probability at least 1/2, a "randomly" picked node j satisfies $a_j \leq \alpha L$ for some constant α.

Almost uniform probability distributions. An easy way to guarantee the performance of our protocol is to use some mechanism that makes each node serve at most a $O(1/n)$ fraction of the address space, e.g. the protocol described in Section 4.2.2, or the standard virtual node scheme [KLL+97, SMK+01]. Suppose each node serves at most a β/n fraction of the address space. This implies that for a randomly picked node j, we have $E[a_j] \leq \beta L$, and thus by Markov's equality $\Pr[a_j \leq 2\beta L] \geq 1/2$, and the condition "$a_j \leq \alpha L$" is satisfied with $\alpha = 2\beta$.

Random node addresses. When the node addresses are chosen uniformly at random, a single node might serve as much as a $O(\log n/n)$ fraction of the address space. However, there is a simple way to still obtain a uniform node sample under this condition, as follows: pick a node i uniformly at random, and pick a random number $k \in \{0, 1, \ldots, m \log n - 1\}$, where $m > 1$ is some constant. Then return node $i + k$, i.e. the k-th successor of node i.

What is the probability of a node being returned by this protocol? This can be seen by a backwards analysis. How could a particular node i_0 have been selected? First, we must have selected one of its $(m \log n - 1)$ predecessors (or itself), and then with probability $1/(m \log n)$ moved to node i. If p_i is the fraction of address space served by node i, then the probability of choosing node i_0 is

$$\frac{1}{m \log n} \sum_{i=i_0 - m \log n + 1}^{i_0} p_i.$$

It can be shown that the total address space occupied by $m \log n$ consecutive nodes is $\Theta(m \log n/n)$ whp. Thus, the probability of selecting node i_0 is $\Theta(1/n)$, as desired.

Arbitrary probability distributions. We will now consider the case where there are no restrictions on the nodes' distribution in the address space, i.e. on the fractions of address space they serve. Clearly, picking a node by selecting an address uniformly at random, and returning the node that serves the address will not yield a uniform selection of nodes.

The good news is that under some circumstances, our load balancing algorithm will still work. The bad news is that the update rate required of a node does not depend on global changes in the network or local changes at the node itself, but might be influenced by local changes on other nodes. This is clearly an unsatisfactory situation, and therefore in real applications, one should try to use one of the approaches mentioned previously to obtain a near uniform selection of nodes.

Lemma 4.13
Given the update rates in Lemma 4.11, and an update rate that is high enough to guarantee $\Pr[a_i \geq 2L/\varepsilon] \leq 1/2$ *at all times, the maximal load is limited to* $\frac{16}{\varepsilon^2}L$ *at all times whp.*

Proof: This is immediate from the preceeding discussion, where $\alpha = 2/\varepsilon$. □

What is a sufficient condition for $\Pr[a_i \geq 2L/\varepsilon]$ to remain below 1/2 at all times? At least half the nodes in the system have a load of $2L$ or less, and they would enter a load exchange with any node of load $2L/\varepsilon$ or more. Thus the update rate of the "light" nodes (with load $2L$ or less) has to be high enough so that the more probable of the nodes i with $a_i \geq 2L/\varepsilon$ are contacted by them before $\Pr[a_i \geq 2L/\varepsilon]$ grows from, say 1/4 to 1/2. It suffices if every light node did $\Omega(\log n)$ update checks during that time.

Obviously, it would be preferable to give a guarantee in terms of the heavy nodes, e.g. have them perform one update per added item. But the probability of the heavy nodes might be so large that it is virtually impossible for them to contact a light node at random.

The sufficient condition in terms of the light nodes is therefore the best we can do. It is not hard to see that $\Pr[a_i \geq 2L/\varepsilon]$ can grow quite rapidly if some nodes have large probabilities. And the number of items inserted to achieve this effect also does not have to be large. The more effect a single item has on the growth of $\Pr[a_i \geq 2L/\varepsilon]$, however, the higher the probability of the node storing the item has to be, and therefore the likelier it is to be contacted by a light node.

4.3.3 Analysis: The Weighted Case

Suppose now that not all nodes are of equivalent ability, e.g. some nodes are faster, have more storage attached or a faster network connection, so it is easier for them to store data items. Or items actually represent programs, and nodes have different computation speeds. We model this by assigning every node i a value c_i, which represents the "cost" of storing a single item. So node i experiences a *weighted load* of $\ell_i = a_i c_i$. We assume that $1 \leq c_i \leq \gamma$ for all i. If γ is a constant, we will obtain essentially the same performance guarantees as in the unweighted case.

Note that balancing the load within a constant of optimum could be a trivial claim if that constant depended on γ (and γ is a constant), because weighted and unweighted loads are related within that factor. However, we are going to show a result where only the update rate and amortized item movement costs, but not the load guarantee, depend on γ. Recall that $L = N / \sum \frac{1}{c_i}$.

Lemma 4.14
Suppose the load exchange protocol is executed $\Omega\left(\gamma \cdot \frac{1}{\log(\frac{\gamma+1}{\gamma+\varepsilon})} \log n\right)$ *times per node while*

(i) the number of items stored at the node itself doubles, or

(ii) the number of items stored in the whole system halves, or

(iii) the number of nodes in the system doubles.

Then whp the weighted load of all nodes is limited to $\frac{16}{\varepsilon}L$. If γ is a constant, the amortized item movement costs are within a constant factor of the ones given in Lemma 4.12.

Proof: Let us first consider the load-balancing guarantee. When selecting a node uniformly at random, with probability $\geq 1/(\gamma+1)$ it has a weighted load of at most $2L$. To see this, note that the set of nodes S with weighted load of $2L$ or more would store at most $N/2$ items if all nodes' loads were equal to L. But that means that there can be at most γ times as many nodes in S than its complement (the nodes with load less than $2L$). This gives the desired probability.

So in γ trials, we have a constant probability of selecting a node with probability $\leq 2L$, and therefore $O(\gamma \log n)$ trials suffice to find a node with load $2L$ whp. Let us analyze what fraction of items the higher loaded node will lose in the exchange. It is not hard to see that the fewest items are lost if the higher loaded node is one with small storage cost (high capacity), and the less loaded node has high storage cost. Let us assume for simplicity that the smaller storage cost is 1, and the higher cost is γ.

If the higher loaded node stores αL items ($\alpha \geq \frac{2}{\varepsilon}$), then the other node has at most $\frac{2}{\gamma}L$ items. If x is the load of the higher loaded node after the exchange, and y is load of the other node after the exchange, then we have

$$x + y \leq \left(\alpha + \frac{2}{\gamma}\right) L \quad \wedge \quad x = \gamma \cdot y \implies x \leq \frac{\alpha\gamma + 2}{\gamma + 1}.$$

So the load of the over-loaded node gets reduced by at least a factor of

$$\frac{\alpha}{\frac{\alpha\gamma+2}{\gamma+1}} = \frac{\gamma+1}{\gamma + \frac{2}{\alpha}} \geq \frac{\gamma+1}{\gamma+\varepsilon}. \tag{4.1}$$

Following the analysis of Lemma 4.11, we therefore conclude that the stated update rate is sufficient.

Next, we have to analyze the cost of the operations involved in load balancing. For this, we will assume that $\gamma = O(1)$. Again, we perform a amortized analysis using a potential function. This time, the potential function is

$$\Phi(\overline{a}) := \delta' \left(\sum_{i=1}^{n} a_i \log(a_i c_i) - N \log L \right),$$

where again δ' is a sufficiently large constant, depending on ε and γ. The analysis of the insertion and deletion of items and nodes is very similar to the proof of Lemma 4.12. We therefore go through the calculations a bit more quickly.

Item insertion: The amortized cost of inserting an item at a node j is

$$\begin{aligned}
&1 + \delta' \left((a_j+1)\log((a_j+1)c_j) - a_j \log(a_j c_j) - (N+1)\log \frac{N+1}{\sum 1/c_i} + N \log \frac{N}{\sum 1/c_i} \right) \\
&= 1 + \delta' \left((a_j+1)\log(a_j+1) - a_j \log a_j + \log c_j - \log L + \Theta(1) \right) \\
&= 1 + \delta' \cdot \Theta(1 + \log(a_j+1) + \log c_j - \log L) \\
&= \Theta(1 + \log(a_j+1) + \log c_j - \log L).
\end{aligned}$$

Item deletion: This is exactly the opposite cost of an insertion.

Node insertion: The actual cost of adding a new node j is zero, as no items are moved. The change in potential function (and therefore the amortized cost) is

$$
\begin{aligned}
&- \delta' N \log \frac{N}{\sum 1/c_i + 1/c_j} + \delta' N \log \frac{N}{\sum 1/c_i} = \delta' \log \left(\frac{\sum 1/c_i + 1/c_j}{\sum 1/c_i} \right)^N \\
&= \delta' \log \left(1 + \frac{1/c_j}{\sum 1/c_i} \right)^{L \sum 1/c_i} = \Theta(\log e^{L/c_j}) = \Theta\left(\frac{L}{c_j} \right).
\end{aligned}
$$

Node deletion: When deleting node j, and moving its items to node k, we incur an actual cost of a_j items. The amortized cost of deleting a node therefore is:

$$
\begin{aligned}
& a_j + \delta' \left((a_j + a_k) \log((a_j + a_k)c_k) - a_j \log(a_j c_j) - a_k \log(a_k c_k) - \Theta\left(\frac{L}{c_j} \right) \right) \\
&= a_j + \delta' \left((a_j + a_k) \log(a_j + a_k) + a_j \log c_k - a_j \log a_j - a_j \log c_j - a_k \log a_k - \Theta\left(\frac{L}{c_j} \right) \right) \\
&= a_j + \delta' \left(\log \left(\left(1 + \frac{a_k}{a_j} \right)^{a_j} \left(1 + \frac{a_j}{a_k} \right)^{a_k} \right) + a_j \log \frac{c_k}{c_j} - \Theta\left(\frac{L}{c_j} \right) \right) \\
&= a_j + \delta' \cdot \Theta\left(a_j + a_k - \frac{L}{c_j} \right) \\
&= \Theta\left(a_j + a_k - \frac{L}{c_j} \right).
\end{aligned}
$$

Load balancing operation: Let m be the number of exchanged items from node j to node k. By equation (4.1), we have that $\eta a_j \leq m$ for some constant $\eta > 0$ depending on ε and γ. Recall that we have $c_j(a_j - m) = c_k(a_k + m)$, since the weighted loads are balanced after the operation.

The amortized cost therefore is

$$
\begin{aligned}
& m + \delta'((a_j - m) \log((a_j - m)c_j) + (a_k + m) \log((a_k + m)c_k) \\
&\qquad - a_j \log(a_j c_j) - a_k \log(a_k c_k)) \\
&= m + \delta' \left(a_j \log((a_j - m)c_j) + a_k \log((a_k + m)c_k) - a_j \log(a_j c_j) - a_k \log(a_k c_k) \right) \\
&= m + \delta' \left(a_j \log(a_j - m) - a_j \log a_j + a_k \log(a_k + m) - a_k \log a_k \right) \\
&= m + \delta' \log \left(\left(1 - \frac{m}{a_j} \right)^{a_j} \left(1 - \frac{m}{a_k} \right)^{a_k} \right) \\
&\leq m + \delta' \log \left(\left(1 - \frac{1}{1/\eta} \right)^{m/\eta} e^m \right) \\
&\overset{(*)}{=} m + \delta' \log \left(e^{-m(1+\tau)} e^m \right) \\
&= m - \delta' \tau \log_2 e \cdot m.
\end{aligned}
$$

In step $(*)$, the constant $\tau > 0$ depends on η. By setting $\delta' > 1/(\tau \log_2 e)$, the amortized cost becomes negative, as desired. □

4.4 Load-balancing: Moving nodes

The protocol described in the previous section moved items arbitrarily between nodes to achieve load balance. It had the disadvantage of making it hard to find items after the load-balancing.

In this section, instead of moving items, we will move nodes (i.e. change their addresses) in order to achieve load balance. The traditional Chord protocol can then be used to find items in the network. This protocol will also be useful for storing *ordered data*, as it easily allows for efficient range searches on ordered data stored in P2P systems (see Section 4.4.3).

Note that our approach is different from the improvement to consistent hashing in Section 4.2 in one important respect: there, we only split the address space evenly, ignoring the actual distribution of items; here, we split the items evenly among nodes, for *any* distribution of items.

There are three reasons why this "moving nodes" approach is not simply superior to the previously considered load balancing schemes. First, as opposed to the item movement scheme in the preceeding section, we cannot easily handle weighted load functions. Second, by allowing each node to take any position in the address space, we make it possible for malicious nodes to exploit this feature to disrupt the network, as described in Section 4.2.1. And finally, if the nodes are distributed unevenly in the address space, the routing times in Chord can grow beyond $O(\log n)$. The last two problems can be circumvented by maintaining a random skip list (much like Chord's routing infrastructure) among the nodes instead of actually changing their addresses (see Section 4.4.3). But all these points should be kept in mind when using the following load balancing protocol.

4.4.1 The Protocol

We assume that the nodes have indices i numbered according to their order in the address space. Also, we will refer to the ordering of items in the address space as the *ordering* of the items themselves. Similar to the item movement protocol described in the previous section, we will have nodes contact random other nodes. If a pair of such nodes has very uneven loads, then the load of the higher loaded node gets split among the two nodes.

In the following, ε is some constant from $[0, 1/4)$ (note the change from the load exchange protocol, where we only had the restriction $\varepsilon \in [0, 1/2)$).

Node balancing: Every node i occasionally contacts a random other node j. If $a_j \leq \varepsilon a_i$ or $a_i \leq \varepsilon a_j$, then do the following (wlog $a_j \leq \varepsilon a_i$, otherwise switch i and j):

1. If $a_{j+1} > a_i$, then $i := j + 1$.
2. Move j's address to between nodes $i-1$ and i, with the address chosen such that node j captures half of node i's items.

Because of its simple formulation, the subtleties of the protocol might not be immediately apparent. It will therefore be illustrative to consider what items get reassigned in an update.

Case 1, $i = j + 1$**:** The first $(a_i - a_j)/2$ items of node i move to node j.

Case 2, $i \neq j + 1$**:** All a_j items of node j move to node $j + 1$, and the first half of node i's items ($a_i/2$ in number) move to node j.

In the item movement protocol, we had no provision corresponding to case 1. We need it here because in case 2, all of node j's previous items get moved to its successor (node

$j+1$). So if node $j+1$ already had a significant load, we would make the load balance in the network even worse. This is why we have every node first load balance with its successor, and only then with other nodes in the network.

An Alternate View

This last description in terms of items moved between nodes also gives a way of implementing the protocol without changing nodes' addresses. Suppose we can efficiently keep track of an ordering imposed on the nodes, for example by using a random skip list. Then all node movements can be done by changing the node's position in the skip list, and by moving the appropriate set of items, as stated above.

Suppose the items have an ordering, and their keys correspond to their position in the ordering. Then the above scheme ensures that each node stores a continuous segment of the ordering. The random skip then makes it possible to retrieve items by referring to their position in the ordering, even though this protocol breaks the DHT-mapping. We will later exploit this for implementing efficient range searching procedures on ordered data.

4.4.2 Analysis

Similar to Section 4.3.2, we will first analyze what update rate is necessary to achieve load balance, and then consider the amortized cost of item or node insertions or deletions in terms of number of items moved.

Update Rate

In terms of its load balancing behavior, the node balancing protocol is remarkably similar to the protocol given in Section 4.3.1. First let us assume that every node contacts its successor (to enter a load exchange if necessary) before either ones load doubles or halves. This keeps the load of neighboring nodes within a factor of 4 of each other, so that step 1 will not be executed when contacting a random node j (recall that $\varepsilon \leq 1/4$). We can therefore ignore that case for our analysis.

Step 2 equalizes load even "better" than the load balancing protocol from Section 4.3.1. More precisely, at the end of the load exchange the higher loaded node either has half the combined load of the nodes involved in the exchange (if $i = j+1$), or half its original load (if $i \neq j+1$). The former is the guarantee given by the protocol from Section 4.3.1, the latter is even less. However, the latter case has the side effect of increasing the load of node $j+1$. That means that node $j+1$ has to perform the same number of load exchanges as if all these items had been added one by one.

Thus, the analysis from Section 4.3.2 can be applied to our present protocol, yielding the following Lemma.

Lemma 4.15
The node balancing protocol limits the load of all nodes to at most $\frac{16}{\varepsilon}L$ and at least $\frac{\varepsilon}{8}L$ if each node performs $\Omega(\log n)$ executions of the protocol while each of the following events occur

(i) the number of items stored at the node doubles or halves, or

(ii) the number of items stored in the whole system doubles or halves, or

(iii) the number of nodes in the system doubles or halves. □

Update Cost

Also with respect to the update costs, the node balancing protocol is similar to the protocol of Section 4.3.1 – the update costs are higher only by a constant factor.

Lemma 4.16
The amortized number of items moved by the node balance protocol are within a constant factor of the ones stated in Lemma 4.12 for the load exchange protocol.

Proof: We are going to use the same potential function as in the proof of Lemma 4.12, namely

$$\Phi(\overline{a}) := \delta'' \left(\sum_{i=1}^{n} \ell_i \log \ell_i - N \log L \right),$$

where δ'' is a sufficiently large constant, e.g. $\delta'' = 8$. The cost for inserting or deleting an item or node can be analyzed just as in the proof for Lemma 4.12. So it remains to show that the expected amortized cost of a load exchange is negative. For this, we will need the assumption that $\varepsilon < 1/4$.

The case $i = j + 1$, where the load is split equally among two nodes, follows the same analysis as for Lemma 4.12. We will therefore now concentrate on the case $i \neq j + 1$, which involves the three nodes i, j and $j + 1$. To simplify notation, we set $x := \ell_i$, $y := \ell_j$ and $z := \ell_{j+1}$. Recall that we have $y \leq \varepsilon x$ and $z \leq x$.

The actual number of items moved is $x/2 + y$. The three nodes' contribution to the potential function is $\delta''(x \log x + y \log y + z \log z)$ before the update, and

$$\delta'' \left(\frac{x}{2} \log \frac{x}{2} + \frac{x}{2} \log \frac{x}{2} + (y + z) \log(y + z) \right)$$

after the update. So the change in potential function is

$$\begin{aligned} \Delta\Phi &= \delta'' \left(2\frac{x}{2} \log \frac{x}{2} + (y + z) \log(y + z) - (x \log x + y \log y + z \log z) \right) \\ &= \delta'' \left(-x + (y + z) \log(y + z) - y \log y - z \log z \right). \end{aligned}$$

Note that the function $f(y, z) := (y + z) \log(y + z) - y \log y - z \log z$ is increasing in y and z for $y, z \geq 0$, since $\frac{1}{\partial y} f = \log(y + z) - \log y \geq 0$ (and likewise $\frac{1}{\partial z} f \geq 0$ by symmetry).

Thus, the cost of the load balancing operation gets maximized for $y = \varepsilon x$, and $z = x$. The maximal amortized cost therefore is

$$\begin{aligned} &\frac{x}{2} + y + \delta''(-x + f(\varepsilon x, x)) \\ &= \frac{x}{2} + y + x\delta''(-1 + (1 + \varepsilon) \log((1 + \varepsilon)x) - \varepsilon \log(\varepsilon x) - \log x) \\ &= \frac{x}{2} + y + x\delta''(-1 + (1 + \varepsilon) \log(1 + \varepsilon) - \varepsilon \log \varepsilon) \\ &= \frac{x}{2} + y + x\delta'' \left(-1 + \log(1 + \varepsilon) + \log \left(1 + \frac{1}{\varepsilon} \right)^{\varepsilon} \right) \\ &\leq x \left(\frac{3}{4} + \delta'' \left(-1 + \log \frac{5}{4} + \log \sqrt[4]{5} \right) \right) \\ &\leq x \left(0.75 - 0.0975\delta'' \right). \end{aligned}$$

This is less than 0 if $\delta'' > 7.7$. $\square$

4.4.3 Searching Ordered Data

So far, we have assumed that there is no structure on the data items themselves, and the only way to retrieve an item from the network is via its key. In this section, we are going to assume that the items have an underlying order, and we will give efficient protocols for searching on them. If S is the set of items stored in the P2P system, then our protocols will support the following queries.

Successor: Given a query q, return the minimal item $x \in S$ with $q \leq x$.

Range Search: Given items a, b with $a < b$, return the set $\{x \in S \mid a \leq x \leq b\}$.

The system will have the following performance guarantees.

Successor: Have to contact $O(\log n)$ nodes whp to resolve a query.

Range Search: Have to contact $O(\log n + K/L)$ nodes whp to resolve a query, where K is the size of the answer, and $L = N/n$.

Insert/delete item: $O(\log n)$ nodes to contact, and $O(1)$ amortized cost, in terms of number of items moved.

Insert/delete node: $O(L)$ amortized cost, in terms of number of items moved.

Load balance: Every node stores $\Theta(N/n)$ items whp.

The algorithms are quite straightforward, using the protocols described in previous sections.

Data Structure: We use Chord, and balance load with the node balancing protocol. Item keys are chosen according to the ordering of the item. In other words, item p is smaller than item q iff p's key is smaller than q's key. Insertion of items and nodes is handled as usual in Chord.

Successor Search: Find the node responsible for q's key in system. This node or its successor stores the successor of q.

Range Search: Find the node storing the successor of a, then output the elements at that node, and its succeeding nodes, until we pass b's address.

The claimed performance bounds are immediate. Finding a node responsible for a given address takes $O(\log n)$ hops in Chord. And every node stores $\Theta(L)$ items, so we have to query $\Theta(K/L)$ nodes in a range search.

As pointed out in the introduction, the $O(\log n)$ lookup time does not hold in Chord if the nodes are too unevenly distributed in the address space. It is however possible to implement this data structure without resorting to Chord's routing infrastructure. As mentioned in Section 4.4.1 ("An Alternate View"), the node balancing protocol can be used when we maintain a data structure that allows lookup of nodes in the order of the elements they are storing. If we also maintain a skip list on the nodes, then locating a node responsible for a key takes $O(\log n)$ hops, yielding the claimed bounds.

4.5 Open Problems

Load balancing remains a challenging area for research in P2P systems. In particular when storing structured data, load balancing the data while still providing efficient access to it, leads to interesting problems. We mention several open problems in the area of load balancing.

Consistent Hashing: We improved consistent hashing by requiring only one instead of $\Theta(\log n)$ virtual nodes per real node in the network. While this reduces the network traffic overhead to maintain the Chord routing infrastructure by a factor of $\Theta(\log n)$, we note that our scheme causes an increase in the number of items moved upon the insertion or deletion of nodes (Corollary 4.7). It remains open whether there is an improvement to consistent hashing that reduces the number of virtual nodes, but does not increase the overhead in terms of item transfers. It might well be that there is a tradeoff between the two quantities, and they cannot be minimized simultaneously.

Items ordered according to many criteria: A shortcoming of our range search protocols for ordered data is that they do not easily generalize to more than one order. For example when storing music files, one might want to index them by both artist and song title, requiring lookups according to two orderings. Since our protocol rearranges the items according to the ordering, doing this for two orderings at the same time seems difficult.

A simple, but inelegant, solution is to rearrange not the items themselves, but just store pointers to them on the nodes. This requires far less storage, and makes it possible to maintain two or more orderings at once.

Chapter 5

Proximity Searching in P2P Systems

5.1 Introduction

Most research on peer-to-peer systems makes the simplifying assumption that all nodes are somehow the "same". It does not matter which node performs a computation, it does not matter which node stores an item, the communication time does not depend on where nodes are located. Clearly, these assumptions are not valid in a real system. Nodes have different computational speeds, different storage capacities, and the communication between nodes close to each other in the network is usually faster. The design of practical network protocols therefore has to take into account the different properties of the network participants. Some of our load balancing protocols from the previous chapter were able to model different node capabilities by assigning node-dependent storage costs. In this chapter, we will consider the influence of the physical locations of nodes on the performance of a system.

In a real-world peer-to-peer system, the location of nodes in the network determines how quickly or easily they can communicate. It should come as no surprise that two computers on the MIT Ethernet tend to communicate with a higher bandwidth and shorter roundtrip times than a computer at MIT and a computer in China. Let us consider one example where this becomes important for distributed data storage. Suppose we have stored several copies of a data item in a P2P network, and a computer a MIT issues a request for the item. Which copy should we return, the one located on another computer at MIT, or the one on the computer in China? The answer seems obvious, yet the P2P system Chord (as described in Section 3.2.1) is equally likely to return either, since its protocols do not take into account where nodes are physically located. This suggests an important improvement to Chord.

Find closest copy: Given a node q and a key k, return the node closest to q that stores an item with key k.

By "node closest to q" we could mean for example the node with shortest roundtrip time from q, or the node to which q has the highest bandwidth. Clearly, a protocol for this task would lead to lower access times, and less bandwidth usage, since items do not have to be transferred as far.

There is a flip side to this problem, though. One reason to replicate data items is to alleviate congestion. Since Chord's replication scheme does not take network distance into account, however, copies of an item popular at MIT (say the 6.046 course homepage), might be made in China, Nepal, and Peru. If all users at MIT use a "find closest copy" protocol, we will never see an access load distribution due to the replication, since all users will continue to access the server at MIT.

Instead, it would make sense to distribute the item onto nodes "close" to the original server. A first step in accomplishing this goal would be to *find* these nodes close to the original server.

Find closest node: Given a node q, return the node closest to q (but not q itself).

Find nodes within distance r: Given a node q and a distance r, return all nodes within distance r of q.

Find K closest nodes: Given a node q and number K, return the K nodes closest to q.

There are other P2P applications where this might be useful. In sensor networks, nodes often want to exchange information with nearby nodes. So it is important to be able to find the closest node, or a set of closeby nodes.

In this chapter, we will give efficient protocols that can resolve the above queries by contacting $O(\log n + K)$ nodes, where K is the size of the output, and use only $O(\log n)$ storage per node.

Other Applications

Based on our P2P protocol, we also develop a dynamic offline data structure for nearest neighbor searches in so-called "growth-restricted metrics" (for a definition see Section 5.2). The space requirement for this data structure is $O(n \log n)$, queries can be answered in time $O(\log n + K)$, and insertions and deletions of points take $O(\log n)$ amortized time. The data structure has applications is areas such as machine learning (see Section 5.2.3)..

5.1.1 Outline of the Chapter

In Section 5.2, we first introduce and discuss "growth-restricted metrics". We then discuss where these metrics appear in practice (Sections 5.2.3-5.2.5), in particular pointing out their use in machine learning (Section 5.2.3), and finally discuss their use in P2P networks (Section 5.2.6). In Section 5.3, we describe a non-dynamic version of our protocol (i.e. we do not allow the insertion or deletion of nodes), and show how it can be used for finding nearest neighbors. In Section 5.4, we show how range searches can be performed in our protocol. We then discuss the dynamic maintenance of the data structure in Section 5.5. Implementation issues for Chord, including a simpler dynamic maintenance of the data structure are discussed in Section 5.6. An offline data structure for nearest neighbor searches in growth-restricted metrics is described in Section 5.7. We conclude the chapter with a discussion of open research problems in Section 5.8.

5.1.2 Related Work

The problems we are concerned with in this chapter are *metric proximity queries*, a well-studied problem in computer science. Formally, we have a set S of nodes, and a distance metric d on the set, yielding a metric space (S, d). In this space (S, d), we intend to efficiently answer nearest neighbor queries as well as range queries (asking for all nodes within a certain distance of a point).

Since metric proximity queries are a well-studied subject, it is natural to ask what previous research can be applied to our problem.

Most research on nearest neighbor search has focused on the case of vector spaces and/or Euclidean metrics $(\mathbb{R}^d, L_p)$ [Ben75, BWY80]. A large number of data structures have been developed that perform very well (with logarithmic time per operation) in *low dimensional* Euclidean spaces.

However, the distance metrics encountered in P2P networks are unlikely to be Euclidean, so it makes sense to consider research on general metrics. In a recent survey [CNBYM01], Chávez et al. give an overview on the data structures developed for general metrics. The most common approach is to use "pivoting" [Uhl91, Yia93], i.e. the space is partitioned into two halves by picking a random pivot, and points are put into either half of the partition according to their distance to the pivot element. Variations use multiple pivoting elements per split [Bri95]. While these structures can determine in $O(\log n)$ time whether a point is in the set S, they cannot be used efficiently to find nearest neighbors, or perform range queries unless the distances involved are very small. This is because in general the search ranges can split at every pivoting step, requiring the exploration of a substantial part of the search tree. (Comparable to the performance guarantee of nearest neighbor searches using quad-trees, which is $O(\sqrt{n})$.) Also, dynamic maintenance of the trees, in particular deletion, is difficult.

Clarkson [Cla99] developed two data structures for non-Euclidean spaces. He assumes that the samples S and q are drawn from the same (unknown) probability distribution. While his data structures apply to the growth-restricted spaces that we consider (as long as the non-trivial assumption of random inputs is satisfied), they have super-logarithmic query times, and do not allow for insertion or deletion of elements.

So to obtain good performance results for our searches, we have to exploit some properties of the metrics found in actual P2P systems. A result in this direction is the work by Plaxton et al [PRR99]. They describe a distributed system that stores replicated copies of data items, and a protocol that lets any node retrieve a "nearby" copy of data items – more precisely, their randomized scheme finds a copy whose expected distance is close to that of the nearest copy. Their scheme makes assumptions that are similar to (but more restrictive than) the ones made in our work. And while their scheme only guarantees to find an approximate closest copy of a data item (our protocol returns the closest copy), the search path taken by the query is only a constant factor longer than the shortest possible path, a property that our scheme does not satisfy without modifications.

Recently, Krauthgamer and Lee [KL03] have given deterministic nearest neighbor search data structures for a class of metrics that includes the metrics considered in this chapter. However, it is not clear whether their data structure can be distributed without causing bottlenecks in the load-distribution.

5.2 Growth-Restricted Metrics

5.2.1 Definitions

In this chapter, we will describe protocols and data structures for metric proximity queries in "growth-restricted point-sets". They are defined as follows.

Definition 5.1 ($B_S(p,r)$)
Let (M,d) be a metric, and $S \subseteq M$ be a point-set. For any $r > 0$ and $p \in M$ we set $B_S(p,r) := \{s \in S \mid d(p,s) \leq r\}$. If S is clear from context, we simply write $B(p,r)$. □

Definition 5.2 (Growth-restricted point-set)
*Let (M,d) be a metric, and $S \subseteq M$ be a finite point-set. We call S (c,ρ)-*growth-restricted *(for $c > 1$) iff for all $p \in M$ and $r > 0$ we have*

$$|B_S(p,r)| \geq \rho \implies |B_S(p,2r)| \leq c \cdot |B_S(p,r)|. \tag{5.1}$$

*We call S c-*growth-restricted *iff S is $(c, \Theta(\log |S|))$-growth-restricted.* □

Note that the multiple "2" for the radius in inequality (5.1) was arbitrary, changing it to some other constant will simply make point-sets growth-restricted for different values of c.

In the remainder of this section, we will first prove some basic properties about growth-restricted metrics, then discuss some examples where growth-restricted metrics occur in practice, and conclude in Section 5.2.6 by discussing their relevance to P2P networks.

5.2.2 Basic Properties

In general, "growth-restricted" means that the points of S "come into view" at a limited rate when increasing the radius of a ball around any point $p \in M$. In particular, if $|B_S(p,r_0)| = \rho$, then $|B_S(p,r)| \leq c^{\log(r/r_0)} \cdot \rho = (r/r_0)^{\log c} \cdot \rho$. So if c is a constant, the number of points in a ball is polynomial in its radius.

The most natural examples of growth-restricted point-sets are found in low-dimensional Euclidean spaces $\mathbb{R}^d$. In these spaces, regular grids and random point-sets in a box are growth-restricted because the number of points in a ball is roughly proportional to the ball's volume. The following Lemma implies that these point-sets whp are 2^{d+1}-growth-restricted.

Lemma 5.3
Let (S,d) be a (c,ρ)-growth-restricted set. Then a random sample $Z \subseteq S$ will with high probability in $|Z|$ be $(2c, \max(c\rho, O(\log |Z|)))$-growth-restricted.

Proof: Let $n := |Z|$. Consider some $p \in Z$ and $r > 0$ such that $B_Z(p,2r)$ contains at least $\max(c\rho, \Omega(\log n)))$ points. We need to show that whp in n, we have

$$|B_Z(p,2r)| \leq 2c|B_Z(p,r)|. \tag{5.2}$$

Let us condition the probability that (5.2) holds on $k := |B_Z(p,2r)|$. These k points in $B_Z(p,2r)$ are selected at random from $B_S(p,2r)$, and since $|B_S(p,2r)| \leq c|B_S(p,r)|$ (recall $|B_S(p,r)| \geq |B_S(p,2r)|/c \geq |B_Z(p,2r)|/c \geq \rho$), the probability for each of the points to be in $B_Z(p,r)$ is at least $1/c$. Thus, we expect k/c points to be in $B_Z(p,r)$. By a standard Chernoff bound, using $k = \Omega(\log n)$, we obtain that $|B_Z(p,r)| \geq k/2c$ whp in n. This shows (5.2), conditioned on a particular arbitrary value of k. But since (5.2) is independent of k, it must hold whp regardless of the actual value of k.

We have shown (5.2) for an arbitrary choice of p and r. But since for each point p, $B_Z(p,r)$ changes only if $r = d(p,q)$ for some $q \in Z$, there are at most $\binom{n}{2}$ different sets $B_Z(p,r)$, and therefore inequality (5.2) whp holds for all pairs p and r, showing the Lemma.
□

Doubling Metrics

Growth-restricted metrics are special case of so-called doubling metrics [GKL03]. In doubling metrics every ball of radius r can be covered by a constant number of balls of radius $r/2$. That this is true for $(O(1), O(1))$-growth-restricted metrics is a consequence of the following technical Lemma.

Lemma 5.4
Let S be (c, ρ)-growth-restricted, and $p \in S$, $r > 0$ such that $|B(p,r)| \geq c^5\rho$. Then there exists a point-set U with the properties

(i) $U \subseteq B(p,r) \setminus B(p,r/2)$,

(ii) $d(u,u') > r/8$ *for all distinct* $u, u' \in U$, *and*

(iii) $(B(p,r) \setminus B(p,r/2)) \subseteq \bigcup_{u \in U} B(u, r/8)$.

Moreover, any point-set U satisfying properties (i)-(iii) will also satisfy

(iv) $|B(u,r/16)| \geq c^{-5}|B(p,r)|$ *for all* $u \in U$, *and*

(v) $|U| \leq c^5$. $\square$

Let us defer the proof, and first show as a corollary that (c, ρ)-growth-restricted metrics are also doubling metrics, if c and ρ are constants.

Corollary 5.5
In a (c, ρ)-growth-restricted metric, every ball $B(p,r)$ can be covered by $\max(c^5\rho, c^5 + 1)$ *balls of radius $r/2$.*

Proof: Fix a point p and radius $r > 0$. If $|B(p,r)| < c^5\rho$, then use $c^5\rho$ balls to cover all points in $B(p,r)$ individually. Otherwise, construct a set U as in Lemma 5.4. Then it directly follows that $B(p,r)$ is covered by the c^5 balls $B(u,r/2)$ for $u \in U$, and the ball $B(p,r/2)$. $\square$

Proof (Lemma 5.4): Let $B := B(p,r) \setminus B(p,r/2)$. Let $U \subseteq B$ be maximal with $d(u,u') > r/8$ for all distinct $u, u' \in U$. So one could construct U greedily by repeatedly selecting points of B that have a distance of more than $r/8$ to all previously selected points. Clearly, this set U satisfies properties (i)-(iii) of the Lemma (see Figure 5-1).

To prove properties (iv) and (v), consider the balls $B_u := B(u, r/16)$ for elements $u \in U$. Since any two points in U are at least $r/8$ apart, we have $B_u \cap B_{u'} = \emptyset$ for $u \neq u'$. On the other hand, each B_u contains a constant fraction of the points in B, which can be shown as follows.

For each $u \in U$, we have $B(p,r) \subseteq B(u, 2r)$. So by the growth-restrictiveness, we have $|B(u,r/16)| \geq c^{-5}|B(u,2r)| \geq c^{-5}|B(p,r)| \geq c^{-5}|B|$, proving property (iv). This implies that each B_u contains $c^{-5}|B|$ elements not contained in any other $B_{u'}$, so we must have $|U| \leq \frac{|B|}{c^{-5}|B|} = c^5$. $\square$

Metric Embeddings

There has been much recent study of embeddings of metric spaces (see e.g. [Mat02, Ind01]). An *embedding* is a mapping from one metric space to another metric space. Embeddings are often used to map complicated metrics to "simple" metrics, such as metrics based on

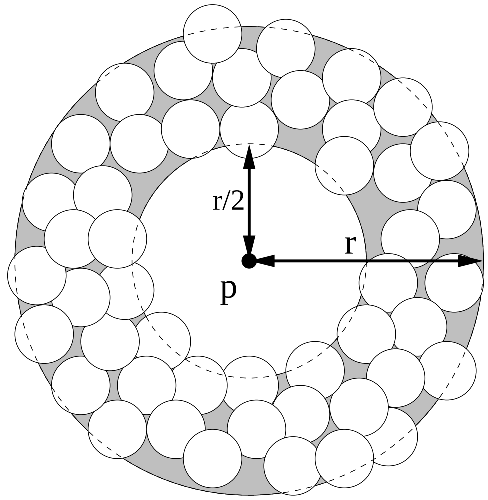


Figure 5-1: Illustration of the covering given in Lemma 5.4. The gray annulus represents $B(p, r) \setminus B(p, r/2)$. The small white discs represent the $B(u, r/8)$ for $u \in U$. They cover all points of S that fall within the annulus.

the Euclidean L_2-norm. For many geometric problems, embeddings allow us to concentrate on solving the problems in simple metrics, and then mapping the solutions back to the complicated metric.

An important measurement of the "accuracy" of a metric embedding is its *distortion*, which is the maximal factor by which pair-wise distances change due to the mapping. It has been observed that growth-restricted metrics can be embedded with $O(\sqrt{\log n})$ distortion into the L_2-metric [Dun01, GKL03]. This embedding is optimal within constants, as there is a corresponding lower bound of $\Omega(\sqrt{\log n})$ for the distortion of embedding a class of $(2, 1)$-growth-restricted metrics into the L_2-metric [GKL03].

5.2.3 Machine Learning Applications

A current thread of machine learning research [TdSL00, Ten98] postulates that feature vectors representing data points that are being analyzed form a low-dimensional manifold within a high-dimensional space. There is no a-priori specification of the manifold; rather, a large set of example points is provided. The distance metric on these points is given in the high dimensional space. Identifying near neighbors among the example points is useful – for example, to implement the standard k-nearest neighbors algorithm for classification, or to identify neighborhoods in the manifold in order to construct local parameterizations of the manifold. Under assumptions of limited curvature (which are also made in the AI literature) and random selection of example points from the manifold, the low-expansion property will hold (as it does for low dimensional Euclidean spaces) and our near-neighbor structure could be applied. In Section 5.7, we will develop an offline version of our data structure that is useful for this purpose.

Host	Pings sent	Pings returned	Fraction returned
MIT	575900	10925	1.897 %
NC	211233	3870	1.832 %
KR	982300	18359	1.869 %

Table 5.1: Numbers of pings sent and returned from random IP addresses.

5.2.4 Computers on the Internet

Given our interest in P2P networks, it seems reasonable to ask whether hosts in a network such as the Internet are distributed in a growth-restricted manner, where the distance considered is the latency between hosts.

Unfortunately, measuring the growth-restrictiveness of the Internet is not an easy problem, since there is not much reliable data on the structure of the Internet. But it is possible to test the growth-restrictiveness relative to any particular node p. This can be done by measuring the latencies from the node p to a random set of nodes on the Internet. We can then check whether $|B(p, 2r)| \leq c|B(p, r)|$ for some small constant c.

The following graphs are based on data collected by Frank Dabek, a graduate student at MIT, during October and November of 2002. His program pinged random IP addresses from three machines: one at MIT, one in North Carolina (NC), and one in South Korea (KR) (see Table 5.1). It turns out that roughly 1.85% of all IP-addresses that were pinged actually returned an answer. Ignoring the issues of firewalls and network problems blocking pings and of not all the 2^{32} IP-addresses being usable on the Internet, this gives an estimate of about 80 million machines on the Internet.

Table 5.2 shows the distribution of ping-response times for the three hosts. The plots show that MIT is pretty well-connected – many hosts are within a short latency distance of MIT. The computer in North Carolina is connected to the Internet via a cable modem, which introduces a high initial latency. This is the reason why almost no hosts are reachable below 150 ms, and the number increases dramatically after that. The host in South Korea is able to reach a small number of hosts in South Korea quickly, but then it takes some time to reach the Internet "at large", leading to the observed delay in contacting most hosts.

In Table 5.3 the same data is depicted as a cumulative fraction of hosts reachable within a certain latency. From this graph, we can read of the sizes of balls $B(p, r)$ for arbitrary latencies r, which allows us to compute $|B(p, 2r)|/|B(p, r)|$ for all radii r (see Table 5.4).

These plots show that the growth factors can be dramatically high. However, it turns out that the factors are only high while $|B(p, r)|$ is small, something that is not obvious from Table 5.4. In Table 5.5 the same data is plotted, but instead of plotting the growth factors in terms of the latency, we plot them in terms of the size of the network within a certain latency. More precisely, the points in these plots are $(|B(p, r)|, |B(p, 2r)|/|B(p, r)|)$, where the x-coordinates are given in percentage of the total sample size. In these plots, it is obvious that the spikes of growth factors occur when the sample is of a miniscule size.

So far, when speaking about growth characteristics, we have concentrated on the growth factor c, ignoring the parameter ρ. By fixing ρ, we could actually determine growth-factors c for the whole network as

$$c := \max_{r, |B(p,r)| \geq \rho} \frac{|B(p, 2r)|}{|B(p, r)|}.$$

What is a suitable value for ρ? The larger we pick ρ, the smaller c becomes. But we will

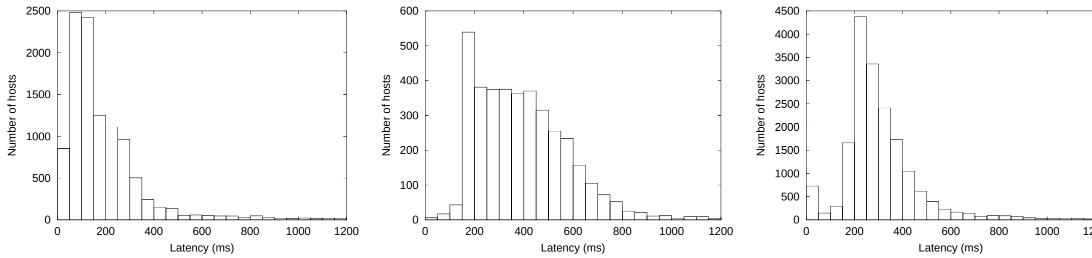


Table 5.2: Number of hosts reached within certain latencies from hosts at MIT (left), in North Carolina (center) and South Korea (right).

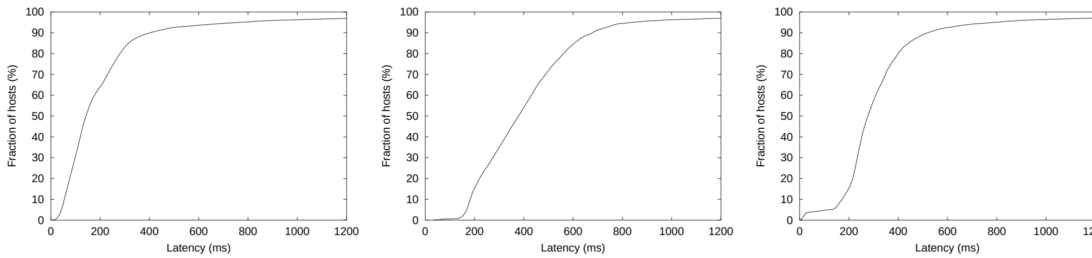


Table 5.3: Cumulative percentage of hosts reached within certain latencies from hosts at MIT (left), in North Carolina (center) and South Korea (right).

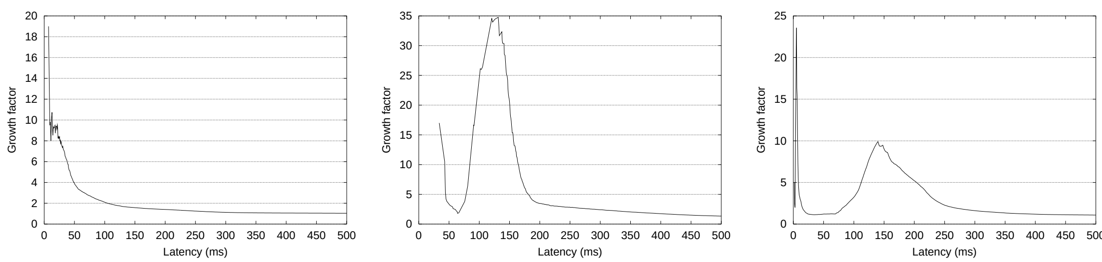


Table 5.4: Growth factors $|B(p, 2r)|/|B(p, r)|$ plotted versus latency r from hosts at MIT (left), in North Carolina (center) and South Korea (right).

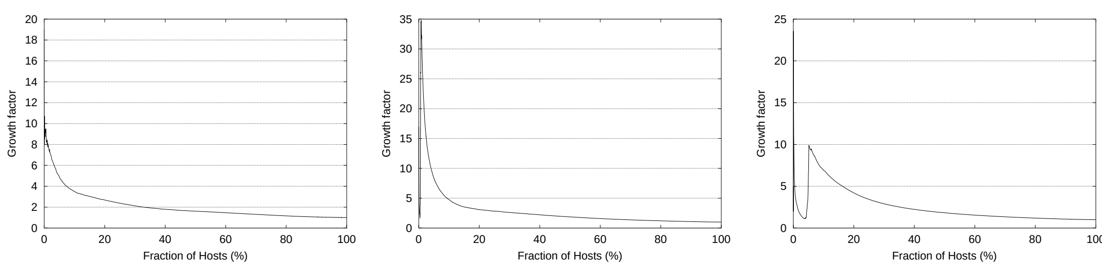


Table 5.5: Growth factors $|B(p, 2r)|/|B(p, r)|$ plotted versus size of $B(p, r)$ for hosts at MIT (left), in North Carolina (center) and South Korea (right).

Host	c
MIT	10.73
NC	34.77
KR	11.04

Table 5.6: Growth-constant c resulting from setting $\rho = 0.001\times$(nodes in sample).

later see that the query times of our protocol are of the form $O(\log n + \rho)$, thus they increase with ρ, leading to a trade-off. Say that we are willing to contact 0.1 % of the nodes in a network to resolve a query (that is 10 nodes in a 10,000 node network). Then we should choose ρ as 0.1 % of the nodes in the sample. This leads to the values of c shown in Table 5.6.

As is obvious from the graphs in Table 5.5 however, the effective values of c for larger latencies are much smaller than the maxima in Table 5.6. Thus, on a large scale the metric is growth-restricted very well.

5.2.5 People in the World

We now consider another natural metric, the distribution of the world's population. Luckily, there is much more data available on this than on the structure of the Internet.

In our experiments we use the data created by the "Gridded Population of the World" project [GPW00]. In this data set, the globe is partitioned into a grid with a resolution of 2.5 arc minutes. For each grid cell, an approximate number of inhabitants was determined. This data was generated from census data, which underwent further statistical refinements. The data is supposed to approximate the world-wide population distribution in 1995, when the total population was 5.67 billion. For our purposes, this seems sufficiently recent enough.

Let us first consider a few examples to describe our methodology. Again, we are going to compute the "local" growth-constants around some points p. In the following examples, we chose Boston (Massachusetts, USA), Freiburg (the author's hometown in southwestern Germany), Tokyo (Japan) and Honolulu (Hawaii, USA) as centers for our calculations.

Table 5.7 shows the distribution of the world population according to distance from Boston, Freiburg, Tokyo and Honolulu. The Boston distribution shows two humps: the first one from 6000 to 9000 kilometers corresponds to Europe and South America, the second (larger) hump from 11,000 to 13,000 kilometers corresponds to Asia (most notably India and China). The distribution for Freiburg suggests that it is more centrally located in the world, most people live within 12,000 kilometers of Freiburg. In Tokyo's plot, the first two spikes correspond to China and India, respectively. Honolulu's plot differs by the fact that the closest significant accumulation of population near the Hawaiian islands is the Californian coast in about 4000 kilometers distance.

In Table 5.8, we again see what fraction of the world is within a certain distance of our four example cities. Table 5.9 shows the growth factors $|B(p, 2r)|/|B(p, r)|$ associated with these distributions. Surprisingly enough, the growth factors are below 5.5 for the first three cities. This compares well to the factor of 4 that we would have expected for a random distribution of people on the globe. This property seems only true for densely populated areas, however. For Honolulu, the growth factors are up to 503. This is due to the fact that when repeatedly doubling the radius, the population size increases dramatically upon hitting the main continental mass.

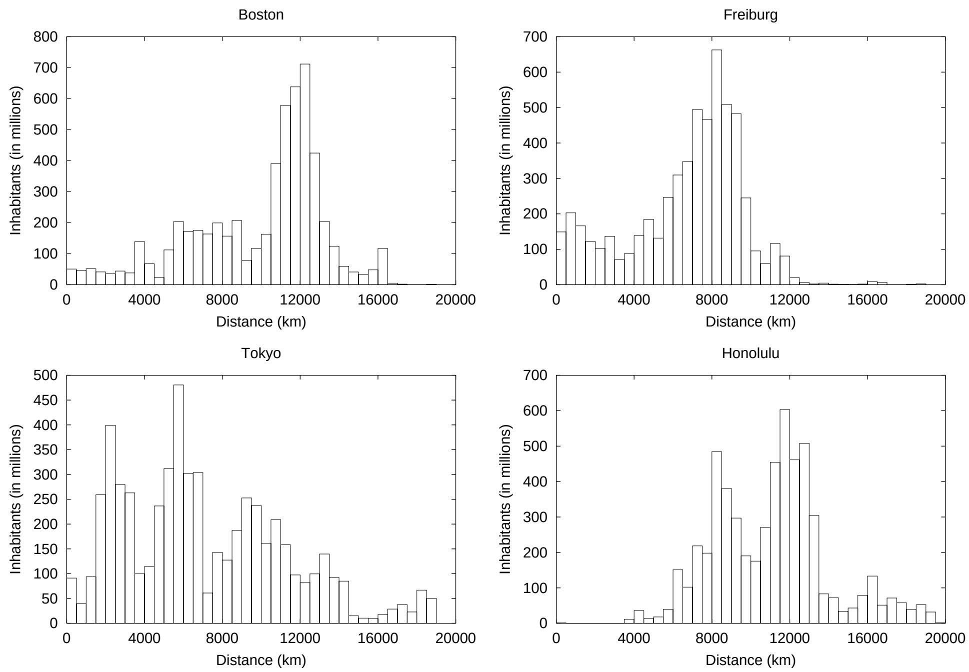


Table 5.7: Number of people living within certain distances from Boston, Freiburg, Tokyo and Honolulu.

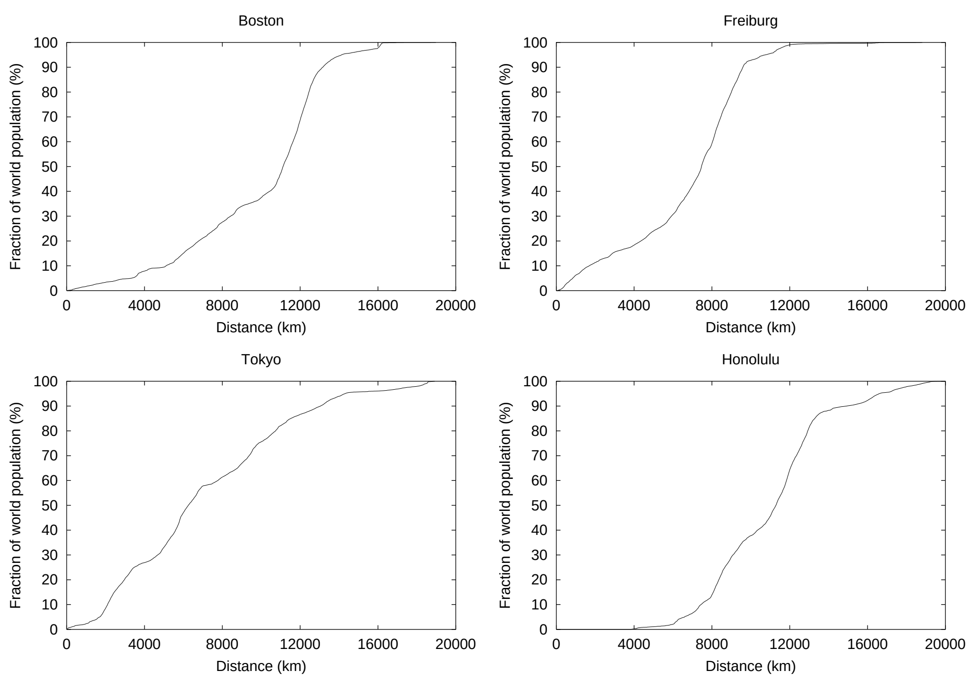


Table 5.8: Cumulative percentage of people within certain distances of Boston, Freiburg, Tokyo, and Honolulu.

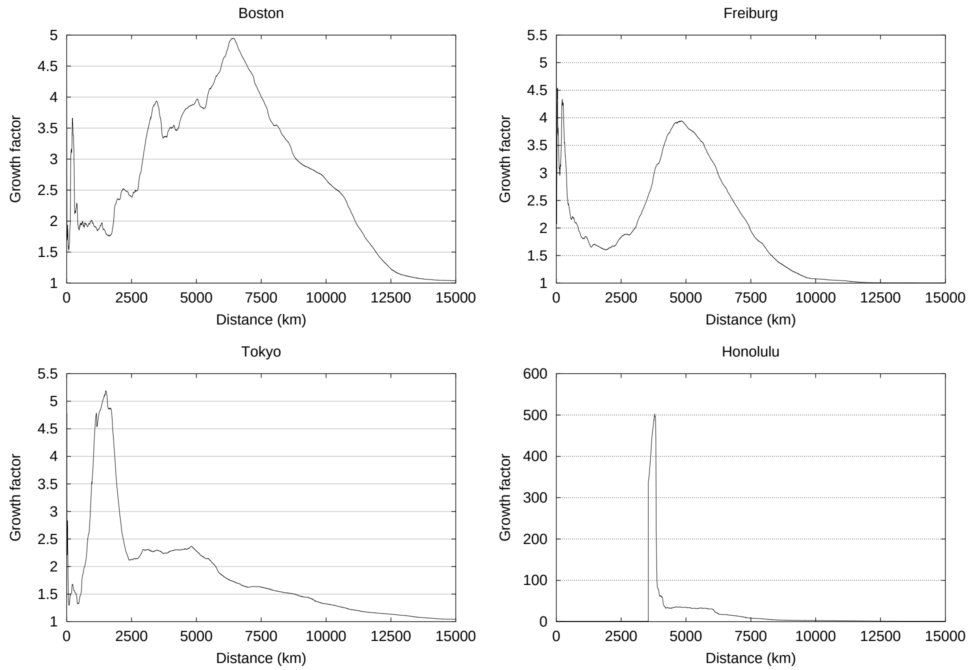


Table 5.9: Growth factors $|B(p, 2r)|/|B(p, r)|$ plotted versus distance r from Boston, Freiburg, Tokyo, and Honolulu.

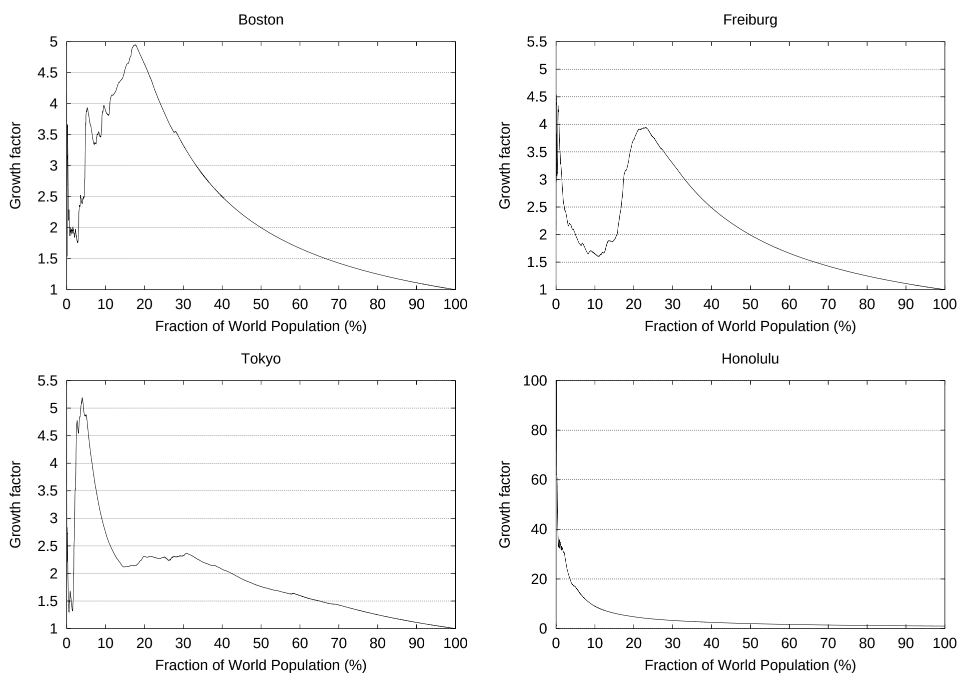


Table 5.10: Growth factors $|B(p, 2r)|/|B(p, r)|$ plotted versus size of $B(p, r)$ for Boston, Freiburg, Tokyo, and Honolulu.

City	c
Boston	4.95
Freiburg	4.34
Tokyo	5.19
Honolulu	115.42

Table 5.11: Growth-constant c resulting from setting $\rho = 0.001\times$ (world population).

As with the experiment for Internet hosts, we set ρ to 0.1% of the world population. Reading off Table 5.10, this yields the growth factors c shown in Table 5.11.

As opposed to the Internet measurements carried out before, there is no reason why we should stop at four samples to verify growth-restrictiveness. We performed the same measurements for all points on the globe with longitude and latitude being integral multiples of 10 degrees, resulting in 614 points.

It turns out that the growth factors are only low for densely inhabited areas of the globe. In Figure 5-2, a representation of growth-constants c, derived by setting $\rho = 0.001\times$(world population), is shown. The smallest value $c \approx 4.04$ is obtained at 50°N 20°E, located in Poland, the largest value $c \approx 453.53$ for 80°N 180°E, near the North Pole.

The converse question is how large we have to set ρ, such that c becomes smaller than a prescribed value, say 10. Figure 5-3 shows a plot summarizing the answer to this question. The largest dots correspond to a value of $\rho \leq 0.001\times$(world population), the smaller dots to successively higher values of ρ. Again, we see that the metric is growth-restricted when seen form densely populated areas, but less so from the rest of the world.

In summary, it seems that restricted to densely populated areas on the globe, the distribution of people is growth-restricted. This might be a result of the fact that in these areas, the population is more or less evenly distributed. This is not true for sparsely populated areas, where we do not observe a growth-restricted population distribution.

5.2.6 P2P Networks

Our primary reason for studying growth-restricted metrics is because they might appear in P2P networks, where distances correspond to latencies or transfer times. By this we mean that P2P networks tend to be c-growth-restricted for some *small* constant c. This assumption has been made before, in the paper by Plaxton et al [PRR99] mentioned in the introduction. In fact, that paper used even stronger assumptions on the underlying metric. While we bound the "growth" of balls $B(p, 2r)$ over $B(p, r)$ from above, Plaxton et al also assume a corresponding linear bound from below. Plaxton et al also state that several common network topologies (such as meshes) have this growth-property, but they acknowledge that no such claim can be made about arbitrary networks.

What do typical P2P networks look like? We could make the reasonable assumption that they consist of random nodes on the Internet, or of computers owned by random people in the world. The measurements performed in the previous sections then seem to imply that such a network will be c-growth-restricted for small c (e.g. $c \leq 10$) for all hosts in *dense areas* of the network. That is, for computers located in densely networked or populated areas, the surrounding network will be well growth-restricted. This implies that the protocols developed in the following will work well for such users. However, our experiments also suggest that the constant c will be large for users in poorly populated

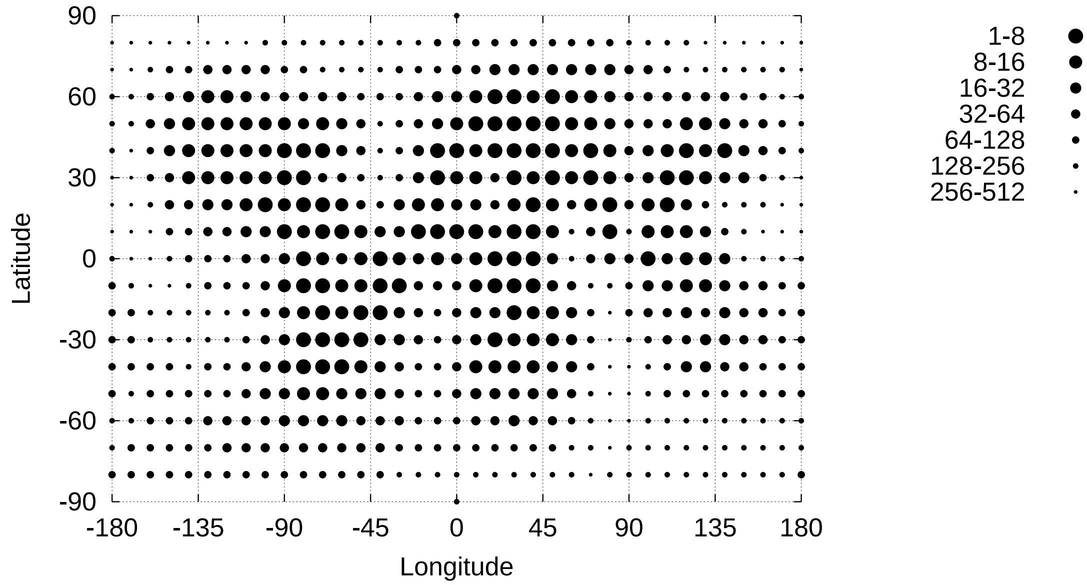


Figure 5-2: Growth factors c of the world population for different points on the globe, setting $\rho = 0.001\times$ (world population). Larger dots correspond to smaller values of c.

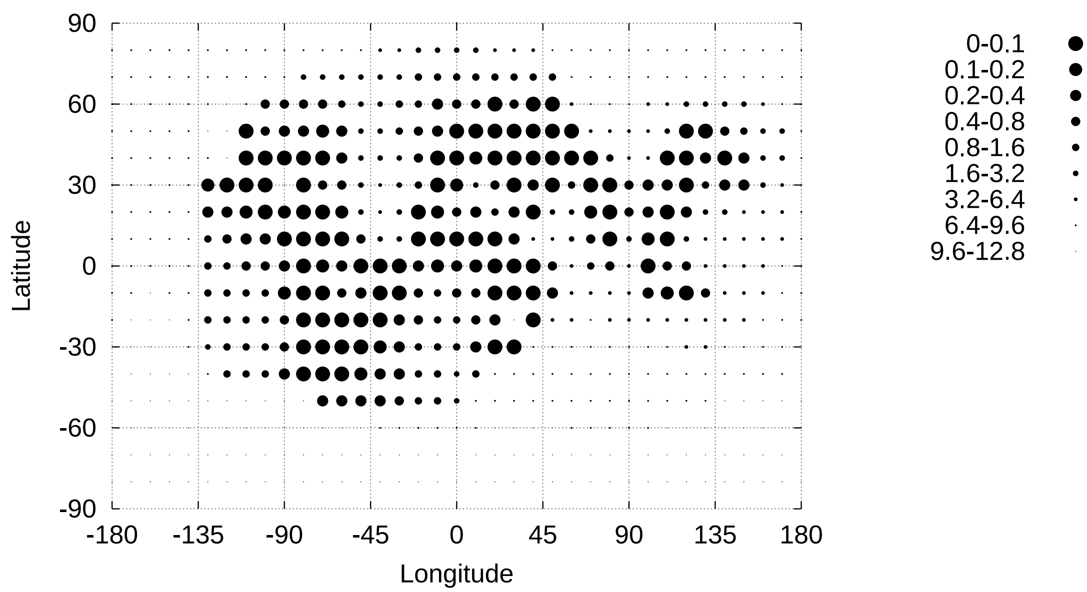


Figure 5-3: Smallest value of ρ (in percentage of the world population) for which $c \leq 10$. The smaller the dots, the larger ρ has to be chosen.

areas of the globe. For these cases, our protocols would lead to poor performance, so that alternative solutions should be found. One might consider using proxys in better connected locations, or keeping a directory of poorly connected nodes.

5.3 Finding Nearest Neighbors

In this section, we are going to present a protocol for finding nearest neighbors in (c, ρ)-growth-restricted metrics. For the rest of this chapter, we are going to assume that c is a constant with $c \geq 2$, which implies that $\log \log c \geq 0$. Before defining our data structure and stating the query algorithms, we will analyze a simple sampling based algorithm to find nearest neighbors in growth-restricted metrics. This algorithm will later motivate the design of our data structure.

5.3.1 A Sampling Algorithm

Let us consider the following algorithm to find a nearest neighbor in S for a point $q \in M$.

Algorithm 5.6 (Sampling to Find Nearest Neighbor)

1. Let $p_0 :=$ an arbitrary point in S, $i := 0$
2. Repeat until $p_i =$ nearest neighbor of q in S:
   (a) Let $S' :=$ random sample of $(4 + 2 \log \log c)c^2$ points in $B_S(p_i, 2d(p_i, q))$.
   (b) Let $p_{i+1} :=$ closest point to q in S'.
   (c) Let $i := i + 1$.
3. Output p_i.

Intuitively, the algorithm constructs a sequence of points (p_i) that gets successively closer to the query q, and therefore eventually ends up with the nearest neighbor of q. There are several issues that prevent us from implementing the algorithm as stated, for example how to compute the random samples S', and how to determine that p_i is the nearest neighbor of q. But ignoring these problems for the moment, let us analyze the above algorithm.

Lemma 5.7
Algorithm 5.6 finds the nearest neighbor of any point q in $O(\log |S|)$ steps whp in any $(c, 1)$-growth-restricted set S. □

For the proof, we need the following bit of notation.

Definition 5.8 (q-rank)
Let (M, d) be a metric space, $S \subseteq M$ be a finite point-set, and $q \in M$ be a point. Let $<$ be an ordering on S such that $s < s' \implies d(q, s) \leq d(q, s')$. Then the q-rank of an element of S is its position in the ordering $<$ (the first element having q-rank 1, the second q-rank 2, and so on).

Note that by slightly perturbing the metric, we can assume that all pairwise distances are distinct, and therefore the q-rank is uniquely defined. We will make this assumption whenever referring to q-ranks. □

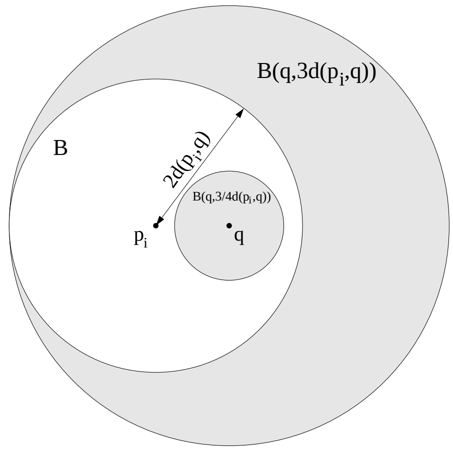


Figure 5-4: The algorithm draws samples from the ball $B = B(p_i, 2d(p_i, q))$ (shown in white), which is wedged between the balls $B(q, \frac{3}{4}d(p_i, q))$ and $B(q, 3d(p_i, q))$ (shown in gray). Samples fall in the smaller gray ball with constant probability.

Proof (Lemma 5.7): Let r_i be the q-rank of point p_i in the execution of the algorithm. We will now show that the sequence (r_i) performs an exponentially decreasing random walk, and therefore quickly converges to 1. This corresponds to finding q's nearest neighbor in S, which has q-rank 1. Our main goal therefore is to show that r_{i+1} is likely to be smaller than r_i.

Consider some point p_i with q-rank r_i. Let $B := B(p_i, 2d(p_i, q))$ be the ball from which we draw the sample S' to determine p_{i+1} in our algorithm (see Figure 5-4). By the triangle inequality we have that $B(q, d(p_i, q)) \subseteq B$, i.e. B contains all points in S that are closer to q than p_i. In particular, it contains all elements of S with q-ranks between 1 and $r_i/2$. We will show that our sample is likely to include one of these elements.

We have $B \subseteq B(q, 3d(p_i, q))$ by the triangle inequality. Since $B(q, \frac{3}{4}(p_i, q))$ contains fewer than r_i points (as it does not contain p_i, the r_i-th closest point to q), we have

$$|B| \;\leq\; |B(q, 3d(p_i, q))| \;\leq\; c^2 |B(q, \frac{3}{4}d(p_i, q))| \;<\; c^2 r_i. \tag{5.3}$$

So what is the probability that at least one point in S' has q-rank $r_i/2$ or lower? The probability for a single random element of B to have a q-rank of at most $r_i/2$ is at least $(r_i/2)/(c^2 r_i) = 1/(2c^2)$. Thus, the probability that *at least one* of α samples has such a q-rank is at least

$$1 - \left(1 - \frac{1}{2c^2}\right)^{\alpha} \geq 1 - e^{-\alpha/2c^2} =: P(\alpha).$$

On the other hand, with probability at most $1 - P(\alpha)$, the q-rank of the next element p_{i+1} might increase over r_i. But by inequality (5.3), the q-rank of all elements in B is at most $c^2 r_i$, so $r_{i+1} \leq c^2 r_i$. Thus we have

$$E[\log r_{i+1}] \leq P(\alpha)(\log r_i - 1) + (1 - P(\alpha))(\log r_i + 2\log c) = \log r_i + (1 - P(\alpha))2\log c - P(\alpha).$$

To guarantee convergence within $O(\log n)$ steps, we must have that $\log r_i$ decreases by a constant in expectation in every step. By a Chernoff bound we can then convert this into a high probability convergence proof for the sequence (r_i).

So a sufficient condition for convergence is

$$(1 - P(\alpha))2\log c - P(\alpha) < -1/2 \iff P(\alpha) > \frac{2\log c + 1/2}{2\log c + 1} \iff e^{\alpha/2c^2} > 4\log c + 2.$$

For $c \geq 2$, we have $4\log c + 2 \leq 6\log c$. Thus it suffices to choose α such that $\alpha > 2c^2 \cdot \ln(6\log c)$, which is satisfied by $\alpha = (4 + 2\log\log c)c^2$. $\square$

5.3.2 The Basic Data Structure

We will now convert the above sampling idea into an actual data structure. The basic idea is to pre-choose the samples for each point in S. The data structure is defined as follows, where $\alpha = (19 + 6\log\log c)c^2$ is a constant dependant on c:

1. Let $s_1, s_2, \ldots, s_n$ be a random permutation of the points in S.
2. For $i = 1, \ldots, n$ and $r > 0$ set $F(i, r) :=$ indices of the first α points s_j with $d(s_i, s_j) \leq r$ among $s_{i+1}, s_{i+2}, \ldots, s_n, s_1, \ldots, s_{i-1}$. We call the $F(i, r)$ the *finger lists* of s_i.

So for large enough r, $F(i, r)$ just contains the α points following s_i in the ordering, while for smaller r, $F(i, r)$ contains points further spread out along the ordering.

We will show in Section 5.3.4 that the above data structure whp uses only $O(\log n)$ space per node, because for each s_i, whp only $O(\log n)$ of the $F(i, r)$ are actually distinct.

Finding Nearest Neighbors

We can use the above data structure to find the nearest neighbor of any given point $q \in M$ by adapting the sampling algorithm in a straightforward manner:

Algorithm 5.9 (Finding Nearest Neighbors)

1. Let $i_0 :=$ arbitrary number $\in \{1, 2, \ldots, n\}$.
2. Let $i := i_0$.
3. Let $p := s_i$.
4. Repeat:
   (a) Let $r := 2d(s_i, q)$.
   (b) If $F(i, r)$ wraps around the starting index i_0 (i.e. $i < i_0$ and all indices in $F(i, r)$ are $\geq i_0$), return p.
   (c) If there is an element s_j in $F(i, r)$ such that $d(s_j, q) < d(s_i, q)$ then let j be the first index with this property in $F(i, r)$. Otherwise, let j be the last index in $F(i, r)$.
   (d) If $d(s_j, q) < d(p, q)$ then let $p := s_j$.
   (e) Let $i := j$.

Intuitively, the above algorithm is derived from Algorithm 5.6 by replacing every occurrence of a sample around s_i with radius r by the set $F(i,r)$. Since the finger lists $F(i,r)$ are not completely independent, this introduces some dependency among the samples, but will see that this effect is not too large. In fact, we will see in the following section that this algorithms has the same sort of performance guarantees as Algorithm 5.6.

We remark that step 4c of the above algorithm can be replaced by the slightly more efficient

> If there is an element s_j in $F(i,r)$ such that $d(s_j,q) < d(s_i,q)$ then let j be the **index of the element closest to** q in $F(i,r)$. Otherwise, let j be the last index in $F(i,r)$.

It is not hard to see that Algorithm 5.9 visits a superset of the elements s_i that this modified algorithm visits, since it never skips an element closer to q than the currently visited element. So any performance bound for the original algorithm also holds for the modified one.

5.3.3 Analysis: Running Time

In this section, we will analyze the running time of our nearest neighbor algorithm.

Theorem 5.10
Algorithm 5.9 returns the nearest neighbor of a point q in a set S (even if (S,d) is not growth-restricted). For a (c,ρ)-growth-restricted set (S,d), the algorithm takes $O(\log n + \rho)$ steps whp.

Proof: Our proof consists of two parts. First, we show that the above algorithm outputs the correct answer. For this, we show that the algorithm will not "skip over" the nearest neighbor of q, i.e. the nearest neighbor will be among the points s_i visited by the algorithm.

Second, we show that the algorithm's running time is $O(\log n + \rho)$. This proof is essentially the same as for Lemma 5.7 with the added complication that the finger lists $F(i,r)$ are not completely independent as were the samples in Algorithm 5.6. The main problem is therefore to argue that the finger lists are "independent enough". We will do this using the deferred decision principle to choose the ordering of the elements of S.

But let us first show the correctness of the algorithm. By symmetry, it suffices to consider the case $i_0 = 1$. We will show the invariant that at the beginning of the loop, p is the point closest to q among $\{s_1,\ldots,s_i\}$. This directly implies the correctness of the algorithm.

We show the invariant by contradiction. Suppose that after moving from s_i to s_j, the invariant no longer holds. Then there must be a point s_ℓ ($i < \ell < j$) with $d(s_\ell,q) < d(s_i,q)$. This implies $d(s_\ell,s_i) < 2d(s_i,q) = r$. But such an s_ℓ should have been included in the finger list $F(i,r)$, and therefore would not have been skipped. The contradiction shows the invariant, i.e. the algorithm is correct.

To analyze the running time of the algorithm, we will employ two tricks. The main problem with carrying over the analysis of the sampling algorithm from Lemma 5.7 is the fact that the "samples" $F(i,r)$ are not chosen independently. We deal with this as follows.

1. We can use the deferred decision principle by choosing the ordering on S only as needed. We start with an empty ordering. When querying an $F(i,r)$, we decide on the ordering after s_i by repeatedly removing random elements from $S \setminus \{s_1,s_2,\ldots,s_i\}$, and appending them to the end of the ordering. While doing this, some of these elements will appear in $F(i,r)$. We stop when we choose the first element s_j that is

closer to q than s_i, or with the α-th element added to $F(i,r)$. Note that we did not need to construct the ordering beyond the point s_j that we move to next.

The advantage of the deferred decision principle is that we can assume our the analysis that $F(i,r)$ is chosen, while not randomly from S, at least randomly among the elements in $S \setminus \{s_1, \ldots, s_i\}$, and independently of any previously seen $F(i', r')$'s. This will be a sufficient replacement for full independence in our analysis.

2. The other trick for our analysis is to make sure that $S \setminus \{s_1, \ldots, s_i\}$ is growth-restricted. For this purpose, we split the analysis into two parts, and bound the running time of the algorithm on the subsets $\{s_1, \ldots, s_{n/2}\}$, and $\{s_{n/2+1}, \ldots, s_n\}$. We will bound the running time to $O(\log n + \rho)$ with high probability on both subsets, leading a total running time of $O(\log n + \rho)$.

   By symmetry, it suffices to consider the first half of the elements. But these are a random subset of S. Thus, Lemma 5.3 implies that $S \setminus \{s_1, \ldots, s_i\}$ is $(2c, \max(\log n, c\rho))$-growth-restricted whp for $1 \le i \le n/2$.

We can now carry over the proof from the sampling algorithm almost verbatim. We consider the random walk that the q-ranks of the points s_i encountered by the algorithm perform. If suffices to show that in expectation, the q-rank reduces by a constant factor in every step. Note that $F(i,r)$ is a random sample from $B := B(s_i, r) \setminus \{s_1, \ldots, s_i\}$.

Let us consider the case where $|B| \ge \max(\rho, \log n)$ first. Suppose s_i's q-rank is r_i. Since B is $(2c, \max(\log n, c\rho))$-growth-restricted, we conclude from equation (5.3) (page 81) that $|B| \le 4c^2 r_i$. The probability for a single random element of B to have half of s_i's rank is therefore at least $(r_i/2)/(4c^2 r_i) = 1/(8c^2)$. Thus, the probability that *at least one* of α samples has such a rank is at least

$$1 - \left(1 - \frac{1}{8c^2}\right)^{\alpha} \ge 1 - e^{-\alpha/8c^2} =: P(\alpha).$$

By the same analysis as in the proof of Lemma 5.7, it suffices to have

$$\begin{aligned}
& (1 - P(\alpha)) 2 \log 2c - P(\alpha) < -1/2 \\
\iff\ & P(\alpha) > \frac{2 \log 2c + 1/2}{2 \log 2c + 1} \\
\iff\ & e^{\alpha/8c^2} > 4 \log 2c + 2 = 4 \log c + 6.
\end{aligned}$$

For $c \ge 2$, we have $4 \log c + 6 \le 10 \log c$. Thus it suffices to choose α such that $\alpha > 8c^2 \cdot \ln(10 \log c)$, which is satisfied by $\alpha = (19 + 6 \log \log c)c^2$.

In at most $\max(\rho, \log n)$ steps we will then get to the case of $|B| < \max(\rho, \log n)$, where we can make no inferences from the growth-restriction of S. In particular, we cannot infer that the q-rank of the next element visited after s_i is at most $4c^2 r_i$. But we can still conclude that the next element's rank is at most $4c^2 \max(\rho, \log n)$. This jump in rank can happen only $\max(\rho, \log n)$ times, and therefore extends our random walk only by $O(\max(\rho, \log n))$ steps. Thus, the total running time remains $O(\max(\rho, \log n)) = O(\log n + \rho)$. □

5.3.4 Analysis: Space Requirements

At first, it might seem that our data structure consists of finger lists $F(i,r)$'s for an infinite number of values for r. But it turns out that for fixed i, whp only $O(\log n)$ of the $F(i,r)$

are distinct.

Lemma 5.11
Let each finger list $F(i,r)$ contain α elements. Then whp $|\bigcup_{r>0} F(i,r)| = O(\alpha \log n)$. Thus, our data structure requires only $O(\alpha \log n)$ storage per point in S.

Proof: By symmetry, it suffices to consider $i = 1$. Let X_j be the event that s_j occurs in some finger list $F(1,r)$. Our goal is to show that

$$\sum_{j=2}^{n} X_j = O(\alpha \log n) \quad \text{whp.} \tag{5.4}$$

We will first prove that equation (5.4) holds in expectation, i.e. $E[\sum X_j] = O(\alpha \log n)$. Then, we will argue that the X_j are actually independent, so a standard application of Chernoff bounds shows the result.

The event X_j happens if and only if s_j is among the α elements closest to s_1 in the set $\{s_2, s_3, \ldots, s_j\}$. Since the ordering is random, the probability for each element in the set $\{s_2, \ldots, s_j\}$ to be among the α closest to s_1 is the same, namely $\alpha/(j-1)$. Thus, $E[X_j] = \alpha/(j-1)$, and therefore

$$E[\sum_{j=2}^{n} X_j] = \sum_{j=2}^{n} E[X_j] = \sum_{j=2}^{n} \alpha/(j-1) = \alpha \cdot H_{n-1} = O(\alpha \log n),$$

where $H_n = \sum_{i=1}^{n} 1/i$ is the n-th Harmonic number.

To obtain the claimed high probability result, it suffices to show that the X_j are independent. To see this, consider constructing the ordering of $S \setminus \{s_1\}$ from the end. That is, we choose s_n at random from S, then s_{n-1} at random among the remaining elements, and so on. When we pick s_j, with probability $\alpha/(j-1)$ it is among the α smallest elements not yet added to the ordering. But note that this event X_j is independent of all the $X_{j'}$ with $j' > j$. This shows the claim. □

We can also view the finger lists $F(i,r)$ as pointers to other nodes in the data structure. The above Lemma shows that the out-degree of a node is $O(\alpha \log n)$ whp. We will now show that the in-degree of a node, or equivalently, the number of nodes which contain a given node in its finger lists, is also bounded. The interesting thing about the following bound is that it holds unconditionally, i.e. not just with some probability.

Lemma 5.12
For any node i, the number of other nodes j in whose finger lists i appears, i.e. the number $|\{j \mid \exists r : \; i \in F(j,r)\}|$, is $O(\alpha \log n + \rho)$.

Proof: The basic idea of the proof is to partition S into sets $S = S_1 \cup S_2 \cup \ldots S_m$ with the property that for each S_k, any pair of element in S_k is closer to each other than to s_i.

Why is this useful? Consider some fixed set S_k. Recall that an element $s_j \in S_k$ will only include s_i in one of its finger lists if there are *not* α elements closer to s_j in the ordering between s_j and s_i. But since the elements of S_k are closer to each other than to s_i, this means that only the α elements of S_k closest before s_i in the ordering can include s_i in their finger lists.

We thus obtain an upper bound of $O(\alpha m)$ on the number of elements that include s_i in their finger lists. It remains to determine a bound on m, the number of sets that we partition S into.

First, we ignore the $c^5\rho$ elements closest to s_i, assuming that in the worst case, they all include s_i in their finger lists. For the remaining elements of S, we will repeatedly invoke Lemma 5.4. This Lemma shows how to partition the ring $B(s_i, r) \setminus B(s_i, r/2)$ into at most c^5 balls of the form $B(u, r/8)$, for u in some set U_r. The elements in each of these balls are closer to each other than to s_i, and thus we can use these balls as sets S_k's.

This suggests using Lemma 5.4 for all of S by setting r equal to successive powers of 2. If we perform a naive bound on the total size of the sets U_r constructed this way, we obtain $m = \Theta(c^5 \log D)$, where D is the diameter of the set S. We have to exert a little more care to reduce the factor $\log D$ to $\log n$.

First, let us choose the sets U_r in a specific order. We construct the sets U_r by starting with $r = D$, and then repeatedly *halving* the value of r. After constructing each U_r, we delete all points in $\bigcup_{u \in U_r} B(u, r/8)$ from S. This has the useful consequence that the balls $\{B(u, r/16) \mid u \in U_r, r > 0\}$ are all disjoint, as follows from a simple triangle inequality.

For fixed $\ell \in \{5, \ldots, \log_c(n/\rho)\}$ now consider the set

$$V_\ell := \bigcup \left\{ U_r \mid c^\ell \rho \leq |B(p, r)| \leq c^{\ell+1}\rho \right\}.$$

For each $u \in U_r \subseteq V_\ell$ we have $|B(u, r/16)| \geq c^{-5} \cdot c^\ell \rho$ by Lemma 5.4. This implies that V_ℓ can have at most $\frac{c^{\ell+1}\rho}{c^{\ell-5}\rho} = c^6$ elements. Summing over all possible values of ℓ, we see that are at most $c^6 \log_c(n/\rho) = O(\log n)$ elements in all the sets U_r. This is therefore the total number of balls in our collection of S_k's. $\Box$

5.4 Range Searches

There are two kinds of range searches that are frequently considered in metric spaces:

(a) Given a point $q \in M$ and a radius $R > 0$, return all points $p \in S$ with $d(p, q) \leq R$.

(b) Given a point $q \in M$ and a number $K \geq 1$, return the K closest points to q in S.

We will now see that the first question can be efficiently answered by a trivial modification of our nearest neighbor algorithm. The second question requires considerably more work and an extension of our data structure, to be answered efficiently. In summary, we can can answer both kinds of queries in time $O(\log n + \rho + K)$, where K is the size of the answer.

5.4.1 Finding All Points Within a Given Distance

To report all points within distance R of a given query q, we modify the nearest neighbor Algorithm 5.9. The changes are noted in **bold-face** in the following algorithm.

Algorithm 5.13 (Find All Elements Within Radius R of q)

1. Let $i_0 :=$ arbitrary number $\in \{1, 2, \ldots, n\}$.
2. Let $i := i_0$.
3. **If** $\mathbf{d(s_i, q) \leq R}$ **then let** $\mathbf{P := \{s_i\}}$ **else let** $\mathbf{P := \emptyset}$.
4. Repeat:
   (a) Let $r := \mathbf{d(s_i, q) + max(d(s_i, q), R)}$.
   (b) If $F(i, r)$ wraps around the starting index i_0 (i.e. $i < i_0$ and all indices in $F(i, r)$ are $\geq i_0$), return $\mathbf{P}$.
   (c) If there is an element s_j in $F(i, r)$ such that $d(s_j, q) < d(s_i, q)$ then let j be the first index with this property in $F(i, r)$. Otherwise, let j be the last index in $F(i, r)$.
   (d) $\mathbf{P := P \cup \{s_\ell \in F(i, r) \mid d(q, s_\ell) \leq R\}}$.
   (e) Let $i := j$.

Lemma 5.14
Algorithm 5.13 outputs all points in S that are within distance R of q. For (c, ρ)-growth-restricted metrics, its running time is $O(\log n + \rho + K)$, where K is the size of the output.

Proof: We only point out the differences to the analysis of the nearest neighbor algorithm 5.9 (cf. proof of Theorem 5.10). To show correctness, note that we will "see" and therefore output all points within distance R of q. This is because $r \geq d(s_i, q) + R$ is chosen large enough so that $F(i, r)$ does not skip any points in $B(q, R)$.

For the running time, the analysis for the nearest neighbor algorithm can be carried over, except when s_i's q-rank is less than K (where K is the number of elements within distance r of q), in that case with probability $P(\alpha)$, the next element's q-rank is at most c^2K (a weaker statement than having the rank halve). Since this can happen at most K times, the running time increases to $O(\log n + \rho + K)$. □

5.4.2 Finding the K Nearest Neighbors

At first, it might seem that one can reduce this problem to the previous problem: simply determine the radius r such that $|B(q, r)| = K$, and then use the above algorithm to find all points in $B(q, r)$. But how do we determine this radius r? It would be too time-consuming to always maintain these radii r for all possible values of q and K. We instead maintain some information that allows us to efficiently compute r from q and K.

Extending the Data Structure

To support finding the K nearest neighbors, we extend our data structure as follows. In addition to the finger lists $F(i, r)$, we store at each node:

- A radius $R(i)$ such that $c \max(\log n, \rho) \leq |B(s_i, R(i))| \leq 7c \max(\log n, \rho)$.

These radii $R(i)$ will allow us to efficiently compute the radius r satisfying $|B(q, r)| = K$. Our algorithm is based on the following skeleton.

Algorithm 5.15 (Find K Nearest Neighbors of q)

1. Let $s_i :=$ nearest neighbor of q in S.
2. Let $r := R(i)$.
3. Repeat until $|B(q, r)| \geq K$:
   (a) Increase r.
4. Output the K elements closest to q in $B(q, r)$.

The rest of this section is concerned with the proof of the following performance guarantee.

Lemma 5.16
There is an implementation of the "Increases r" operation in Algorithm 5.15, such that the algorithm contacts a total of $O(n + \log \rho + K)$ nodes to return the K nearest neighbors of a point q.

Proof: Note that if $K \leq \max(\log n, \rho)$, the above algorithm already solves our problem in time $O(\log n + \rho)$. In all other cases, we have to find a suitable way to implement the "Increase r" step. Intuitively, we want to increase r such that $|B(q, r)|$ increases by a constant factor. We will illustrate this by considering a special kind of growth-restricted metrics.

A Simple Case: Plaxton's Metrics. Implementing "Increase r" would be easy if our definition of growth-restriction did not only include an upper bound, but also a lower bound on $|B(p, 2r)|$, i.e. if there was a constant $c' > 1$ such that

$$c' \cdot |B(p, r)| \leq |B(p, 2r)| \leq c \cdot |B(p, r)| \tag{5.5}$$

for all $p \in S$ and $r > 0$ with $B(p, r) \neq \emptyset$. This assumption was made by Plaxton et al in their work on P2P data storage [PRR99].

If (5.5) holds, then the "Increase r" operation can simply be a doubling of r. The lower bound in (5.5) guarantees that only $\log_{c'} K = O(\log n)$ doublings are necessary, and the upper bound guarantees that $|B(q, r)| = O(K)$ at all times. The latter is important since the computation of $|B(q, r)|$ takes time $O(\log n + \rho + |B(q, r)|)$. This is equal to $O(|B(q, r)|)$, since $|B(q, r)| \geq \max(\log n, \rho)$. Because $|B(q, r)|$ increases by a constant factor in each iteration of the loop, the total running time collapses as a geometric sum to $O(K)$. Adding to this the time to find the nearest neighbor s_i of q, this yields a running time of $O(\log n + \rho + K)$.

General Growth-Restricted Metrics. We will now consider the general case of (c, ρ)-growth-restricted metrics, and describe a procedure that given a radius r will output a radius r' such that

$$\gamma' \cdot |B(q, r)| \leq |B(q, r')| \leq \gamma \cdot |B(q, r)| \tag{5.6}$$

for constants $\gamma > \gamma' > 1$. The running time of this procedure will be $O(|B(q,r)|)$. By the same analysis as for the simple case above, that brings the total running time of the range search algorithm to $O(\log n + \rho + K)$.

Intuitively, for the correct value of r', there will be many elements in $B(q,r') \setminus B(q,r)$. These elements will be interleaved with the elements of $B(q,r)$ in the ordering of the data structure. We will consult the finger lists of elements in $B(q,r)$ to provide possible elements of $B(q,r') \setminus B(q,r)$.

More concretely, our procedure to compute r' from r is the following.

Algorithm 5.17 (Increase r)

1. Let $b_1, b_2, \ldots, b_\ell$ be the elements of $B(q,r)$.
2. For $i = 1, 2, \ldots, \ell$, let a_i be the element closest to b_i in b_i's finger lists that satisfies $d(b_i, a_i) > 2r$.
3. Return $r' :=$ median of $\{\ d(q,a_i) \mid 1 \le i \le \ell\ \}$.

The elements a_i are all points outside $B(q,r)$. Some of the a_i might be far away from q, so to obtain an r' such that $B(q,r')$ is not too large, we take the median distance from q to the a_i as the new radius.

Before showing that the above choice of r' satisfies (5.6), let us briefly comment on implementation details. To compute a_i, we have to look up the finger list $F(b_i, r_i)$ with minimal r_i that contains an element a with $d(b_i, a) > 2r$. This means we have to refer to $F(b_i, 2r)$ or the next different $F(b_i, r_i)$ with larger $r_i > 2r$.

It is also possible that an a_i of the desired form does not exist. In that case we can replace it by a dummy element at distance ∞ for the purpose of the median-computation. Unless, $|B(q,r)| = \Omega(n)$, whp more than half of the a_i's actually exist, so our algorithm still works. If $|B(q,r)| = \Omega(n)$, we can simply return $r' = \infty$, which satisfies (5.6).

We note that the above algorithm can be simplified by instead of computing the median, we pick a random element of $\{\ d(q,a_i) \mid 1 \le i \le \ell\ \}$ as r'. For this we only have to compute a_i for a random element $b_i \in B(q,r)$. With constant probability this will lead to a good choice, and we can redo our random selection if $|B(q,r')|$ turns out to be too large.

After these observations, we will now show that inequality (5.6) holds, in particular that if $|B(q,r)| \ge \rho$ (which holds in Algorithm 5.15), we have

$$\left(1 + \frac{1}{2\alpha}\right) |B(q,r)| \le |B(q,r')| \le 7c^3 |B(q,r)|. \tag{5.7}$$

The lower bound follows from the fact that the a_i's are not in $B(q,r)$, and we will include at least half of them in $B(q,r')$. The upper bound follows because each b_i is likely to include an element of q-rank between $|B(q,r)|$ and $7c^3|B(q,r)|$ in its finger lists.

Lower Bound. For the lower bound in (5.7), note that $B(q,r')$ will include at least half of the a_i's, because we selected their median distance to q as the new radius r'. Also, none of the a_i's are in $B(q,r)$, because $d(q,a_i) \ge d(a_i,b_i) - d(q,b_i) > 2r - r = r$. This does not mean, however, that $B(q,r')$ contains $\frac{1}{2}|B(q,r)|$ new elements, because there might be repetitions among the a_i's. We will now show that at most α consecutive a_i's can be the same.

Let us assume that $b_1, b_2, \ldots, b_\ell$ is actually the order in which the elements of $B(q,r)$ appear in the random ordering underlying our data structure. Then we can show that for $j \geq i+\alpha$, we will have $a_i \neq a_j$. To see this, note that b_{i+1}, b_{i+2}, $\ldots$, $b_{i+\alpha}$ will be between b_i and a_j in the ordering. Since $d(b_i, b_{i+t}) \leq d(q, b_i) + d(q, b_{i+t}) \leq 2r$, there will be α elements b_{i+t} in the ordering between b_i and a_j that have distance at most $2r$ from b_i. Thus, a_j cannot appear in b_i's finger lists if $d(b_i, a_j) > 2r$, so we must have $a_i \neq a_j$.

We have just shown that each a_i is equal to at most α other a_j's, and therefore the number of a_i's within distance r' of q is at least $|B(q,r)|/(2\alpha)$, showing the lower bound in (5.7).

Upper Bound. To show the upper bound in (5.7), we have the prove that more than half of the a_i are "not too far" from q, i.e. r' is not too large. For this, we define two sets of points in S, where R is a "not-too-large" distance to chosen later.

- $U := \{ \ p \in S \mid d(p,q) \leq 3r \ \}$.
- $V := \{ \ p \in S \mid 3r < d(p,q) \leq R \ \}$.

We will choose R, and therefore V, large enough so that in the ordering of the data structure, the first element of $B(q,R)$ following b_i will be in V (and not in U) with probability at least $2/3$. Let X_i be the event that this happens.

Suppose X_i happens for some i, and let $v_i \in V$ be the first element of V following b_i. We have $d(b_i, v_i) \geq d(q, v_i) - d(q, b_i) > 3r - r = 2r$. So v_i would be considered as choice for a_i, if it is included in b_i's finger lists. In that case, $d(a_i, b_i) \leq d(v_i, b_i) \leq d(v_i, q) + d(q, b_i) \leq R + r$.

If, on the other hand, v_i is *not* in the finger lists of b_i, then there must be α elements in b_i's finger lists that come before v_i in the ordering, and are within distance $d(b_i, v_i)$ of b_i. Since we assumed that X_i occurred, none of these α elements are in U. So their distance to q must be more than $3r$, and the distance to b_i more than $2r$. Thus, they will be considered as choices for a_i. This implies that a_i is no further from b_i than these elements, and therefore $d(q, a_i) \leq d(b_i, q) + d(b_i, a_i) \leq d(b_i, q) + b(b_i, v_i) \leq 2d(b_i, q) + b(q, v_i) = 2r + R$.

For the choice of R, note that the probability of a single b_i being followed by an element of V before any element of U (i.e. the event X_i) is roughly $|V|/(|U| + |V|)$. Since $|U| < c^2|B(q,r)|$ this implies that we want

$$\frac{2}{3} \geq \frac{|V|}{|V| + c^2|B(q,r)|} \iff 2c^2|B(q,r)| \geq |V|.$$

Thus, we can set R to be the distance between q and the element of q-rank $3c^2|B(q,r)|$. Then with high probability (see below) more than half of the X_i events occur, and more than half of the r_i satisfy $r_i \leq 2r + R \leq 2R$. So the median r' also satisfies $r' \leq 2R$, and therefore $|B(q,r')| \leq |B(q,2R)| \leq c|B(q,R)| \leq 3c^3|B(q,r)|$.

To get a high probability bound, we have to expend a bit of care, because the events X_i are not independent. We can argue as follows. Pick a random order of $|V|$ red balls and $|U|$ blue balls. By a Chernoff bound, with high probability, a $2/3 - \varepsilon$ fraction of the blue balls are directly followed by a red ball (since there are many more red than blue balls). The red balls represent the elements of V in the ordering, and the blue balls the elements of U. Note that the b_i are also in U, so they are represented by a set of $|B(q,r)|$ blue balls.

If we pick this set of blue balls representing the b_i at random, then with high probability, we can expect $2/3 - 2\varepsilon$ of these balls to be directly followed by a red ball, which shows the high probability bound on more than half of the X_i's occurring. $\square$

5.5 Dynamic Maintenance

The above data structure contains not enough information to support the efficient insertion and deletion of nodes. Consider the insertion of a new node. As we will see below, computing the finger lists of the new node is pretty easy by a slight modification of the nearest neighbor search algorithm. However, the new node also has to appear in the finger lists of several previously existing nodes, and there is no efficient way to find these affected nodes. We therefore augment our data structure with additional information that allows for efficient insertion and deletion of nodes.

5.5.1 Computing Finger Lists for New Nodes

The following algorithm outputs all nodes that have to appear in the finger lists of a node i_0. It is a simple modification of the nearest neighbor query algorithm 5.9. Recall that the nearest neighbor algorithm visited all points that were closer to q than any previously seen point (we call these points the "record-breakers"). The record-breakers are exactly the points we would include in finger lists of size $\alpha = 1$.

To compute finger lists for general α, we want to visit all points that are among the α closest points seen so far. We therefore modify the definition of r such that we never miss any element closer than the α-closest element seen so far. We collect all elements to be included in the finger list we are computing in the set F.

Algorithm 5.18 (Compute Finger Lists of Size α for Node i_0)

1. Let $i := i_0 + \alpha$.
2. Let $F := \{s_{i_0+1}, s_{i_0+2}, \ldots, s_{i_0+\alpha}\}$.
3. Repeat:
   (a) Let $r' :=$ α-smallest element of $\{d(s_{i_0}, p) \mid p \in F\}$.
   (b) Let $r := d(s_i, s_{i_0}) + \max(r', d(s_i, s_{i_0}))$.
   (c) If $F(i, r)$ wraps around the starting index i_0 (i.e. $i < i_0$ and all indices in $F(i, r)$ are $\geq i_0$), return F.
   (d) If there is an element s_j in $F(i, r)$ such that $d(s_j, s_{i_0}) < \max(r, d(s_i, s_{i_0}))$ then let j be the first index with this property in $F(i, r)$. Otherwise, let j be the last index in $F(i, r)$.
   (e) If $d(s_j, s_{i_0}) < r$ then let $F := F \cup \{s_j\}$.
   (f) Let $i := j$.

The algorithm does not "skip" any elements that should be in the finger lists of s_{i_0} by the same argument as for the correctness of the nearest neighbor algorithm (cf. the proof of Theorem 5.10). As for the running time, first consider the case where $|B(s_{i_0}, r)| \geq \rho$. Then with probability 1/2 any addition s_j to F has half the s_{i_0}-rank of the α-closest element in F. Thus, the s_{i_0}-rank of the α-closest element of F is expected to halve after adding 2α new elements to F. Taking into account that we might visit all ρ nodes closest to s_{i_0} (since the above probability bound does not hold), this leads to a total running time of $O(\alpha \log n + \rho)$.

5.5.2 An Incremental Construction

The above algorithm is all we need to construct a variant of our nearest neighbor search structure for a given set of points S in time $O(n(\log n + \rho))$. In this variant, the elements are not arranged on a cycle, but just in a linear sequence. All searches then have to start at the beginning of the ordering.

Algorithm 5.19 (Offline Construction of Data Structure)

Input: Growth-restricted point-set S.

1. Order elements of S randomly as $s_1, s_2, \ldots, s_n$.
2. Create empty data structure.
3. For $i := n, n-1, n-2, \ldots, 2, 1$ do:
   (a) Prepend s_i to ordering in data structure.
   (b) Call Algorithm 5.18 on node i to compute finger lists $F(i, r)$.

The above construction can only be used for dynamic maintenance if we assume that elements are inserted in a random order. In the remainder of this section we will consider extensions of the data structure that do not rely on that assumption.

5.5.3 The Complete Data Structure

We will now describe two different ways of augmenting our data structure to allow for efficient insertions and deletions of data items. The first variant is simpler, allows for a cleaner analysis and guarantees high probability results on space usage and update times. Its disadvantage is that it is a Monte Carlo data structure, i.e. with a polynomially small probability, the data structure might get broken during an insertion or deletion, causing it to return incorrect answers. Additionally, it only guarantees insertion and deletion costs of $O(\log^2 n + \rho \log n)$ whp.

The second variant always maintains a correct data structure and has a $O(\log n + \rho)$ insertion and deletion time, but the provable space usage and processing time guarantees only hold in expectation. We call this variant the "Las Vegas" version of our data structure, emphasizing the fact that opposed to the Monte Carlo version, it always returns correct results.

5.5.4 Version 1: Monte Carlo

We augment our data structure by finger lists F^r that "point backwards" along the data structure. More precisely, for every node i and radius $r > 0$, we store

1. the finger lists $F(i, r)$, as defined in Section 5.3.2, and

2. the finger lists $F^r(i,r)$, defined as $F^r(i,r) :=$ indices of the first α points s_j preceeding s_i (i.e. in $s_{i-1}, s_{i-2}, \ldots, s_1, s_n, s_{n-1}, \ldots, s_{i+1}$) with $d(s_i, s_j) \leq r$.

By Lemma 5.11, the total space used per node is $O(\log n)$ whp.

Finding all nodes i that should include a new node in their finger lists $F(i,r)$ and $F^r(i,r)$ is quite easy by the following Lemma. To simplify the statement of the Lemma, let $F(i,r,\ell)$ be the first ℓ elements s_j following s_i (i.e. in $s_{i+1}, s_{i+2}, \ldots, s_n, s_1, \ldots, s_{i-1}$) with $d(s_i, s_j) \leq r$, and define $F^r(i,r,\ell)$ similarly.

Lemma 5.20
If $i \in F(j, r', \alpha)$ for some r' then either

- *s_j is among the ρc^5 elements closest to s_i, or*
- *$j \in F^r(i, d(s_i, s_j), \Theta(\alpha \log n))$.* $\square$

We will prove the Lemma shortly, but let us first see how it implies a maintenance algorithm for our data structure.

Corollary 5.21
The Monte Carlo version of our data structure can be maintained by contacting $O(\log^2 n + \rho)$ nodes whp per insertion and $O(\log^2 n + \rho \log n)$ nodes whp per deletion.

Proof: For an insertion of a new node i_0, we create finger lists $F(i_0, r)$ and $F^r(i_0, r)$ as in Algorithm 5.18 (or just extract them out of the computations below).

To determine all nodes i which have to include the inserted/deleted node i_0 in some of their finger lists $F(i,r)$ or $F^r(i,r)$, we use Lemma 5.20. It states that i_0 can appear in some $F(i,r)$ whp only in two cases. Either i is among the ρc^5 elements closest to i_0. We can determine this set of nodes using the range search algorithm from Section 5.4.2. Or, i appears in a size $\beta := \Theta(\alpha \log n)$ finger list $F(i_0, r, \beta)$ or $F^r(i_0, r, \beta)$ of node i_0. These finger lists can be computed using Algorithm 5.18 by setting the parameter α equal to β.

By Lemma 5.20 this will yield all affected nodes i with high probability, and takes time $O(\log^2 n + \rho)$ to do so.

For an insertion, we simply check for all found nodes whether the new node i should be included in their finger lists. For a deletion, we have to ensure that when node i_0 gets deleted from some finger list, that that finger list does not contain fewer than α elements. As mentioned below in Section 5.6.2, this can be assured implicitly with a Chord-like background process.

Suppose, however, that we want to maintain the affected finger lists explicitely. By Lemma 5.12, only $O(\log n + \rho)$ nodes actually include i_0 in their finger lists. Thus, we have to compute new finger lists only for these elements. Finding the new points after i_0 to be added to the finger lists of each of these elements takes time $O(1)$ in expectation, and $O(\log n)$ whp (cf. Theorem 5.10), yielding the claimed time bound. $\square$

Proof (Lemma 5.20): Let i be some fixed node. The intuition behind the proof is that if j is *not* in $F^r(i, d(s_i, s_j), \Theta(\alpha \log n))$ then j and i are "far apart" in the address space. But that makes it likely that there are many elements between j and i that are closer to s_j than s_i, and thus i does not occur in j's finger lists either.

Let r be a radius with $|B(s_i, r)| > \rho c^5$. We will show that there is a constant δ independent of r and i such that whp all nodes j with $r/2 < d(s_i, s_j) \leq r$ that include i in some finger list $F(j, r', \alpha)$ will be included in $F^r(i, r, \delta\alpha \log n)$.

Since there are only a polynomial number of choices for i and r for which the above statement is different, the claim whp holds independent of i and r, implying the Lemma.

By Lemma 5.4, there is a covering of $B := B(s_i, r) \setminus B(s_i, r/2)$ by a constant number of balls with the following properties.

(i) Each ball in the covering contains a constant fraction of the points in $B(s_i, r)$.

(ii) The points in each balls are closer to each other than to s_i.

Now consider the elements of $F^r(i, r, \delta\alpha \log n)$; they form a random sample of $\delta\alpha \log n$ elements from $B(s_i, r)$. Thus, each ball in the covering whp contains at least α elements of $F^r(i, r, \delta\alpha \log n)$ (by property (i), and choosing δ large enough).

Now consider some element $s_j \in B$. If j is not in $F^r(i, r, \delta\alpha \log n)$, then by property (ii) of the covering, there will whp be α elements closer to s_j than s_i between s_j and s_i in the ordering of the data structure. Thus, s_j will not include s_i in any finger list $F(j, r', \alpha)$. This shows the claim. □

5.5.5 Version 2: Las Vegas

In this variant of the data structure, we extend the basic data structure by storing so-called "query lists" at every node i. Recall that to determine its own finger lists using Algorithm 5.18, a node i_0 has to query several finger lists $F(i, r)$ of other nodes i. For each finger list $F(i, r)$, we define the query list $Q(i, r)$ to be the set of nodes i_0 that queried finger list $F(i, r)$ to compute their own finger lists.

Upon insertion of a new node, the query lists allow us to "work backwards" to figure out which nodes should include the new node in their finger lists.

In summary, for every node i, we store:

1. for every $r > 0$, the finger lists $F(i, r)$, as defined in Section 5.3.2,
2. for every $r > 0$, the query list $Q(i, r)$ of elements j that queried $F(i, r)$ when constructing their finger lists $F(j, r')$ for radii $r' > R(j)$,
3. a radius $R(i)$ such that $c \max(\log n, \rho) \leq |B(s_i, R(i))| \leq 7c \max(\log n, \rho)$, and
4. the size $N(i)$ of the set $B(s_i, R(i))$.

This data structure has the following properties.

(i) The space used per node is $O(\log n + \rho)$ in expectation.

(ii) There are algorithms to insert and delete nodes by contacting an expected number of $O(\log n + \rho)$ nodes.

First, let us see that the space requirement per node is $O(\log n + \rho)$ whp per node. We have seen before (cf. Lemma 5.11) that for each node i the finger lists $F(i, r)$ only take up space $O(\log n)$ whp, and $R(i)$ and $N(i)$ take constant space. So it suffices to consider the query lists $Q(i, r)$.

Lemma 5.22
For every node q, the expected number of nodes p that during their finger list construction encounter an $F(j', r)$ containing q is $O(\log n + \rho)$. $\square$

Obviously, for a node j to actually query a finger list $F(i, r)$ of a node i, the node j first has to encounter i in its finger list construction. Thus, we have the following immediate corollary.

Corollary 5.23
The expected size of $\bigcup_{r>0} Q(i, r)$ is $O(\log n + \rho)$ for every node i. $\square$

Before we prove Lemma 5.22, let us show a useful proposition.

Proposition 5.24
Let S be (c, ρ)-growth-restricted, and $p, q \in S$. If q's p-rank is $r \geq c\rho$, then p's q-rank is between r/c and cr.

Proof: Let r' be p's q-rank. For the lower bound, we have

$$B(p, d(p, q)) \subseteq B(q, 2d(p, q)) \implies r \leq |B(q, 2d(p, q))| \leq c|B(q, d(p, q))| = cr'.$$

By symmetry, we have $r' \leq cr$, yielding the upper bound. $\square$

Proof (Lemma 5.22): The intuition behind the proof is that the further an element is from q, the less likely it is to encounter q in its finger list construction. To obtain a bound on the total number of elements encountering q, we could prove a bound on the probability that the element with q-rank r encounters q, and then sum over all values of r. It turns out that the reverse problem, determining how likely it is that an element p encounters the element with p-rank r during its finger list construction, is easier to analyze. But by Proposition 5.24 the q-rank of p, and the p-rank of q are within a constant factor of each other, so we can use this analysis to prove our result. We present our proof backward to keep our goal better in focus.

Let us say that p "*sees*" q if during the finger list construction of p, any finger list containing q is queried. In the following we will show the following probability bound for some constant $\delta > 0$, and any $r > c^2\rho$:

$$\Pr[p \text{ "sees" element with } p\text{-rank } r] = O\left(\frac{1}{r} + \left(\frac{\rho}{r}\right)^{1+\delta}\right). \tag{5.8}$$

Let us first see that this implies the Lemma. Fix an arbitrary node q. Let X_r be the event that the node with q-rank r "sees" q during its finger list construction. We want to prove $E[\sum_r X_r] = O(\log n + \rho)$, or equivalently $\sum_r E[X_r] = O(\log n + \rho)$. We bound $E[X_r]$ as follows: for $r \leq c^3\rho$, we use $E[X_r] \leq 1$. For $r > c^3\rho$, note that q's p-rank is at least r/c. Thus $E[X_r] = O(\frac{1}{r/c} + (\frac{\rho}{(r/c)})^{1+\delta})$ by equation (5.8). This gives us the following bound:

$$
\begin{aligned}
\sum_{r=1}^{n} E[X_r] &= \sum_{r=1}^{c^3\rho} E[X_r] + \sum_{r=c^3\rho+1}^{n} E[X_r] \\
&\le c^3\rho + O(1) \cdot \sum_{r=c^3\rho+1}^{n} \left(\frac{1}{r/c} + \left(\frac{\rho}{r/c}\right)^{1+\delta} \right) \\
&\le c^3\rho + O(1) \cdot \sum_{r=1}^{n} \left(\frac{1}{r} + \left(\frac{\rho}{r}\right)^{1+\delta} \right) \\
&\le c^3\rho + O(1) \cdot \left(\log n + \frac{\rho^{1+\delta}}{n^\delta} \right) = O(\log n + \rho).
\end{aligned}
$$

So to show the Lemma, it suffices to prove equation (5.8).

We will first reduce (5.8) to the following statement. Let Y_k be the number of elements with p-rank between ρc^k and ρc^{k+1} that p *visits* during its finger list construction. By *visit*, we mean that p actually considers a finger list stored at that node. Then for some constant $\varepsilon < 1$, we can show that

$$E[Y_k] \le O(1) + \rho\varepsilon^k. \tag{5.9}$$

This equation is not hard to show. Recall the analysis of the nearest neighbor algorithm 5.9 (and therefore of Algorithm 5.18) in Theorem 5.10. The p-ranks of the points p visits during its traversal of the data structure perform a reflecting random walk. If the rank is greater than ρ, then if α is large enough, for some probability $P > 1 - 1/c \ge 1/2$, the rank halves, and otherwise increases by at most a constant factor. If the rank is $\le \rho$, it remains below ρ with probability P, and otherwise increases by a constant factor. This can easily be modeled as a Markov process.

We will use some simple facts about Markov processes (or random walks, for that matter). Let us split the random walk into two phases: the first phase is from the beginning of the random walk until we first visit an element with p-rank less than ρ, the second phase is the remainder of the random walk. During the first phase, the expected number of visits to elements with rank between r and rc is $O(1)$ for every $r \ge \rho$.

During the second phase, the reflecting nature of the random walk comes into play. The stationary distribution of the Markov process decays exponentially as ranks increase as ρ, ρc, ρc^2, and so on. So the expected number of points visited in any rank-range ρc^k to ρc^{k+1} is $\rho\varepsilon^k$ for $\varepsilon = \frac{1}{P} - 1 < 1$. Combining these expected values for the two phases of the random walk yields equation (5.9).

We now use (5.9) to prove equation (5.8). When can p see an element q with p-rank r? If $r \ge \rho c^2$, then this can only happen if p visits an element with p-rank greater than r/c^2. This is because for elements with a p-rank smaller than that, the considered finger lists $F(i, r)$ would not include any elements with rank r or higher.

Consider the set U of all points with p-rank $\le r/c^2$. This set occurs in a random order in the data structure. Consider the partition of our data structure into r/c^2 intervals between the elements of U. Our point q is equally likely to be in any of these intervals. But while constructing p's finger lists, in most of these intervals, we will only visit points with a p-rank of less than r/c^2, and therefore not see q. We are now going to bound the expected number of intervals in which p could possibly see q. This will allow us to bound the probability that p sees q.

In the interval beginning with p, i.e. the first interval that gets traversed in the finger list construction, p could see q. After traversing that interval, p will visit the first point of U, since p never passes a point that is closer than any point seen so far.

For all other intervals, if p visits a point with rank greater than r/c^2 in some interval, then q could be seen if it is in the same or one of the following α intervals, because that is as far as any considered finger list could reach. So the number of intervals in which p could see q is bounded by α times the number of times p goes from visiting an element in U to visiting an element not in U. This number is bounded by the total number of elements visited with ranks between r/c^2 and r.

Suppose we have $r = \rho c^k \implies k = \log_c(r/\rho)$. Then by equation (5.9), the number of such visited intervals is at most

$$\begin{aligned} E[Y_k] &= O(\rho\varepsilon^k) = O\left(\frac{r}{c^k}\varepsilon^k\right) = O\left(r\left(\frac{\varepsilon}{c}\right)^k\right) = O\left(rc^{-k(1+\delta)}\right) \\ &= O\left(rc^{-\log_c(r/\rho)(1+\delta)}\right) = O\left(r\left(\frac{\rho}{r}\right)^{1+\delta}\right) = O\left(\frac{\rho^{1+\delta}}{r^\delta}\right), \end{aligned}$$

for $\delta = \log_c(1/\varepsilon)$. Since the probability that q is in any particular interval is c^2/r, we obtain that the probability that p sees q is at most

$$\frac{c^2}{r}\alpha\left(1 + E[Y_k]\right) = O\left(\frac{1}{r} + \left(\frac{\rho}{r}\right)^{1+\delta}\right),$$

where the additional 1 is due to the first interval where the search starts. This shows equation (5.8), and therefore proves the Lemma. □

5.5.6 Las Vegas: Node Insertion

After analyzing the space usage of our data structure, we can now turn to describing how to insert and delete nodes. Let us first consider the algorithm to insert a new node q into the data structure.

Algorithm 5.25 (Insert Node q into the Las Vegas Data Structure)

1. Insert q at a random position in the ordering.
2. Construct q's finger lists using Algorithm 5.18.
3. Find the $7c^2\max(\log n, \rho)$ nodes closest to q, and compute a radius $R(q)$ such that $N(q) = |B(q, R(q))| = 3c\max(\log n, \rho)$.
4. For the $7c^2\max(\log n, \rho)$ nodes s_i closest to q, update $R(i)$ and $N(i)$ as necessary.
5. Follow query pointers backwards to find all nodes that see q in the construction of their finger lists. For these elements and the $7c^2\max(\log n, \rho)$ closest neighbors of q:

   (i) Update their finger lists as necessary to include q.

   (ii) Update the query list entries they caused, as necessary.

We now analyze and further discuss the steps of the above algorithm to show its correctness, and to determine how many nodes have to be contacted to perform an insertion.

Step 2: Constructing q's Finger Lists. By the analysis for Algorithm 5.18 (cf. Theorem 5.10), we have to contact $O(\log n + \rho)$ nodes whp to compute the finger lists.

Step 3: Finding the $7c^2 \max(\log n, \rho)$ Nearest Neighbors. This can be done by contacting $O(\log n + \rho)$ nodes, using the algorithm from Section 5.4.2. It is then trivial to compute a $R(q)$ such that $N(q) = |B(q, R(q))| = 3c \max(\log n, \rho)$.

Step 4: Increase $N(i)$ if Necessary. By Proposition 5.24, any node s_i for which q is in $B(s_i, R(i))$, i.e. q is among the $7c \max(\log n, \rho)$ closest nodes to i, must be among the $7c^2 \max(\log n, \rho)$ nodes closest to q. So this step of the algorithm suffices to maintain the correct values of $R(i)$ and $N(i)$ for all nodes i.

When the addition of q causes some $N(i)$ to increase above $7c \max(\log n, \rho)$, we have to compute a suitable $R(i)$ with $|B(s_i, R(i))| = 3c \max(\log n, \rho)$. But since this can happen only once for every $\Omega(c \max(\log n, \rho))$ insertions, the amortized cost is only $O(1)$ per node.

Thus, step 4 can be carried out by contacting an amortized number of $O(\log n + \rho)$ nodes.

Step 5: Following Query Pointers Backwards. In the next step of the algorithm, we follow query pointers backwards to determine which finger lists have to be updated due to the insertion of q. This is motivated by the observation that an element can include q in its finger lists only if it queried a finger list that should also include q.

By "following query pointers backwards" we mean the following. First, we insert q in the finger lists of its α predecessors. Then we proceed recursively as follows: for every finger list $F(i, r)$ into which we insert q, we also check whether to insert q in the finger lists of the nodes in $Q(i, r)$.

In working backwards, we eventually contact all nodes that "see" q during the construction of their finger lists. Many of these do not actually include q in their finger lists. In these cases, we do not have to follow query-pointers further back.

By Lemma 5.22, the number of nodes contacted in this phase of the algorithm is $O(\log n + \rho)$. Since a node might encounter q several times during its finger list construction, it can receive more than one update message during this step of the algorithm. The following Lemma shows that this number is constant, however.

Definition 5.26
Let q be a node in our data structure, and $F(i, r)$ a finger list of node s_i. We say that $F(i, r)$ overlaps *q iff in the ordering of elements in the data structure, q is between s_i and the last element of $F(i, r)$.*

Lemma 5.27
Let s_i and q be two nodes in the data structure. Then the number of finger lists that overlap q and are queried in the construction of s_i's finger lists is at most $\alpha = O(1)$.

Thus, s_i can "see" q only a constant number of times during the construction of its finger lists.

Proof: When constructing s_i's finger lists, the first time we access a finger list that overlaps q in the ordering, this finger list by definition contains all elements before q that we might include into s_i's finger lists. This number is therefore at most α. If we move to one of these points, then by the next time we access a finger list overlapping q's position, their number has decreased by one. So after seeing q's position at most α times, we will move past it. □

Step 5(i): Updating Finger Lists. As argued above, recursively following query lists will lead us to all nodes that might include q in their finger lists. So this step is straightforward.

Step 5(ii): Updating Query Lists. For any node i we its query path be the sequence of nodes that it visits during its finger list construction. Conversely, this is the set of nodes in whose query lists node i appears.

Due to the insertion q, each node that "sees" q in the construction of its finger lists might now take a different query path than before. So we have to construct as much of the new query path as might be influenced by the insertion of q.

Consider some node i that encounters q in its finger list construction. The insertion of q affects i's query path in at most two places. First, from the point that i encounters q until the next element included in i's finger list. The query path will not be affected after that element, since i's query path would have visited that node in any case. The second case applies only if q is included in i's finger lists. The query path is then affected when the radius r in step 3a of Algorithm 5.18 is equal to $d(s_i, q)$.

Let us now summarize the total number of changes on the query path of a node i. First, consider the case where q is included in i's finger lists. Then both places where the query path changes are from one element in i's finger lists to the next. By the analysis in Theorem 5.10, we know that the expected number of steps between elements in the finger lists is $O(1)$. By Lemma 5.12, only $O(\log n + \rho)$ nodes include q in their finger lists, thus the total expected number of changes is $O(1) \cdot O(\log n + \rho) = O(\log n + \rho)$. (Note that the number in Lemma 5.12 is not dependent on any random choices, so we can just multiply the expected number of changes per node with it.)

Second, consider the case where q is encountered by i, but *not* included in its finger lists. Then the number of steps from q to the next element in i's finger lists need not be constant, but depends on the rank of q and the rank of the element determining the current search radius in step 3a of algorithm 5.18. Suppose the rank determining the search radius is r, and q's rank is rc^k. Then the expected number of steps from q to the next element in the finger lists is $O(k)$. But on the other hand, the probability that i encounters an element of rank rc^k while looking for an element of rank r, is only $O(\varepsilon^k)$ for some constant $\varepsilon < 1$ (this can be argued similar as in the proof of Lemma 5.22). Thus the expected number of steps after seeing q is $O(\sum_k k\varepsilon^k) = O(1)$, conditioned on i actually encountering q. The total expected number of changes for all elements i encountering q is therefore $O(\log n + \rho)$, by Lemma 5.22.

In summary, we have to contact $O(\log n + \rho)$ nodes in expectation to perform the insertion procedure of Algorithm 5.25.

5.5.7 Las Vegas: Node deletion

Deleting a node can be done similar to the insertion of Algorithm 5.25. We note the changes in the algorithm. Steps 1 and 2 can obviously be skipped. In step 3, it suffices to compute the $7c^2 \max(\log n, \rho)$ closest nodes, which can be done with Algorithm 5.15.

Step 4: Updating $R(i)$ and $N(i)$. In Step 4, if q gets deleted from some ball $B(s_i, R(i))$, we have to update $N(i)$. As with an insertion, we have to recompute $R(i)$ if $N(i)$ falls below $c \max(\log n, \rho)$. But the amortized cost of this recomputation is $O(1)$ per node.

Step 5: Updating Finger Lists and Query Lists. Again, finding the nodes that encounter q in their finger list construction can be performed in $O(\log n + \rho)$ steps as for the insertion. Deleting q from the finger lists of an element i requires to add one or more other elements to i's finger lists. Also, q's disappearance can change the query paths of all elements that encountered it in their finger list construction.

As with the insertion, one can show that this causes $O(1)$ changes in expectation for each of the $O(\log n + \rho)$ nodes that might encounter q.

Thus, in summary the affected number of nodes is the same as for an insertion, and we have proved the following statement.

Lemma 5.28
The Las Vegas version of our data structure can be maintained by contacting $O(\log n + \rho)$ nodes in expectation for each insertion or deletion of a node. □

5.6 Implementation in Chord

The above data structure can easily be implemented for Chord. Clearly, the finger lists for every point can be stored with the node located at that point. Also, Chord already provides a random order of the points by their assignment to the address space.

5.6.1 Queries

Finding Nearest Neighbors. To find a node closest to a given point in the metric space (an application useful for sensor or mobile networks), we can just query the data structure in the obvious way. A nice feature of the data structure is that the query can be started at any node of the network, leading to a uniform load distribution. Since the data structure depends on the physical locations of the nodes, queries that are unequally balanced in the metric space can also lead to an unequal load balance in the network, since nodes "close" to the query points are more likely to be involved in resolving a query than faraway nodes.

Finding Closest Copies of Data Items. In a second application, the data structure can be used to find closest copies of redundantly stored data items. For this, a query consists of both a point in the metric space, and the key of an item (or its hash h which determines the item's position in Chord's address space). Recall that the nodes storing the copies of an item form a continuous segment in the Chord address space, beginning at the hash value h of the item's key.

To search for the copy of an item that is closest to some position q, we first find the primary node responsible for the item using Chord's built-in lookup mechanism. Then we start our nearest neighbor search for q at this primary node, and continue the search as long as the current node s_i still has a copy of the data item. The closest node to q seen so far is the node with the closest copy. This is true because our nearest neighbor search algorithm never misses a node that is closer to q than any previously seen node. The total running time of the lookup therefore is $O(\log n)$ for the lookup of the primary node, and $O(\log n + \rho)$ for the nearest neighbor search, for a total of $O(\log n + \rho)$.

This approach has the unfortunate side-effect of first having to locate the primary node, which might be far away even though there are close copies of the data item. This can be avoided if we know in advance the address of the *last* node s_ℓ holding a copy of the data item. So for example, if Chord replicated data items forward instead of backwards in the address space, the primary node would be the last node. If we know the last node s_ℓ, we can modify the nearest neighbor algorithm 5.9 as follows to never "overshoot" s_ℓ while finding the nearest copy.

In step 4b of the algorithm, if $F(i, r)$ contains no elements between s_i and s_ℓ in the ordering, we can stop the search if we have already located a copy of the data item, because it will be the closest copy. If we have not yet seen a copy, then we increase r until $F(i, r)$ does contain some item between s_i and s_ℓ (inclusive) in the ordering. In the following steps of the algorithm, we will only consider the items in $F(i, r)$ before s_ℓ.

Range Searches. When implementing the range queries from Section 5.4, one should note the following.

The range query asking for all elements within a radius R of a point q has the disadvantage that the number of returned elements is not bounded, which could result in prohibitively expensive network traffic if that number is large. A simple fix for this could be to count the number of returned elements, and cancel the query after returning, say, $\Theta(\log n)$ nodes. Due to the random ordering of elements in the address space, the returned set will then be a random sample of size $\Theta(\log n)$ from among the elements within radius R of q, depending only on the point that the query started at. For many applications, like finding "some set" of sensors close-by to a location, this solution seems sufficient, in the sense than one does not really need an exhaustive list of all nodes within the given radius.

We mentioned in Section 5.4.2 that to efficiently implement range queries, every node i has to store a radius $R(i)$ such that $c \max(\log n, \rho) \leq |B(s_i, R(i))| \leq 7c \max(\log n, \rho)$. If suffices to recompute $R(i)$ such that $|B(s_i, R(i))| = 3c \max(\log n, \rho)$ once every half-life of the P2P system. Then a simple Chernoff bound shows that whp within a half-life, $|B(s_i, R(i))|$ will stay in the required range, i.e. the value will grow or decrease by at most a factor of $(2 + \varepsilon)$.

5.6.2 Maintaining Data Structure Integrity

In practical P2P networks, one cannot expect every node leaving the network to invoke a deletion procedure. This is because nodes might run out of power, or crash, or their network connection might break, all without enough forewarning to inform other nodes about this. As mentioned above, Chord does not even have a deletion mechanism, since the network has to function anyway if nodes do not invoke it.

Dynamic Maintenance. For the implementation in a P2P system, the elaborate insertion and deletion mechanisms of Section 5.5 are not appropriate. Instead, we increase the size of all finger lists from α to 2α. Each node then recomputes its finger lists using

Algorithm 5.18 whenever they contain fewer than α alive nodes, or once per half-life of the system (whatever occurs earlier). This can be determined by repeatedly checking the status of all nodes in the finger lists, assuming that there is some reliable mechanism (e.g. timeouts) for discovering whether a node is dead.

If one assumes that failures are random, it will take a failure of a constant fraction of all nodes to require a constant fraction of all nodes to recompute their finger lists, i.e. a node expects to recompute its finger lists only $O(1)$ times per half-life of the system. This cost is in the same order of magnitude as the maintenance protocols that have to be run to keep the Chord routing infrastructure functioning, and can therefore be considered efficient.

However, it is not unrealistic to assume that failures can be localized in the metric space, causing nodes to fail depending on their position in the metric space. In this case, nodes close to the failure spots are more likely to have to update their finger lists. We could think of a "localized half-life", i.e. the time it takes half the nodes to change in some neighborhood of a node. This would provide for a better characterization of when updates are necessary to maintain the effectiveness of the data structure.

Let us now see how to insert a node into the network. The newly inserted node notifies its α predecessors of its presence, and this information will percolate backwards as nodes recompute their finger lists. Similar to the analysis of the query-pointers in Section 5.5.6, it can be shown that $O(\log n)$ rounds of finger list recomputations by all nodes will mean that all nodes that should include the new node in their finger lists will have done so.

We will now see that running the finger list recomputations as stated is sufficient to guarantee successful operation of our protocols. That is, we will show that the nearest neighbor algorithm 5.9 (and similarly, the range search algorithms) will whp operate correctly and use $O(\log n + \rho)$ hops.

Correctness. First, note that the query protocol will always return the correct answer if not all nodes in a finger list $F(i, r)$ are dead, since it cannot possibly pass over the nearest neighbor.

Even in the case when we do not recompute finger lists frequently enough, the algorithm can be modified to increase r if all elements in $F(i, r)$ are dead. We will still have correctness if some $F(i, r')$ with $r' > r$ contains an alive node. For routing to work, Chord always maintains the successor of a node. Thus, we will know about an alive node that should be in $F(i, \infty)$, and our data structure will return correct answers as long as the Chord routing infrastructure does not break – clearly the best result we could have hoped for.

Efficiency. The fact that queries still run in $O(\log n + \rho)$ hops is immediate, since in our proof we only required that each finger list contains at least α nodes, which is maintained by the above implementation.

Open Problems. We note that it might be possible to still have fault-tolerance if finger lists get recomputed only once per half-life of the system (and not necessarily if finger lists contain fewer than α alive nodes). Clearly, the protocol still remains correct. However, the performance might deteriorate, as suggested by the following. Whenever a finger list $F(i, r)$ contains only dead nodes, we might increase r to move to some alive node s_j after s_i. But since s_j is likely to come before the dead nodes of $F(i, r)$ in the ordering of the data structure, s_j is likely to also include these dead nodes in its own finger lists. Thus, one would have to show that the search quickly "recovers" from an empty finger list. We leave this as an interesting open problem.

5.7 The Offline Data Structure

So far, we were interested in the performance of our data structure in a peer-to-peer network. As is customary, our running time analysis ignored the computation at each node, and simply counted the number of nodes that had to be contacted in order to resolve a query. In a setting where communication costs dominate local computational costs, this is clearly a sensible simplification.

However, as mentioned in the introduction, our data structure has other applications (such as in machine learning) where the whole data structure is stored on a single machine. In this case, there is obviously no distinction between local and distributed computation, and we want to minimize the actual number of operations performed in a query evaluation. Let us first consider the evaluation of a nearest neighbor query.

Suppose each node i stores its finger lists $F(i,r)$ in a linear list, ordered by r. To use only $O(\alpha \log n)$ storage, we store only distinct finger lists, i.e. if $F(i,r) = F(i,r')$ for $r < r'$, we store only $F(i,r)$. To look up a finger list $F(i,r)$, we can perform a binary search on this list of finger lists, and return the finger list $F(i,r')$ with r' maximal satisfying $r' \leq r$. The binary search takes $O(\log\log n)$ operations and dominates the constant $O(\alpha)$ time spent on the finger list itself, assuming that distance-computations $d(s_i,q)$ are atomic, i.e. take constant time.

Since the computation per node is $O(\log\log n)$, we can give bounds on the total number of operations performed by multiplying our previous results by that factor. For example, we obtain a $O((\log n + \rho)\log\log n)$ nearest neighbor search time. Corresponding results hold for range search, insertions and deletions of elements.

But we can do better. The $O(\log\log n)$ lookup time per node can be reduced to $O(1)$ by exploiting the locality in the reference to finger lists $F(i,r)$, similar to fractional cascading (see e.g. [dBSvKO00]). For the following, assume that we store the finger lists $F(i,r)$ in a sorted list, omitting duplicates (i.e. store each finger list only once), as described previously. Furthermore, our data structure contains the following information.

- For each $F(i,r)$, we store a pointer from $F(i,r)$ to $F(i,2r)$.
- For each $j \in F(i,r)$, we store a pointer from $F(i,r)$ to $F(j,r)$.
- We store radii $R(i)$ satisfying $c\max(\log n, \rho) \leq |B(s_i, R(i))| \leq 7c\max(\rho, \log n)$.
- We store the number $N(i)$ of elements in $B(s_i, R(i))$.
- In the Las Vegas data structure, for each $j \in Q(i,r)$, we store a pointer from $Q(i,r)$ to the list $F(j,r')$ that was the reason for the query.

These changes are only practical for an offline data structure, maintaining pointers into the memory locations of other nodes in a P2P network would be poor engineering, and introduce all kinds of possible faults.

In the following, we will sketch the necessary changes to the algorithms to obtain better offline performance.

5.7.1 Nearest Neighbor Queries

The most fundamental operation in our data structure is the query for nearest neighbors (Algorithm 5.9), in particular since it also forms the basis for range searches and the insertion and deletion of nodes.

We have to change Algorithm 5.9 slightly. Recall the algorithm visits a sequence a nodes, following finger lists from one node to the next. There is some correlation between which finger lists get queried at successive points. Consider two points i and j that our algorithm visits in sequence, querying finger lists $F(i, r)$ and $F(j, r')$. Note that since we store a pointer from $F(i, r)$ to $F(j, r)$, we can quickly find $F(j, r')$ if r and r' are not too far apart.

We distinguish two cases. First, suppose s_j is at least as far from q as s_i. This implies that $r \leq r'$. However, by the way our algorithm works, we have $r' < 3r$. By using the pointers from $F(j, r)$ to $F(j, 2r)$ and $F(j, 4r)$, we can find a suitable finger list, namely $F(j, 4r)$, in two steps. Note that our analysis still works out if we use the radius $4r$ instead of r', as long as we increase the size α of the finger lists by a constant factor.

In the second case, s_j is closer to q than s_i, i.e. $r' < r$. In this case, we start at $F(j, r)$, and up to $2\alpha = O(1)$ times go to the finger list $F(j, r'')$ with next smaller index r''. This might lead us to the correct finger list $F(j, r')$, but then again, we might be left with a value of r'' that is too high. In this latter case, however, with probability at least 1/2, the q-rank of the elements in $F(j, r'')$ is half of the q-rank of s_i (cf. the discussion after Algorithm 5.18). Since this was all we needed in the analysis of Algorithm 5.9, by increasing α by a constant factor (to account for the probability 1/2 of no rank-improvement) the algorithm still steps through only $O(\log n + \rho)$ points, needing $O(1)$ operations per point.

5.7.2 Range Searches

The above modification of the nearest neighbor algorithm immediately gives a range search algorithm that returns all points within a given radius R around a point q. The main change in the algorithm was that r could never drop below R, which only helps for our purposes.

Some more work is required for the case when we want the K nearest neighbors of a point q. Initially, we set $r = R(i)$, where s_i is the nearest neighbor of q (which can be found in time $O(\log n + \rho)$, as argued above). Since we know how to compute $B(q, r)$ in time $O(\log n + \rho + |B(q, r)|)$, it suffices to consider the "Increase r"-algorithm 5.17.

While computing $B(q, r)$, we look at all $F(i, 2r)$ for the nodes i encountered in the computation. Since we have pointers between finger lists of the same radius on different nodes, this takes $O(1)$ time per node. For each $s_i \in B(q, r)$, the first $F(i, r')$ with $r' > 2r$ that is different from $F(i, 2r)$ contains a_i. Thus, we can compute a_i in $O(1)$ time per point in $B(q, r)$. Since the median of the $d(a_i, q)$ can be also be computed in time linear in $|B(q, r)|$, this shows that the whole "Increase r" operation can be done in $O(|B(q, r)|)$ time. Thus, we can find the K nearest neighbors of a point in time $O(\log n + \rho + K)$.

5.7.3 Monte Carlo: Insertion and Deletion

Since the Monte Carlo data structure just required the computation of finger lists forwards and backwards in the data structure, we can carry over the nearest neighbor query analysis to argue that this part of the insertions and deletions takes time $O(\log n + \rho)$. (Note that we always have to maintain the α closest elements seen so far, but since $\alpha = O(1)$, this takes only $O(1)$ time per step.)

To maintain $R(i)$ notice that as for the Las Vegas data structure, we only need to consider the $7c^2 \max(\log n, \rho)$ closest elements to q, which can be computed in time $O(\log n + \rho)$, as seen above. By checking whether $d(q, s_i) \leq R(i)$, we can then add or remove q from $N(i)$ as necessary.

If $N(i)$ drops below $c \max(\log n, \rho)$, or rises above $7c \max(\log n, \rho)$, we recompute $R(i)$ such that $|B(s_i, R(i))| = 3c \max(\log n, \rho)$. This can be done by finding the $3c \max(\log n, \rho)$-th nearest neighbor of s_i in the set of $7c^2 \max(\log n, \rho)$ nearest neighbors of q. This is just requires computing the element of rank $3c \max(\log n, \rho)$ in an ordered set, and can therefore be carried out in linear time $O(\max(\log n, \rho))$. Since it takes at least $\Omega(\max(\log n, \rho))$ insertions and deletions to cause such a recomputation of $R(i)$, the amortized time per insertion or deletion is $O(1)$.

Thus, the running times become amortized $O(\log^2 n + \rho)$ whp per insertion and amortized $O(\log^2 n + \rho \log n)$ whp per deletion.

5.7.4 Las Vegas: Insertion and Deletion

In this data structure, we also maintain $R(i)$ and $N(i)$ as explained above. The running times therefore also become amortized.

For the Las Vegas data structure, we have to show how to maintain the query lists in time $O(1)$ per node. For this note that when following query-pointers backwards, we can do so in $O(1)$ time because we maintain the corresponding pointer to a finger list in our data structure. Maintaining this pointer also only takes $O(1)$ when it is modified, since we know which finger list we are currently accessing.

5.8 Open Problems

In this chapter, we gave new algorithms for location-aware data storage in peer-to-peer networks. Clearly, many challenging problems for making peer-to-peer networks function more efficiently remain. We are going to point out a few examples related to the work presented in this chapter.

Items from metric space: We implemented a nearest neighbor search algorithm if the nodes lie in a metric space. How about a protocol that allows for efficient nearest neighbor queries among a set of items that lie in a metric space? The "ordered items" we considered in Chapter 4 are a "simple" special case of this general domain. A possibility to reduce the general case to orderings are locality-preserving hash functions [IMRV97], but this will usually lead to query times beyond $O(\log n)$. Finding more efficient solutions for interesting classes of metrics is a challenging problem.

Fault-tolerance: As mentioned in Section 5.6.2, it is an open problem whether our search data structure can still yield efficient query rates if the update rate is low, e.g. finger lists get recomputed only once per half-life per node. And if not, what modification could obtain such performance?

Metrics that are almost growth-restricted: We have seen in Section 5.2 that important metrics such as the distribution of hosts on the Internet, or of people in the world, are growth-restricted only in dense areas. Can our data structures be adapted to work efficiently for all points in such a space?

Chapter 6

Models for Massive Data Set Computations

6.1 Introduction

Recently, massive data sets have appeared in an increasing number of application areas. The sheer size of this data, often in the order of terabytes, means that "polynomially computable" is no longer synonymous with "efficiently computable,"; in fact, any problem that requires significantly super-linear computation time is practically impossible to solve on these inputs.

In theoretical computer science, this has led to an interest in computational models for massive data sets. This is because having a realistic computational model is a crucial first step for reasoning about practical algorithms and for defining complexity classes that contain the efficiently solvable problems on massive data sets. This can lead to new, efficient algorithms, or to hardness results showing that a problem cannot be solved within the limitations of existing hardware.

Several such models have been proposed in the literature, the following examples having been particularly influential.

Streaming Computations [MP80, AMS99, HRR99]: In the "streaming model", a machine with small memory is allowed only a small number of linear read-only passes over the input data. Due to its limited computational power, this model can only compute approximate answers for many problems. The model is therefore most appropriate for situations where (a) exact answers are not required, (b) huge amounts of data are produced sequentially, such as log files, or sensor readouts, and (c) there is no desire to store the data even temporarily.

Tape-based external storage algorithms [Knu98]: Data in this model is stored on tapes that can only be read or written sequentially by a machine with small memory. (Instead of speaking of "tapes", we will often just talk about reading and writing "streams"). So it extends the streaming model by being able to write data, and by being able to read and write several streams simultaneously. Motivated by the limitations of early computing hardware in the 1940s-1960s, this model was studied extensively during that time.

External Memory Algorithms [Vit01]: In this model, a machine with small memory has read-write access to a block-based external storage medium, such as a hard-disk.

So as opposed to the previous models, it is possible to access data non-sequentially, but only in blocks of several data items. The motivation of this model are modern data storage devices, such as hard-disks, that store data in blocks, and where most of the access time is due to seeking to the correct position on the medium, after which it takes the same amount of time to retrieve just one item or an entire block.

Clearly, these models give very different answers to the question of what is efficiently computable on massive data sets. This is not surprising, since they were developed at different times, and with different intentions.

In this chapter, we examine the relative computational power of several computational models for massive data set computations, and also try to define a realistic model for computations on today's hardware. Our approach is different from the other chapters, where a single computational model appropriate for the domain already existed. Here, there is a multitude of them, and it is not clear which one of them (if any) is appropriate to model the capabilities of modern computing hardware.

Three facts about modern computing platforms motivate our discussions.

(i) Storage on disk is readily available and cheap (in the order of less than a dollar per gigabyte).

(ii) Sorting a large collection of fixed size records can be done extremely efficiently on commodity hardware, with rates that bottleneck the I/O subsystem (see for instance [Aga96, ADADC$^+$97, Wyl99, Cha02]).

(iii) Sequential disk access rates are comparable to (or in some cases faster than) random access rates in main memory. A modern commodity I/O subsystem (for example RAID controllers) can deliver a 100 MB/s read rate. On a 1+ GHz Pentium PC, random access of 2GB of data in 32 byte chunks from main memory takes approximately 26 seconds – an effective rate of only 80 MB/s [Raj02].

The first point implies that large amounts of temporary storage can, and should, be used. In particular, this means that the classical streaming model may be too restrictive by not allowing data storage at intermediate stages of the computation.

The second point contrasts with the fact that sorting is formally *hard* in the streaming model. This seems to suggest that the streaming model is overly pessimistic in modeling the capabilities of today's computing platforms. If we assume that sorting is possible, i.e. we enhance the classical streaming model by providing a "sort box", then clearly the class of problems solvable by this "streaming and sorting model" is strictly larger than that solvable in the classical streaming model.

The third point shows that for sequential access to external storage, not only does the seek time become irrelevant, but the bottleneck becomes the processor, not the data access. Thus, external memory algorithms that read and write disks sequentially are likely to be very efficient. Sequential access to data also has the advantage of using modern caching architectures optimally, making the algorithm cache-oblivious [FLPR99], and making the block size of the disk irrelevant for performance. The latter is significant because there are cases where the algorithm designer has no influence on the block size, or the block size is unbounded, such as on tapes.

The fact that processing streaming data makes caching unnecessary has also led to its widespread use in computation-intensive areas such as graphics rendering [HHN$^+$02, BH03]. Without the need for caches, which take up a large area on today's microprocessor chips, processors can be manufactured more cheaply, and can achieve much higher data throughput rates. Most recently, graphics card manufacturers have started selling programmable streaming processors, capable of performing a wide range of streaming computations [ATI03, NVI03].

Our Contributions

Motivated by the above discussion, we study four computational models that try to capture the capabilities of modern computing hardware:

1. the streaming model,
2. the streaming model augmented by a "sorting" primitive,
3. tape-based external storage algorithms, as well as an equivalent model called "streaming networks", where functions are computed by a set of networked nodes that exchange data via streams, and
4. external memory algorithms that access their external storage in a mostly sequential way.

We show complexity-theoretic and algorithmic results in this context. First, we discuss the capabilities of the models, showing that their computational power strictly increases in the order they are listed above, except for the third and fourth model, which turn out to be equivalent. The most involved proof is the separation between the streaming and sorting model and tape-based computations. That proof elucidates the key difference between the two models: the ability of tape-based computations to interleave several streams while reading them. If we take away this capability from tape-based computations, they are no more powerful than the streaming and sorting model. The equivalences we show demonstrate that the classes considered here are somewhat "natural", as they can be defined in several, seemingly unrelated ways.

Following the complexity-theoretic discussion, we then demonstrate that streaming and sorting admits efficient solutions to a number of natural problems, such as undirected connectivity, minimum spanning trees, suffix array construction, minimum cut computations, and even some geometric problems. These problems are all known to be hard in the streaming model, suggesting that the addition of a sorting operation, while still giving a "weak" model, extends streaming in a meaningful way.

Related Work

The *streaming model* was defined implicitly in the work of Munro and Paterson [MP80], and even earlier, in the context of algorithms for systems with tape based storage and little memory. The growing interest in massive data set computations has led to numerous publications on this topic in recent years; a comprehensive survey of this area is beyond the scope of this thesis.

Many results and references in the field of *tape-based algorithms* can be found in the third volume of Knuth's "The Art of Computer Programming" [Knu98]. The computational powers of tape-based computations are very similar to the capabilities of graphics hardware for stream processing. These hardware architectures have been studied in the past few years in the graphics community (see [BH03, PBMH02], which also list further references), and have recently also received some interest in the theory community [GKMV03, AKMV03]. Our *streaming networks* model is most similar to the Chromium project [HEB$^+$01, HHN$^+$02] for distributed graphics rendering, where an input stream of OpenGL commands is processed by a network of nodes that communicate via streams.

The *external memory algorithms* (EMA) model was introduced by Aggarwal and Vitter [AV88] (for a recent survey on the topic see [Vit01]). Closely related variants of the *mostly-sequential EMA* model studied in this thesis haven been considered by several researchers [FFM98, FCFM00, BCDFC02] to yield more practically meaningful analyses of algorithms. Abello et al [ABW02] consider a subclass of external memory algorithms that are built up from several primitives, including sorting. While there are some similarities to our models (several algorithms considered in their model can also be implemented in ours), they do not restrict the data access to be mostly sequential, and thus our results (particularly, the lower bounds) are not directly comparable.

Outline of the Chapter

In Section 6.2, we begin by defining the traditional streaming model on which all our other models are based in some sense. We also consider an extension that allows the writing of intermediate streams, and find that this does not greatly improve the model's computational power.

In Section 6.3, we introduce the "streaming and sorting" model, give an example algorithm, and discuss its relation to the traditional streaming model. In Section 6.4, we define "streaming networks", consider their relation to tape-based computations, and show some basic facts about their relationship to the other classes. This is followed in Section 6.5 by the proof that streaming networks are much more powerful than the streaming and sorting model. In Section 6.6, we discuss the computational power of mostly-sequential external memory algorithms, and their relation to the previously considered classes. We continue with the description of algorithms for several problems in the streaming and sorting model in Sections 6.7 and 6.8. We summarize our results and discuss open problems in Section 6.9.

6.2 The Streaming Model

6.2.1 Definitions

The *streaming model* was invented to capture what is computable on massive data sets that can only be accessed in a linear fashion. Examples are data stored on tapes, which can only be read sequentially, or measurements produced by sensors, which are also generated in a sequential fashion. The model can be formalized as follows.

Let Σ be some alphabet. We call a sequence of symbols $\mathcal{S} = x_1 x_2 \dots x_n$, with $x_i \in \Sigma$, a *stream*. Let M be a RAM-machine with a local memory of m bits. A *streaming pass* is the computation performed by M when reading a stream $\mathcal{S}$ sequentially, i.e. first reading x_1, then x_2, x_3, and so on.

We say that a function on streams is computable in one streaming pass if M outputs the value of the function after reading the stream once, i.e. after performing one streaming pass. Likewise, a function is computable in p streaming passes, if it can be computed by M after p consecutive streaming passes. Note that the local memory of M is maintained between successive streaming passes.

This leads to the following definition.

Definition 6.1 (Stream)
Let $p, m : \mathbb{N} \to \mathbb{N}$ be functions on the natural numbers. Then $\textbf{Stream}(p, m)$ *denotes the class of functions that can be computed by a RAM-machine using $m(n)$ bits of local memory and $p(n)$ streaming passes for an input stream of length n.* □

In this definition, we implicitly assume that Σ is a problem-dependent fixed countable set, and there is an injective mapping $\Sigma \to \mathbb{N}$, so that the items can be represented as integers in the memory of the RAM-machine. It turns out that in practice, the choice of Σ and the mapping generally follow naturally from the problem description, so we will usually not make them explicit.

In a slight abuse of notation, we will also use the following abbreviation:

$$\textbf{Stream}(O(f), O(g)) := \{ \textbf{Stream}(p, m) \mid p = O(f), m = O(g) \}.$$

We will use similar notational shortcuts for the other complexity classes studied in this chapter.

The set of complexity classes for which $p(n), m(n) \ll n$ is usually referred to as *the streaming model*, capturing the notion that the size of the input, n, is enormous compared to the machine's memory or the number of passes allowed. Classes that are frequently studied in this context are $\textbf{Stream}(O(1), O(\operatorname{polylog} n))$, and more recently also $\textbf{Stream}(O(1), O(n^{\varepsilon}))$.

Note that our definitions impose no restrictions on the actual amount of computation performed by the RAM-machine. This will be typical for all of our discussions. In this work, we are mostly interested in how restrictions on the way that an algorithm can access its input data (or interact with external storage) influence its computational capabilities. However, we note that for all algorithms given in this chapter, the computational time per stream-element is $O(\operatorname{polylog} n)$ or even constant, whereas our hardness results hold even for an unbounded amount of computation.

It turns out that few functions can be computed exactly in the streaming model. In the past few years, numerous papers have given approximation algorithms for various problems in the streaming model, and proved a significant number of hardness results. It is not hard to see that the streaming model is very pessimistic in modeling the capabilities of modern computing hardware, as discussed in the introduction. Particularly, in the case of multiple streaming passes, it seems unreasonable that one is not able to modify the stream in between passes, given that it has to be stored somewhere anyway in order to be read multiple times.

6.2.2 Writing Streams

The crucial limitation of the streaming model is its inability to modify the stream itself. In practice, it is not unreasonable to assume that if one is able to read a stream, then one should also be able to write a stream at the same time. This newly written stream can then serve as input for later streaming passes. Let us see how this slight modification of the streaming model affects its computational powers.

Consider a RAM-machine that reads a stream sequentially (performs a pass on the stream), and while doing so, outputs a sequence of elements of Σ. We call such a pass on a stream a *streaming-writing-pass* (or *s/w-pass* for short). Using s/w-passes, we can naturally define a computational model that maps streams to streams.

Suppose we are given an input stream $\mathcal{S}_0$. We will produce a sequence of streams $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_p$ as follows. We let our machine perform p s/w-passes, where the input of the i-th pass is $\mathcal{S}_{i-1}$, and the output of the pass is $\mathcal{S}_i$. The output of this computation is the stream $\mathcal{S}_p$.

This leads to a set of natural complexity classes.

Definition 6.2 (W-Stream)
Let $p, m : \mathbb{N} \to \mathbb{N}$ be functions on the natural numbers. Then **W-Stream**(p, m) *denotes the class of functions from streams to streams that can be computed by a RAM-machine using $m(n)$ bits of local memory and $p(n)$ s/w-passes for an input stream of length n, where all intermediate streams $\mathcal{S}_i$ are of length $O(n)$.* $\square$

The restriction on the size of intermediate streams is in keeping with the notion that the input stream is already huge, so that increasing the stream length significantly is not reasonable. Actually, in the **W-Stream** model, this limitation has no significant impact on the computational power, as seen below in Lemma 6.3.

Clearly, we have **Stream**$(p, m) \subseteq$ **W-Stream**(p, m), since the RAM-machine could just output the input stream, such that $\mathcal{S}_i = \mathcal{S}_0$ for all p passes. The new model therefore is at least as powerful as the streaming model. Is it more powerful?

To compare **Stream** and **W-Stream** meaningfully, we have to restrict ourselves to functions that are not stream-valued. Let us therefore consider the subset of **W-Stream** where the output stream consists of a single element. We denote this class by **W-Stream-1**. Unfortunately, it turns out that **W-Stream-1** is not much more powerful than **Stream**, as evidenced by part (iii) of the following Lemma.

Lemma 6.3
The following inclusions hold between **Stream** *and* **W-Stream-1***:*

(i) **W-Stream-1**$(1, m) =$ **Stream**$(1, m)$

(ii) **W-Stream-1**$(p, m) \supseteq$ **Stream**(p, m)

(iii) **W-Stream-1**$(p, m) \subseteq$ **Stream**(p, mp)*, even if the lengths of intermediate streams are not bounded by $O(n)$.*

Proof: Claim (i) is immediate from the definition, and we have already argued part (ii) – in each s/w-pass, the RAM-machine can simply output the input stream.

For part (iii), we show that it is not really necessary to create the intermediate streams explicitely – it suffices to generate their elements whenever they are needed by later passes. A similar technique is traditionally used to show that the composition of **LOGSPACE** computable functions remains in **LOGSPACE**.

Suppose we are given a machine M computing some function in **W-Stream-1**(p, m). Let $\mathbb{M}_i$ be the memory content of machine M at the beginning of pass i ($i = 1, 2, \ldots, p(n)$). We will simulate M in **Stream**(p, mp) as follows.

In the i-th streaming pass ($1 \le i \le p(n)$), we simulate i copies of M in parallel. The j-th copy of M (for $j = 1, \ldots, i$) performs the same operations as M would during its j-th pass. Its memory content starts out as $\mathbb{M}_j$, it reads the stream produced by the $(j-1)$-st copy of M, and writes a stream to be read by the $(j+1)$-st copy of M.

The only "trick" in this simulation is that the intermediate streams, i.e. the outputs of the i machines are never explicitely written. We start out by running the i-th copy of M. When it wants to read an input symbol, we run the $(i-1)$-st copy of M until it produces an output symbol. In general, when the j-th copy wants to read a symbol, we run the $(j-1)$-st copy until it produces an output symbol, unless $j = 1$, in which case we read directly from the input stream.

In this manner, in the i-th pass we simulate the first i passes of machine M, ending up with memory contents corresponding to $\mathbb{M}_2, \ldots, \mathbb{M}_{i+1}$ for the next pass.

The output of the i-th machine is generally ignored, unless $i = p(n)$, in which case we can stop our simulation as soon as this last machine outputs a symbol, because that is the output of the streaming algorithm. $\square$

The above result shows that allowing the use of intermediate streams enhances the capabilities of the streaming model slightly, so that computations can be carried out using only m bits instead of mp bits. But assuming $m, p = O(\operatorname{polylog} n)$, this does not lead to a dramatic gain in computational power.

It turns out that the streaming model is not so much limited by not being able to produce intermediate streams, as it is by not being able to *reorder* a stream. This is evidenced by the following Lemma, which follows directly from the multi-party communication lower bound proved by Bar-Yossef et al ([BYJKS02], Theorem 23, part 1).

Lemma 6.4 (Bar-Yossef, Jayram, Kumar, Sivakumar)
Sorting (or even reversing) a stream is in **W-Stream**(p, m) *only if* $mp \geq n$. $\square$

It therefore seems that a meaningful extension of the streaming model has to allow not only intermediate streams, but also allow for some kind of reordering of the input stream.

6.3 Streaming and Sorting

In this and the following section, we will describe two different ways in which to extend the streaming model. First, we are going to consider the streaming model augmented with a sorting primitive. Second, we are going to consider tape-based computations, where several streams can be read and written concurrently.

Since all these models allow streams to be read and written in every pass, we will refer to "s/w passes" simply as "streaming passes" from now on.

6.3.1 Definitions

As argued in the introduction, sorting a stream is an operation that can be performed efficiently on modern computing hardware. We will now define the notion of a "streaming and sorting" algorithm.

For the sorting operation, we want to restrict the comparison function from begin too complicated. More precisely, a *memory m sorting pass* is a function defined using a RAM-machine M with local memory m that computes a partial order on Σ, i.e. given two items of Σ, it returns which one is greater, or whether they are equal or incomparable. If $\mathcal{S}$ is the input of a sorting pass, then the output $Sort_M(\mathcal{S})$ is $\mathcal{S}$ reordered according to the partial ordering defined by M. Note that the result is not uniquely defined if there are incomparable items in $\mathcal{S}$; in practice, this problem is easy to avoid. Also note that M's computation in the sorting pass is side-effect-free, i.e. no state is maintained between comparisons.

Definition 6.5 (StrSort)
Let $p_{Str}, p_{Sort}, m : \mathbb{N} \to \mathbb{N}$ be three functions on the natural numbers. Then we define $\mathbf{StrSort}(p_{Str}, p_{Sort}, m)$ *to be the class of functions computable by the composition of up to $p_{Str}(n)$ streaming passes and $p_{Sort}(n)$ sorting passes, each with memory $m(n)$ for an input stream of length n. We also*

- *permit that the local memory is maintained between streaming passes, and*
- *require that streams produced at intermediate stages are of length $O(n)$.*

We set $\mathbf{StrSort}(p, m) := \cup_{p'+p'' \leq p} \mathbf{StrSort}(p', p'', m)$. □

Note that $\mathbf{StrSort}(p, 0, m) = \mathbf{W\text{-}Stream}(p, m)$. However, by Lemma 6.4, we know that $\mathbf{StrSort}(0, 1, 0)$ is not contained in any $\mathbf{W\text{-}Stream}(p, m)$ with $mp < n$. In this sense, the class **StrSort** is *polynomially more powerful* than **W-Stream**, since we have to increase the number of passes by a polynomial factor (i.e. by a factor of n) to get a similar computational power. As we will see in Section 6.7, a great number of problems are efficiently solvable in the **StrSort** model, and thus, arguably, solvable in practice. As an example, let us consider a natural, fundamental problem.

6.3.2 Undirected Connectivity

In this section, we give a **StrSort**-algorithm for undirected s-t-connectivity. For this problem we assume that the input stream consists of edges (u, v) of an undirected graph and two distinguished vertices s and t, and the question is whether there is a path from s to t in the graph.

Lemma 6.6
Undirected s-t-connectivity can be solved in randomized $\mathbf{StrSort}(O(\log n), O(\log n))$.

Proof: Our algorithm is very similar to algorithms commonly used in **RNC** to solve this problem, in that it repeatedly contracts edges in the graph (see [KR90], section 2.3.1). We choose this algorithm for simplicity of implementation, and it is an open problem whether the required number of sorting passes could be reduced further using a different algorithm (cf. Section 6.9).

Our algorithm proceeds as follows. We assign a random number, a $3 \log n$-bit integer, to each vertex of the graph. Then each vertex gets labeled by the smallest number among the ones assigned to its neighbors and itself. We then merge all vertices that receive the same label. In expectation, this reduces the number of nodes in the graph by a constant factor. Thus, by repeating this process a logarithmic number of times, we obtain a graph without any edges, where each vertex represents a connected component of the original graph. By keeping track of which intermediate vertices s and t get merged into, we can answer the connectivity query by simply checking whether they end up in the same component.

The remainder of the proof consists of showing that the number of nodes decreases by a constant factor in each contraction phase, and that each such phase can be implemented in $\mathbf{StrSort}(O(1), O(\log n))$.

Correctness. Suppose the number of nodes before a relabeling phase is n. Since with high probability all randomly assigned numbers are distinct, we can assume without loss of generality that the assigned numbers are actually $\{1, 2, \ldots, n\}$, since only their relative order matters for the relabeling. Further, we can assume that there are no isolated nodes in the graph, because if either s or t are mapped to such a node, we can decide the connectivity question immediately, and otherwise these nodes can simply be ignored.

After each round, a node v has a label greater than $n/2$ only if both itself and its neighbors got assigned random numbers greater than $n/2$. Since this only happens with probability $1/2$ per node, and v has at least one neighbor, v gets a label greater than $n/2$ with probability at most $1/4$. Thus, the expected number of nodes that receive a label greater than $n/2$ is only $n/4$. The expected total number of distinct labels in the relabeled graph is therefore at most $\frac{3}{4}n$ (assuming the worst case that all labels less than $n/2$ are actually used). This shows the desired reduction in size.

Implementation. Let us now sketch the implementation of a contraction phase. We assume that our stream is the list $\mathcal{V}$ of the vertices $v_1 v_2 \dots v_n$, followed by the list $\mathcal{E}$ of the vertex-pairs representing the edges. Since the graph is undirected, $\mathcal{E}$ contains both pairs (u, v) and (v, u) for an edge between u and v.

While both $\mathcal{V}$ and $\mathcal{E}$ are part of one stream, we will sometimes state operations on the two halves independently, assuming that in such a pass, the other half of the stream is passed through unchanged.

1. First, in a pass over $\mathcal{V}$, we produce a stream $\mathcal{R}$ that contains pairs of node names v_i and random distinct numbers r_i, e.g. $3 \log n$-bit random numbers:

$$(v_1, r_1)(v_2, r_2)(v_3, r_3) \dots (v_n, r_n)$$

2. Now we determine the new label for each node, i.e. the lowest value of r_i among its neighbors and itself.

   (a) First, we sort the pair of streams $\mathcal{R}$ and $\mathcal{E}$, so that the pairs (v_i, r_i) are directly followed by all edges whose first component is v_i:

$$(v_1, r_1)(v_1, \_)(v_1, \_) \dots (v_2, r_2)(v_2, \_) \dots$$

   In one pass on this stream, we can produce a new stream of edges $\mathcal{E}'$, where for the first component of each edge, v_i is replaced by the corresponding r_i:

$$(r_1, \_)(r_1, \_) \dots (r_2, \_) \dots$$

   (b) Now we sort $\mathcal{R}$ and $\mathcal{E}'$, so that the pairs (v_i, r_i) appear right before all edges whose second component is equal to v_i:

$$(v_1, r_1)(\_, v_1)(\_, v_1) \dots (v_2, r_2)(\_, v_2) \dots$$

   In this stream, the number r_i assigned to a node v_i occurs right before the list of numbers assigned to the nodes incident to v_i. Thus, in one linear pass, we can determine the smallest number among them for each v_i, which yields a stream $\mathcal{L}$ of nodes v_i with their new labels ℓ_i:

$$(v_1, \ell_1)(v_2, \ell_2)(v_3, \ell_3) \dots (v_n, \ell_n)$$

3. Now that we know the correct labels, all that remains is to relabel the edges in $\mathcal{E}$. This can be done by repeating step 2(a) for both the first and second component of $\mathcal{E}$, using $\mathcal{L}$ instead of $\mathcal{R}$. While producing this new stream, we can also eliminate all edges of the form (ℓ, ℓ), yielding a new stream of edges $\mathcal{E}_{\text{new}}$.

   The new list of vertices $\mathcal{V}_{\text{new}}$ can be obtained from $\mathcal{L}$ by outputting only the second component ℓ_i of each pair, sorting the result, and removing duplicates. □

6.3.3 Simulation of Circuits

The algorithm given above is similar to an **RNC** algorithm for the same problem. This is not a coincidence, since it turns out that in our computational model, it is quite straightforward to evaluate uniform, linear-width, poly-logarithmic depth circuits. Since problems in **NC** (or **RNC**) that require only a linear number of processors can be solved by such circuits, this gives us a systematic way of constructing streaming algorithms for these problems. Examples of such problems are the undirected connectivity algorithm given above, and the computation of a maximal independent set [Lub85] (see also Section 6.7.3).

Lemma 6.7
In $\mathbf{StrSort}(d, d, O(\log n))$*, one can evaluate uniform bounded fan-in circuits that for* n *inputs have width* $O(n)$ *and depth* $d(n)$*.*

Proof Sketch: The proof is similar to the PRAM simulations in the external memory model [CGG$^+$95, ABW02]. To evaluate a circuit, we inductively generate for each level ℓ of the circuit a stream $\mathcal{S}_\ell$ that contains a list of the inputs taken by the circuit nodes on that level, ordered by node. (Note that $\mathcal{S}_1$ can easily be computed from the input.) One streaming pass on $\mathcal{S}_\ell$ can compute the outputs of all nodes on level ℓ. To go from these outputs to $\mathcal{S}_{\ell+1}$, all we have to do is rearrange them according to the input pattern of the next level. This can be done by labeling the outputs with the numbers of the gates that take them as inputs (and creating duplicates if an output is read by multiple gates). Sorting on these labels yields the desired order. □

Note that the inclusion in Lemma 6.7 goes only one way, since all computational steps of a streaming algorithm could linearly depend on each other, so it is not clear how to perform them with a circuit of sub-linear depth. In fact even the traditional streaming model is not contained in **NC** (relative to an oracle), as the following Lemma shows.

Lemma 6.8
Relative to an oracle computing the inverse of a one-way hash-function, there is a function that is not in **NC***, but can be computed in* $\mathbf{Stream}(1, O(\log n))$*.*

Proof: Let Σ be some universe of items, and $h : \Sigma \times \Sigma \to \Sigma$ be a hash-function on pairs of elements of Σ. Given a sequence $a_1, a_2, \ldots, a_n$ of items from Σ, we define a sequence of hash values $(h_i)_{1 \le i \le n}$ as follows:

$$\begin{aligned} h_1 &:= h(a_1, a_1) \\ h_i &:= h(a_i, h_{i-1}) \quad (i \ge 2) \end{aligned}$$

Now consider the function that maps a sequence $a_1, \ldots, a_n$ to the corresponding h_n. We call this the "chain-hash" of the sequence. Obviously this function is computable in a single streaming pass with constant memory, given access to an oracle computing h.

For the following let us assume that h is given by an oracle. If h has no discernible structure, then the chain-hash of a sequence of items cannot be computed in **NC**.

This is not hard to see. Suppose the hash-function h is chosen adversarially such that each of the h_i occurring in the chain-hash computation has only one pre-image, namely the pair (a_i, h_{i-1}) (or (a_1, a_1) in the case of h_1). Thus, as h cannot effectively be composed with itself, an algorithm computing h_n first has to compute h_{n-1}, and in general, the only way to compute h_i is to first compute h_{i-1}. This, however, implies that an **NC** algorithm can compute only one h_i per time step, and therefore would need n time steps to compute h_n, a contradiction to the fact that an algorithm in **NC** can only take a poly-logarithmic number of steps. □

6.3.4 The Power of Sorting

Finishing our complexity-theoretic discussion of the class **StrSort**, we will now show that for every k, there is a problem that requires at least k sorting passes to be efficiently solved in the streaming and sorting model (Lemma 6.4 implies this for $k = 1$).

Lemma 6.9
Suppose we have an oracle computing the inverse of a one-way hash-function. Then for every $k \geq 1$ there is a problem P_k that can be solved in $\mathbf{StrSort}(k, k, O(1))$, *yet any algorithm that uses only $k-1$ sorting passes, and p streaming passes with memory m must satisfy $mp \geq n$.*

Proof: We define the problems P_k inductively. Let P_1 be the problem of computing the chain-hash (cf. the proof of Lemma 6.8) of the sorted input stream (we assume that there is a natural order on the universe Σ of data items). If x is the output of problem P_{k-1} $(k > 1)$ on some stream $(a_1 a_2 \ldots a_n)$, then P_k is the output of P_1 on the stream $(h(a_1, x)h(a_2, x) \ldots h(a_n, x))$. Clearly, P_k can be solved using k sorting passes (to bring the items into sorted order), alternated with k streaming passes (to compute the chain-hashes).

To prove the claimed lower bound, we follow the reasoning from the proof of Lemma 6.8. In particular, since h cannot be effectively composed with itself, the only way to generate the items in the chain-hash computations is via sequential application of the hash function h. Thus, the only way to arrive at the intermediate values of the chain-hashes in P_k (and the output of P_k) is via the sequence of hashes used in the definition of P_k.

This implies that in order to solve P_k, one first has to compute the answer to P_{k-1} since otherwise one does not have access to the elements of Σ whose chain-hash defines P_k. So to show that P_k requires k sorting passes, it suffices to show that P_1 requires one sorting pass.

In order to compute P_1, the algorithm has to process all elements of our input stream in sorted order. So suppose we have an algorithm that solves P_1 (effectively sorts the input stream) using m memory and p streaming passes, but no sorting passes (i.e. by an algorithm $\mathbf{StrSort}(p, 0, m)$). As pointed out in Section 6.3.1, this implies that we can sort in $\mathbf{W\text{-}Stream}(p, m)$. Lemma 6.4 then implies that we must have $mp \geq n$, as claimed. □

6.4 Streaming Networks

6.4.1 Definitions

We now define our second extension of the traditional streaming model. This model subsumes the **StrSort** model discussed in the preceeding section, and is motivated by tape-based computations and the stream processing architectures mentioned in the introduction. We call this computational model *streaming networks*.

Definition 6.10 (Streaming Network)
A streaming network *is defined by its topology, a directed acyclic graph $G = (V, E)$, and an integer m. Each node $v \in V$ represents a streaming computation using a local memory of at most m. Each directed edge $(u, v) \in E$ represents a stream produced by node u and consumed by node v.*

We require that G contains a unique node without any incoming edges, that has exactly one outgoing edge, called the source*. The stream leaving the source is the* input *to the network. Also, G has to contain a unique node without an outgoing edge, that has one incoming edge (the* sink*). The stream entering the sink is the* output *of the network.* $\square$

We can use a streaming network to carry out a computation by assigning a stream processing algorithm to every node, except for the source and sink. This network then, in the natural way, maps input streams (leaving the source) to output streams (entering the sink), and we say that it *implements* the corresponding function. We only impose the following two restrictions (see Sections 6.4.2 and 6.4.3 for a discussion of why they are necessary).

- Any set of independent streams (i.e. streams that are neither ancestor nor descendant of each other in G) has a combined length of at most $O(n)$, where n is the length of the input stream, and
- all streams leaving a node are the same (i.e. each node writes a single stream that gets sent to several recipients.).

In order to examine the relationship between the computational power of streaming networks and the **StrSort** model, we need to study how limitations on the network topology and memory resources affect the computational power of a streaming network. The parameters of interest to us are summarized in the following definition.

Definition 6.11 (StrNet)
Let $p, m : \mathbb{N} \to \mathbb{N}$ be functions on the natural numbers, d a positive integer and $a \in \{I, S\}$. Then we let $\mathbf{StrNet}(p, m, d, a)$ be the set of functions that for an input of size n can be computed using a streaming network for which

- *the number $|V| - 2$ of nodes in the network (besides source and sink) is at most $p(n)$,*
- *the memory used per node is $m(n)$,*
- *the in-degree of nodes $v \in V$ is at most d (this limits the number of streams which can be simultaneously "read" by a node), and*
- *the parameter 'a' limits how nodes can access their incoming streams. For* Interleaved *access ($a = I$), nodes can advance their input streams independently of each other, and for* Serialized *access ($a = S$), nodes have to process their input streams in a fixed order, and cannot start reading a new stream until the previous stream has been exhausted.* $\square$

We will use $\mathbf{StrNet}(S)$ and $\mathbf{StrNet}(I)$ to refer to the serialized and interleaved streaming networks models, respectively.

6.4.2 Tape-Based Computations

Streaming networks as defined above are computationally equivalent to the computational model where data is stored on tapes [Knu98]. We will briefly point out how different aspects of the two models correspond to each other.

The nodes of a streaming network correspond to passes where a machine reads several input tapes, and produces one output stream, which is again written to a tape. These passes can be carried out on a single machine in a topologically sorted order (note that G is acyclic). Or the passes can be distributed over several machines in order to parallelize independent streaming passes.

Obviously, streams can be deleted as soon as no other pass requires their reading. The size limitation on independent streams in our definition then states that the storage required for all unprocessed streams is at most $O(n)$ at any time. So if every tape can hold $\Omega(n)$ items, we only need a constant number of tapes to evaluate a streaming network computation.

The distinction between serialized and interleaved access also has a natural interpretation for tape-based computations. In serialized access, a machine reads one tape after another, for example because it has only one tape reader, into which tapes have to be loaded sequentially. For interleaved access, several tapes can be read concurrently, and can be advanced independently of each other in several tape readers.

6.4.3 Network Streams

Given the name "streaming networks", it is also natural to envision each node of G as a separate computer on a network. This interpretation of streaming networks is somewhat less realistic than the tape-based interpretation given previously. The main problem with this interpretation is that the underlying network has to provide $\Theta(n)$-size buffers for streams that have been written but not yet read, which is unlikely to be possible in practice. Ignoring this problem however, the reader may visualize streaming networks in either of the two frameworks.

In this second interpretation, streams are transmitted sequentially across a network, which could for example be the Internet. We view the network as being implemented asynchronously. Nodes read their input streams and produce output streams at unrelated rates. Streams not yet read by the receiving node are buffered in the network, and whenever a node tries to read from an input stream for which the buffer is empty (i.e. the producing node has not yet written the associated data), the node waits until more of the stream is written.

To ensure that there are no dead-locks in this computation, we required that the network *G is acyclic*. Otherwise, it could happen that a node is waiting on an input stream that depends on output that the node has not yet written.

We also required that *any set of independent streams has size $O(n)$*. This ensures that the network has to buffer at most $O(n)$ intermediate data, i.e. only a constant factor more than the original input. Note that this in particular means that the total length of all streams entering a node is at most $O(n)$, since these streams are independent. Without this restriction, an $O(k \log n)$ size network with in-degree $d \geq 2$ can generate a stream that is a concatenation of n^k copies of the input stream, e.g. generated by repeated doubling of the stream. A single pass on this long stream allows n^k random accesses to the input stream by advancing to the next copy with each request. Thus, without this restriction on intermediate stream sizes, we could answer any **LOGSPACE** computable query with a poly-logarithmic size network. This clearly defeats the purpose of "streaming computations" capturing a sequential nature of data access.

Finally, we require that *all output streams of a node are the same*, i.e. that a node in some sense only produces one output stream. Clearly, we could have allowed D distinct output streams per node, and made that another parameter of our class. However, a node with D distinct outputs can easily be replaced by D nodes with one distinct output each. Thus, there is a direct linear tradeoff between the number of distinct output streams and the number of nodes in the network. This makes this quantity seem less interesting than the maximal in-degree d of a node, for which no such simple tradeoffs exist, as we will see in Section 6.5.1.

The analogy between streaming networks and real-world stream processing networks becomes most fragile when it comes to the issue of unbounded buffering. In actual networks, it is more realistic that every node has a buffer capable of holding some (not too large) number of b items. If we require that our streaming networks can still be evaluated under this restriction, their computational power is much diminished, as the following Lemma shows.

Lemma 6.12
Let some function be computable by a streaming network where each node has a buffer of size b. Then the same function can be computed in $\textbf{W-Stream}(1,(b+m)|V|)$.

Proof: Our **W-Stream**-algorithm will just simulate the whole network at once. The memory always holds the local memory of all nodes ($m|V|$) and the current content of the buffers ($b|V|$). At every step of the simulation, we advance the computation of one node, where the order in which we do this is limited only by the fact that a node cannot read from a stream with an empty buffer, or write to a stream whose buffer is full. But since, by assumption, the network is actually capable of computing the function, there must always be a node that is not "stuck", i.e. that can continue its computation. Note that since the input stream is accessed sequentially, one streaming pass is enough to read it in as necessary.
□

6.4.4 StrNet and W-Stream

Streaming networks are a generalization of the traditional streaming model by allowing the production of intermediate streams, just like the class **W-Stream** defined in Section 6.2.2. However, unlike in the class **W-Stream**, a streaming network pass can read more than one input stream at the same time. Several facts about the relationship between the two classes are summarized in the following Lemma.

Lemma 6.13
The following inclusions hold for any functions $p, m : \mathbb{N} \to \mathbb{N}$, integer $d \geq 1$ and $a \in \{I, S\}$:

(a) $\textbf{StrNet}(1,m,d,a) = \textbf{W-Stream}(1,m)$,

(b) $\textbf{StrNet}(p,m,1,a) \subseteq \textbf{W-Stream}(1,(m+1)p)$,

(c) $\textbf{W-Stream}(p,m) \subseteq \textbf{StrNet}(p+1,m,2,S)$,

(d) $\textbf{StrNet}(p,m,d,S) \subseteq \textbf{W-Stream}(d^p,(m+1)p)$. □

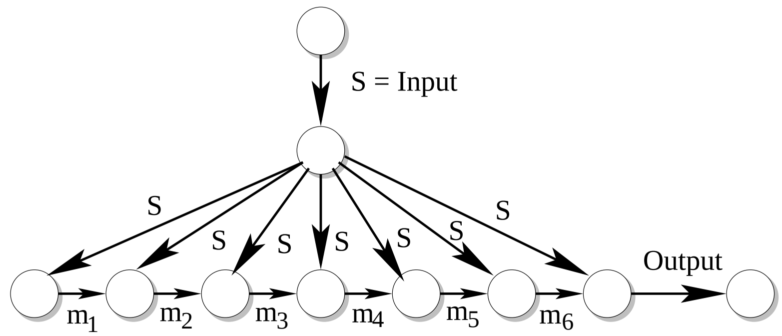


Figure 6-1: Simulating a **W-Stream-1**(p, m)-computation in **StrNet**$(p+1, m, 2, S)$. The first node makes copies of the input stream S, the remaining nodes first read the memory content m_i resulting from the preceeding streaming computation, then read S, and finally write the resulting memory content m_{i+1} as the output stream.

Part (a) states the unsurprising fact that a one-node network is the same thing as a single pass in the **W-Stream** model. Obviously, the in-degree d and access mode a do not matter in this case.

Part (b) shows that streaming networks with in-degree $d = 1$ are relatively weak in computational power – they can be evaluated in a single pass just by using a larger amount of memory. Notice that in the $d = 1$ case, the network is just a linear chain of streaming passes. However, as opposed to the **W-Stream**-model, there is no global memory maintained between passes in the streaming network model.

Obviously, if we modified the definition of streaming networks to maintain the memory between passes, we would have **StrNet**$(p, m, 1, a)$ = **W-Stream**(p, m). While this might be a reasonable modification of the definition of **StrNet**, its main impact would only be felt in the just moderately interesting case of $d = 1$ (for $d \geq 2$, we can pass along the memory content in a separate stream). Moreover, for $d \geq 2$ we then would have to specify in which order the nodes are operating, to pass their memory between each other. These complications led us to adopt the simpler definition of **StrNet** given in Section 6.4.

The fact that two input streams allow for the passing of memory content between nodes leads to part (c) of the above Lemma. Part (d) might be somewhat surprising at first, in particular it states that the computational power of constant size networks is no greater than a constant number of passes in the **W-Stream** model.

Proof (Lemma 6.13): Claim (a) follows from the definition, but let us prove (b), even though it is a special case of (d). The networks in **StrNet**$(p, m, 1, a)$ have in-degree 1, and thus are just linear chains of nodes. Recall the buffering issues we discussed in the previous section. These are obviously not an issue here since the out-degree of every node is 1, i.e. there can be no two nodes reading the same stream at different speeds. Thus, buffering the streams, even with just $b = 1$ does not change the computational powers of the model, so (b) follows directly from Lemma 6.12.

For part (c), we use the "second input stream" of every node to pass memory content between nodes (see Figure 6-1). The first node in our network creates p copies of the input stream, and the remaining p nodes each read such a copy and a stream from their predecessor containing the memory content it had after its pass.

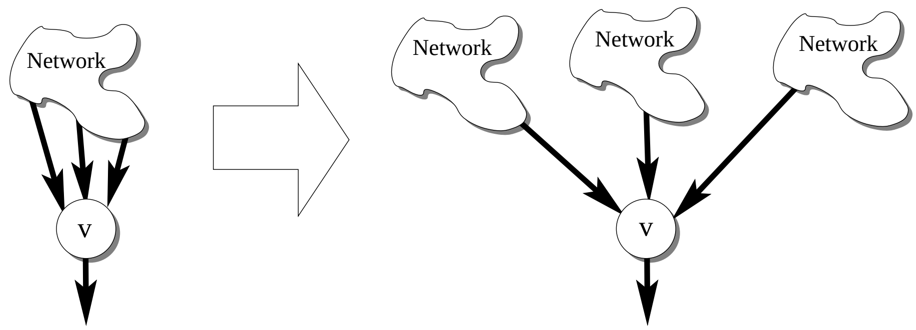


Figure 6-2: Unfolding a streaming network into an in-tree.

To show part (d), we transform our network as follows to make all nodes have out-degree 1. We topologically sort the network, and recursively work backwards from the sink. For each encountered node v, we create d copies of the network preceeding d, one for each input stream to v, and connect them to v in the way shown in Figure 6-2. We then recursively apply the same operation to the d copies of the network.

After applying this operation all the way back to the source, we obtain a network that is a tree on d^p nodes, where every node has out-degree one, and the leaves are d^p copies of the source. Similarly to part (b) we will employ the technique from Lemma 6.12 to simulate the network within the traditional streaming model.

Since our nodes access their input streams in a pre-defined order, this imposes orderings on the edges entering every node, which can be extended in a natural way to an ordering of the leaves. Our simulation can be carried out by considering one leaf-to-root path at a time, in the order of the leaves. It thus requires d^p passes on the input, and we only need to remember the memory and buffer content on the path from the node with the current input stream to the sink. Since that path contains at most p nodes, the simulation can be carried out in a memory of size $m(p+1)$, which shows the claim. Note that the same does *not* hold for functions of streams, since in general, parts of the output stream could be generated during each of the d^p passes on the input stream, and not just on the last pass. $\square$

6.4.5 StrNet and StrSort

We now show that **StrSort** is a equivalent to serialized streaming networks. The equivalence will be exact up to poly-logarithmic factors. These factors are unavoidable since a streaming pass can perform only $O(n)$ work, so we need at least $O(\log n)$ passes in the streaming networks model to sort a stream, while this can be done in a single pass in **StrSort**.

To express our results more succinctly, we define streaming classes where memory and number of passes are limited to be poly-logarithmic in the input length.

Definition 6.14 (PLog-StrSort, PLog-StrNet(a))
We define **PLog-StrSort** *and* **PLog-StrNet**(a) *for* $a \in \{I, S\}$ *as follows:*

$$\textbf{PLog-StrSort} := \textbf{StrSort}(O(\operatorname{polylog} n), O(\operatorname{polylog} n)),$$
$$\textbf{PLog-StrNet}(a) := \textbf{StrNet}(O(\operatorname{polylog} n), O(\operatorname{polylog} n), O(1), a). \ \square$$

We can then begin to characterize the relationship between **StrSort** and **StrNet** by the following Theorem. We will later show that the class **PLog-StrNet**(I) is much more powerful than **PLog-StrSort** and **PLog-StrNet**(S).

Theorem 6.15
We have **PLog-StrSort** $=$ **PLog-StrNet**(S). *More precisely, we have*

(i) $\mathbf{StrSort}(p, m) \subseteq \mathbf{StrNet}(O(p \log n), m, 2, I)$,

(ii) $\mathbf{StrSort}(p, m) \subseteq \mathbf{StrNet}(O(p \log^2 n), \max(O(\log n), m), 2, S)$,

(iii) $\mathbf{StrNet}(p, m, 2, S) \subseteq \mathbf{StrSort}(2p, m)$.

Proof: First note that **PLog-StrSort** $=$ **PLog-StrNet**(S) follows directly from claims (ii) and (iii). They show that a **StrSort**-algorithm can be simulated by a **StrNet**-network with a poly-logarithmic increase in the number of passes, and that a **StrNet**-network can be evaluated by a **StrSort**-algorithm with twice the number of passes.

For claims (i) and (ii), we will show how to sort a stream using streaming networks with interleaved access and serialized access, respectively. Then we can emulate a **StrSort**-algorithm using a streaming network, by replacing every sorting pass by the appropriate sorting streaming network.

For claim (i), we have to show how to sort a stream in $\mathbf{StrNet}(O(\log n), O(1), 2, I)$. This sorting problem has been well-studied in the literature on "external sorting" [Knu98]. We sketch how to implement a variant of Mergesort: view the input stream as a concatenation of length 1 sequences that are already sorted. By concurrently accessing two copies of this stream, neighboring sequence pairs can be merged into single sorted sequences of length 2. Repeating this process yields sorted sequences of length $4, 8, \ldots,$ and a sorted stream after $O(\log n)$ passes, requiring $O(\log n)$ total nodes.

For claim (ii), it suffices to show how to implement a sorting pass in a streaming network with serialized access of size $O(\log^2 n)$ and $O(\log n)$ memory. This is a consequence of the following Lemma.

Lemma 6.16
The sorting of an input tape can be done in $\mathbf{StrNet}(p, m, 2, S)$ *using a poly-logarithmic number of nodes and logarithmic memory. More precisely,*

(i) If the input consists of b-bit integers, it can be sorted with $p = 2b + 2$ nodes, and $m = O(b)$ memory.

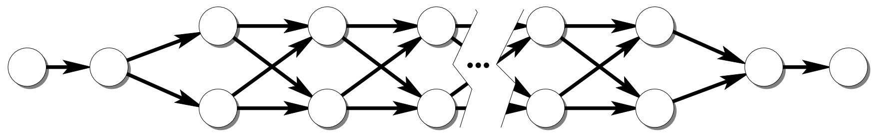
(ii) If only comparison of elements is allowed, then the input can be sorted

  (a) with $p = O(\log n)$ nodes and $m = O(\log n)$ memory using randomization, and

  (b) with $p = O(\log^2 n)$ nodes and $m = O(\log n)$ memory deterministically.

Proof: For part (i), we briefly describe how radix sort can be implemented in our model. For $i = 0, 1, 2, \ldots, b$ we produce a stream S_i where all elements are ordered according to the i least significant bits, so that S_b is the input in totally sorted order. We take S_0 to be the input stream.

To produce S_i from S_{i-1}, we require two nodes. The first node reads S_i, and outputs all elements whose i-th least significant bit is zero, in the order they appear in S_i. The second node similarly writes out all elements whose i-th least significant bit is one. The concatenation of the two produced streams then yields a S_i of the desired form, as can be easily verified inductively. It turns out that we need not concatenate the two half-streams explicitely, but can just send both halves to the next pair of nodes computing S_{i+1}, which do the concatenation internally. This yields the network topology shown in the following figure, containing b layers of node pairs and two additional nodes to duplicate the input stream and merge the two halves of S_b.



Randomized Sorting. For part (ii)(a), we employ a variant of randomized Quicksort. First, we generate a random permutation of the input: in a single pass on the input $a_1 a_2 \ldots a_n$, we write out a stream of pairs (a_i, r_i), where r_i is a random $3 \log n$ bit number. With high probability, all r_i are distinct, so that sorting the stream on the r_i (using radix sort from part (i)), yields the input in a random order.

Now we perform a series of Quicksort type pivoting steps. Initially, we produce a tape with entries (a_i, ℓ_i), where the a_i are the elements to be sorted (in the random order just established), and the ℓ_i are *labels*, initially the empty string for all i. In the following, we maintain the property that the current stream is a permutation of the input, where the elements having the same label appear in a consecutive stretch of the stream.

For each set of elements with the same label ℓ, we use the first element of the group as the pivoting element. A pivoting step itself is done by two nodes. The first node outputs all elements of a set that are less than the pivoting element, indexed with the new label $\ell 0$. The second node outputs all elements greater or equal to the pivoting element, with a new label $\ell 1$. The concatenation of the two streams is the input for the next set of nodes. We repeat the procedure until all labels are unique (or only equal elements have the same label), which by the usual Quicksort analysis takes only $O(\log n)$ passes with high probability, since our pivoting elements were chosen uniformly at random.

After this, the elements are not yet in sorted order, but the lexicographic ordering on the labels corresponds to the ordering of the elements themselves. Thus, by applying radix sort on the labels, we obtain the input in sorted order. The total number of nodes needed is $O(\log n)$.

Deterministic Sorting. Our solution becomes more inefficient when a deterministic algorithm is required. Since we cannot interleave streams, implementing Mergesort is not easily possible.

For part (ii)(b), we therefore derandomize the above algorithm by choosing approximate medians as pivoting elements in each set of elements with the same label. This can be accomplished by a single node and $O(\log n)$ memory using a result by Greenwald and Khanna [GK01]. This node appends an approximate median to each set of elements with the same label. Since we need the pivoting elements to precede their corresponding sets in order to perform the divide operation, we reverse the order of the stream after inserting the approximate medians; this can be done using a sorting pass with $O(\log n)$ nodes. Since $O(\log n)$ Quicksort passes are necessary, this leads to a total number of $O(\log^2 n)$ required nodes. $\square$

We now continue the proof of Theorem 6.15, where it remains to show claim (iii). For this we have to show how a streaming network with serialized access can be evaluated in **StrSort**.

Consider a topological ordering of the streaming network. Let $\mathcal{S}_\ell$ be a concatenation of the streams produced by the first ℓ nodes in the ordering, and consumed by the remaining $p-\ell$ nodes. By definition, the total size of these streams is bounded by $O(n)$. As far as the ordering of the streams in $\mathcal{S}_\ell$ is concerned, we only require that the streams accessed by node $\ell+1$ occur right after each other (in the order they are read by that node).

Our **StrSort** algorithm inductively produces the streams $\mathcal{S}_\ell$, using one streaming and one sorting pass to construct $\mathcal{S}_{\ell+1}$ from $\mathcal{S}_\ell$. Thus, noting that $\mathcal{S}_1$ is just the input stream, we need $2p$ passes to evaluate a p node streaming network. Given $\mathcal{S}_\ell$, we first have one streaming pass that performs the computations of node ℓ on the part of $\mathcal{S}_\ell$ that is the input for that node, and copies the rest of $\mathcal{S}_\ell$ unchanged. By definition, those former streams already occur in the order required by node ℓ. To generate $\mathcal{S}_{\ell+1}$ from this output, we simply apply one sorting pass that rearranges the stream such that the inputs of node $\ell+1$ occur in the right order. This shows the claim. $\square$

6.5 Separating Serialized and Interleaved Access

Due to Theorem 6.15, we know that **PLog-StrSort** and **PLog-StrNet**(S) are the same complexity class. To understand the complexity structure of the classes studied so far, it remains to see how **PLog-StrNet**(I) relates to the other classes.

Clearly, we have **PLog-StrNet**(S) $\subseteq$ **PLog-StrNet**(I), since serialized access is a special case of interleaved access. But does the computational power increase with the ability to interleave streams? And if so, by how much? We will answer these questions in this section. We will state a function problem (Section 6.5.1) and a decision problem (Section 6.5.2) that are easy to solve in **PLog-StrNet**(I), but very hard in **PLog-StrSort** $=$ **PLog-StrNet**(S).

6.5.1 Alternating Sequence

We use a function problem called Alternating Sequence to separate **PLog-StrSort** $=$ **PLog-StrNet**(S) and **PLog-StrNet**(I). It is defined as follows.

Input: A stream of pairs $(a_1, a'_1)(a_2, a'_2)\ldots(a_n, a'_n)$ (b_1, b'_1) $(b_2, b'_2)\ldots(b_n, b'_n)$.

Output: The sequence $a_{i_1} b_{j_1} a_{i_2} b_{j_2} a_{i_3} b_{j_3}\ldots$, satisfying

(i) $i_1 = 1$

(ii) $i_k = \min\{i > i_{k-1} \mid a_i = b'_{j_{k-1}}\}$ for $k \geq 2$

(iii) $j_k = \min\{j > j_{k-1} \mid b_j = a'_{i_k}\}$ for $k \geq 1$, using $j_0 = 0$.

The sequence ends as soon as either $i_k = n$, $j_k = n$ or the minima in equations (ii) or (iii) do not exist.

This problem is best explained by an example. Consider the sequences in Figure 6-3, where the (a_i, a'_i)-pairs are in the top stream, and the (b_j, b'_j)-pairs are in the bottom stream. For this example, the output would start with 1,7,4,17,2,3,1,11,. . . .

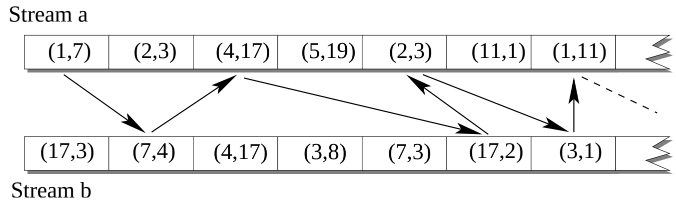


Figure 6-3: An example input for ALTERNATING SEQUENCE.

The ALTERNATING SEQUENCE problem is conceptually easy to solve. Figure 6-3 suggests how the problem can be solved efficiently with interleaved access to two streams. First, one divides the input into two streams, one containing the a-pairs, and the other the b-pairs. Then, we begin by reading a'_1, and scanning the second stream until a matching b_j is found. Then we scan the first stream to find an a_i matching b'_j, and so on, alternating between the two streams. In this sense, the solution makes maximal use of the fact that it can read the two streams independently. We have just shown the following proposition.

Proposition 6.17
The problem ALTERNATING SEQUENCE *can be solved in* $\mathbf{StrNet}(1, O(1), 2, I)$. $\square$

However, solving this problem requires a polynomial number of passes in the **StrSort** and $\mathbf{StrNet}(S)$ models.

Theorem 6.18
The problem ALTERNATING SEQUENCE, *where only comparisons of elements are allowed, can be solved in* $\mathbf{StrSort}(O((n/m)^{1/2}), m)$. *However, it cannot be solved in deterministic* $\mathbf{StrSort}(m, p)$ *unless* $pm = \Omega(n^{1/3})$.

An immediate consequence of Theorem 6.15, Proposition 6.17 and Theorem 6.18 is the following corollary.

Corollary 6.19
$\mathbf{PLog\text{-}StrSort} = \mathbf{PLog\text{-}StrNet}(S) \subsetneq \mathbf{PLog\text{-}StrNet}(I)$. $\square$

Proof (Theorem 6.18): We begin by stating an algorithm in $\mathbf{StrSort}(O((n/m)^{1/2}), m)$ that solves ALTERNATING SEQUENCE. The algorithm proceeds in $2(n/m)^{1/2}$ phases. In the first phase, we construct a stream containing $(n/m)^{1/2}$ copies of the sequence

$$A_1 A_2 \dots A_{\sqrt{nm}} B_1 B_2 \dots B_{\sqrt{nm}}, \tag{6.1}$$

where A_i and B_i are short for (a_i, a'_i) and (b_i, b'_i), respectively (we will continue to use this notation for the remainder of this proof). Clearly, the stream has length $O(n)$, and can be constructed by outputting elements multiple times, indexed by their desired position on the tape, followed by a sorting pass.

We can use this stream to output the beginning of the answer up to the point where either of the indices i_k or j_k becomes greater than $\sqrt{nm}$. This is done as follows. When reading the first sequence of A's, we keep $A_1, A_2, \dots, A_m$ in memory. This enables us to construct the answer up to $i_k \le m, j_k \le \sqrt{nm}$ on the following sequence of B's. When we exhaust the A's in our memory, we continue to the next stretch of A's, and put $A_{m+1}, \dots, A_{2m}$ in memory, use that with the following stretch of B's, and so on.

Since we process m of A-elements for each copy of stream (6.1), after reading the whole stream, the A-elements have been processed up to index $m \cdot (n/m)^{1/2} = \sqrt{nm}$. This shows that we will make a progress of $\sqrt{nm}$ on one of the two indices.

In the second (and later) phases, we repeat the same pattern, but the A- and B-subsequences of length $\sqrt{nm}$ start where we left off in the previous phase. Since one of the indices advances by $\sqrt{nm}$ in each phase, we will have constructed the whole output in at most $2n/\sqrt{nm} = O((n/m)^{1/2})$ phases, which yields the claimed number of passes.

Now for the (harder) lower bound of $p = \Omega(n^{1/3}/m)$ passes for the **StrSort** model. We show this bound by fixing an arbitrary algorithm, and then adversarially choosing its input such that it cannot output the correct solution unless the number of passes meets the claimed bound. In this adversarial model we only decide on the equality of two items when the algorithm compares them.

The proof consists of two main parts. First, we show that in a single pass, we cannot make too much progress towards the solution of the problem. This is because whatever $O(n)$-size input stream we use in that pass, there will be a roughly $\sqrt{n/m}$-length alternating sequence that we cannot output based on linearly scanning the stream.

In the second part of our proof, we apply this construction to a sequence of passes, which slightly weakens the $\sqrt{n/m}$-bound as the algorithm gains more information about the processed data. In the end, it yields the $\Omega(n^{1/3}/m)$ lower bound on the number of passes.

The key lemma for the first half of the proof is the following. It shows that a string that contains all possible Alternating Sequence answers as subsequences must be very long (at least $n^2 + 1$ symbols). The converse of this statement is what we will need: a string of length $O(n)$ can only contain all Alternative Sequence answers for instances of length $O(\sqrt{n})$.

Lemma 6.20
Let S be the set of alternating increasing sequences of the alphabet $\Sigma = \{A_1, \ldots, A_n, B_1, \ldots, B_n\}$, i.e. strings of the form $[A_{i_1}]B_{j_1}A_{i_2}B_{j_2}\ldots A_{i_k}[B_{j_k}]$ where $i_\ell < i_{\ell+1}$ and $j_\ell < j_{\ell+1}$ for $1 \le \ell < k$, and at most the very last symbol is equal to A_n or B_n. By $[\ldots]$ we mean that the symbol is optional. Let $\mathfrak{s}$ be a Σ-string that contains all strings in S as subsequences (i.e. the elements of each $s \in S$ occur in order in $\mathfrak{s}$, but not necessarily consecutively). Then $\mathfrak{s}$ has length at least $n^2 + 1$. $\square$

Although we will not need this later, it is interesting enough to note that the bound in the lemma is actually tight, i.e. there are strings of length n^2+1 of the desired form. Let $\mathfrak{a}_i$ to be the string $A_iA_{i+1}\ldots A_{n-1}$ and $\mathfrak{b}_i$ be the string $B_iB_{i+1}\ldots B_{n-1}$. Then the following string of length n^2+1 has the desired properties:

$$\mathfrak{b}_1\mathfrak{a}_1\mathfrak{b}_1\mathfrak{a}_2\mathfrak{b}_2\mathfrak{a}_3\mathfrak{b}_3\mathfrak{a}_4\mathfrak{b}_4\ldots\mathfrak{a}_{n-1}\mathfrak{b}_{n-1}A_nB_n$$

We leave the details to the reader, and concentrate on the lower bound.

Proof (Lemma 6.20): Fix $\mathfrak{s} \in \Sigma^*$, and let $\mathfrak{s}_i$ be the suffix of $\mathfrak{s}$ that begins at position i (e.g. $\mathfrak{s}_1 = \mathfrak{s}$). We will now define sets of strings S_i such that all strings in S_i have to appear as subsequences in $\mathfrak{s}_i$. We set $S_1 := S$, and define the other S_i inductively. If the i-th symbol of $\mathfrak{s}$ is A_k, then we set

$$S_{i+1} = \{s \mid \text{“}s \text{ does not start with } A_k\text{” and } (s \in S_i \vee A_k s \in S_i)\}.$$

Here "$A_k s$" stands for the string obtained by prepending A_k to s. Replacing A_k with B_k gives the definition in the case that the i-th symbol of $\mathfrak{s}$ is B_k.

It is not hard to see that for each i the strings in S_i necessarily have to be contained as subsequences in $\mathfrak{s}_i$. So to show the desired lower bound on the length of $\mathfrak{s}$, it suffices to show that $S_i \neq \emptyset$ for $i \leq n^2 + 1$.

Let us define predicates $\alpha(i, k, \ell)$ and $\beta(i, k, \ell)$ as "S_i contains a string with the substring $A_k B_\ell$" and "S_i contains a string with the substring $B_\ell A_k$", respectively. If at least one of these predicates is true, it implies that $S_i \neq \emptyset$. We will now show that going from i to $i + 1$, not too many of the predicates change from true to false. The intuition behind this is that some predicates imply other predicates. For example $\alpha(i, k, \ell)$ implies $\beta(i, k', \ell)$ for all $k' > k$, since an alternating sequence which contains $A_k B_\ell$ can be continued as $A_k B_\ell A_{k'}$ for any $k' > k$. These implications limit the number of predicates that can become false. We will show the following.

Claim 6.21
Let k, k', ℓ, ℓ' be numbers between 1 and n with $k \neq k'$ or $\ell \neq \ell'$. If $\alpha(i, k, \ell)$, $\beta(i, k, \ell)$, $\alpha(i, k', \ell')$ and $\beta(i, k', \ell')$ are all true, then at least three of $\alpha(i + 1, k, \ell)$, $\beta(i + 1, k, \ell)$, $\alpha(i + 1, k', \ell')$ and $\beta(i + 1, k', \ell')$ are true.

Proof: Let us consider the case where the i-th element of $\mathfrak{s}$ is an A-element, say A_j (the "B-case" is similar). Then the β-predicates will be unchanged from i to $i + 1$. And the α-predicates can only change if $j = k$ or $j = k'$. In fact, the only way that both these predicates could become false is if $j = k = k'$ holds. This implies $\ell \neq \ell'$, wlog $\ell < \ell'$. But the truth of $\beta(i, k, \ell)$ then implies that S_i and therefore S_{i+1} contain strings that contain $B_\ell A_k B_{\ell'}$, and thus $\alpha(i + 1, k, \ell')$ remains true, which proves the claim. $\square$

Thus, going from i to $i + 1$ there will be at most one $\alpha(\ldots, k, \ell)$, $\beta(\ldots, k, \ell)$ pair for which one of the predicates becomes false, after both having been true so far. Since at the beginning, all n^2 such α, β-pairs are true, it takes at least n^2 elements of $\mathfrak{s}$ to hit all these pairs once. And for the last pair hit, it takes one more character to satisfy the remaining true predicate in it, so $\mathfrak{s}$ has to contain at least $n^2 + 1$ characters. $\square$

The above Lemma shows that is hard to interleave the two streams $A_1 A_2 \ldots A_n$ and $B_1 B_2 \ldots B_n$ into one stream such that all possible answers to Alternating Sequence appear as subsequences in the interleaved stream. For our application of streaming passes, we are however interested in the minimum stream length such that a memory m algorithm could produce all possible answers to Alternating Sequence. This is answered by the following corollary.

Corollary 6.22
Let $\mathfrak{s}$ be a string, such that all alternating sequences of the form stated in Lemma 6.20 can be output by a linear scan of $\mathfrak{s}$ by a machine of memory m. Then the length of $\mathfrak{s}$ is at least $(\frac{n}{m+1})^2 + 1$.

Note that Corollary 6.22 is not tight, consider e.g. the case $m = \Omega(n)$, where the lower bound is just a constant, but clearly each A_i and each B_i has to appear at least once in $\mathfrak{s}$, giving a trivial lower bound of $2n$. But the corollary is still strong enough for our purposes.

Proof (Corollary 6.22): Group the elements of $A_1, A_2, \ldots, A_n$ into $\frac{n}{m+1}$ groups of $m + 1$ consecutive elements each (i.e. $A_1, \ldots, A_{m+1}$ form the first group, $A_{m+2}, \ldots, A_{2m+2}$ the second, and so on), and do the same for the B's. By "interleaving an A-group with a B-group" we mean that we alternate the $m+1$ elements in the A-group in ascending order with the $m+1$ elements of the B-group, for example $A_1B_1A_2B_2\dots A_{m+1}B_{m+1}$ for the first two groups. Now consider only the strings in S that are concatenations of such interleaved groups.

Any memory m algorithm outputting these strings upon reading $\mathfrak{s}$ must for each interleaved group pair read at least one A-element and one B-element from $\mathfrak{s}$. This is because the groups have size $m+1$ each, so they could not possibly have been entirely in memory before outputting the interleaved group pair.

By restricting our view to "representatives of groups", not distinguishing the individual elements of groups, the previous observation implies that $\mathfrak{s}$ must actually be an interleaved string in the sense of Lemma 6.20 on the group representatives. Since there are $\frac{n}{m+1}$ A- and B-groups each, Lemma 6.20 therefore implies that $\mathfrak{s}$ has to have length at least $(\frac{n}{m+1})^2+1$. □

For our adversarial argument, we now fix a particular streaming algorithm. We allow the algorithm to construct construct arbitrary input streams for each of its passes. Even in this more powerful model, we can still prove the lower bound.

To fix the input we will now inductively construct a sequence of numbers $1 = t_1 < t_2 < t_3 < \dots < t_k = n$, and alternating sequences $s_1 \subset s_2 \subset s_3 \subset \dots \subset s_k$ (where by $s \subset s'$ we mean that s is a prefix of s'), such that for all $1 \leq i < k$:

(i) s_k is the correct output to the Alternating Sequence problem,

(ii) all elements of s_i not in s_{i-1} are from the set $\{A_{t_i}, \dots, A_{t_{i+1}-1}, B_{t_i}, \dots, B_{t_{i+1}-1}\}$, and

(iii) s_i is chosen so that, even based on the comparisons the algorithm performed in the first $i-1$ passes, it is not possible for the algorithm to output the elements of $s_i \setminus s_{i-1}$ in order during pass i.

The last point implies in particular that the algorithm cannot output the correct solution within k passes.

For the first step of the inductive construction, we choose t_2 such that $(t_2/m+1)^2+1$ is greater than the length of the stream in pass 1. This means that t_2 can be chosen as $O(m\sqrt{n})$, where the constant depends only on the constant in the $O(n)$ bound we imposed on the maximal stream length. By Corollary 6.22 this implies that there will be an alternating sequence of the A_i and B_i with indices bounded by t_2 that cannot be output by a memory m algorithm upon reading the stream. We let s_1 be one such sequence.

For the inductive step, we have to modify our argument to account for the fact that the algorithm already made some comparisons in the first $i-1$ passes, and our choice of s_i must be consistent with them. This requires us to choose t_{i+1} such that $t_{i+1} - t_i$ is somewhat larger than $O(m\sqrt{n})$.

More precisely, in the previous $i-1$ passes, the algorithm can have performed a total of $O((i-1)nm)$ comparisons – each element in the memory could have been compared to every element in the first $i-1$ streams. We are enforcing the policy that whenever a comparison during pass j involves an element with index greater than t_{j+1}, then we will always return "unequal". Also, we enforce that $a_i \neq a_j$, $a'_i \neq a'_j$, $b_i \neq b_j$ and $b'_i \neq b'_j$ for all $i \neq j$.

For pass i, we therefore want to choose t_{i+1} such that there is an alternating sequence $s_{i+1} \setminus s_i$ with indices between t_i and t_{i+1} not contained in a stream of length $O(n)$, and such that the sequence does *not consecutively contain* any of the $O((i-1)nm)$ pairs for which we already answered "unequal". If one excludes a set of ℓ (A, B)-pairs from appearing consecutively in the strings of S, then a simple modification of the proof of Lemma 6.20 yields a lower bound on $|\mathfrak{s}|$ of $n^2 - \ell + 1$.

So we have to choose t_{i+1} such that $((t_{i+1} - t_i)/(m+1))^2 - nm(i+1) + 1 > O(n)$, which can be satisfied by $t_{i+1} - t_i = \Omega(\sqrt{nm^2 + nm^3(i+1)})$, in particular by $t_{i+1} - t_i = \Omega(\sqrt{nm^3}\sqrt{i+1})$, where the constant only depends on the constant used for the $O(n)$ upper bound on stream lengths.

In summary, we can accomplish the selection of t_i's and s_i's as long as $t_{i+1} - t_i = \Omega(\sqrt{nm^3(i+1)})$ for all i. This gives the following upper bound on k:

$$\sum_{i=1}^{k} \Omega(\sqrt{nm^3 i}) \leq n \implies \sum_{i=1}^{k} \sqrt{i} = O(\sqrt{n/m^3})$$

Since $\sum_{i=1}^{k} \sqrt{i} = O(k^{3/2})$, this means the construction is possible for k up to $(\sqrt{n/m^3})^{2/3} = n^{1/3}/m$, which concludes the proof. $\square$

6.5.2 A Decision Version of Alternating Sequence

A polynomial separation analogous to Theorem 6.18 can also be shown using the following decision problem: the input is just like Alternating Sequence, but now every pair (a_i, a'_i) and (b_j, b'_j) has a *color* which is either red or blue. The desired answer is whether the last element of the output of Alternating Sequence is a red or a blue element.

We briefly sketch how the computation of the Alternating Sequence function can be reduced in **StrSort** to this decision problem. We will do this with a factor $O(\log^2 n)$ increase in the number of required passes. Thus, the hardness of the function implies the hardness of the decision problem.

Suppose we have an algorithm for the decision problem. First, we show how this algorithm allows us to find the last element of the Alternating Sequence output (and not just its color) in $O(\log n)$ streaming passes. By coloring all (a_i, a'_i) pairs red, and all (b_i, b'_i) pairs blue, and invoking our decision algorithm, we know what type the last element is, suppose it is among the a-pairs.

If we color all (a_i, a'_i) with $i < j$ red, and all (a_i, a'_i) with $i \geq j'$ blue (giving the (b_i, b'_i)'s an arbitrary color), then our decision algorithm answers the question "is the index of the last element less than i?". Using binary search, it therefore takes $O(\log n)$ invocations of our algorithm to determine the index of the last element.

This algorithm computing the last element of the Alternating Sequence output can be used to give a divide-and-conquer solution to Alternating Sequence itself. The idea is to find a consecutive pair $(a_i, a'_i)(b_j, b'_j)$ that appears in "the middle" of the Alternating Sequence output. Then we can divide the input into two halves to be solved independently, because we know where to begin with the second half.

To discover these elements in the middle, we consider the first half of the input streams, i.e. $(a_1, a'_1)(a_2, a'_2) \ldots (a_{n/2}, a'_{n/2})$ and $(b_1, b'_1)(b_2, b'_2) \ldots (b_{n/2}, b'_{n/2})$. Given the above algorithm, we can compute the last two elements in the Alternating Sequence output for these half-problems. (The second-to-last element can be computed by first computing the last element, and then deleting it and the part of the stream following it from the input, and again computing the last element.)

This yields our desired divide-and-conquer algorithm for Alternating Sequence.

Note that in each divide step, at least one of the streams gets split exactly in half, showing that only $O(\log n)$ recursions are necessary. Thus, Alternating Sequence can be reduced to the red/blue decision problem, while multiplying the number of passes by $O(\log^2 n)$.

6.6 Linear Access External Memory Algorithms

In this section, we compare the computational power of the previously considered models to another popular model for massive data set computations, "External Memory Algorithms" (EMAs). This model is more general than the other computational models studied so far in this chapter. As opposed to the previous models, it allows a semi-random (as opposed to only sequential) access to data.

6.6.1 Definitions

External memory algorithms study the effect that block-oriented access to external data storage has on the efficiency of algorithms (see [Vit01] for a recent survey). An external memory algorithm can access the external storage (think of it as a hard-disk) only in units of blocks (each containing B items), and performance is measured in terms of the total number of disk accesses (reads and writes). Thus, a good data arrangement scheme and exploitation of locality are necessary for efficient algorithms. The *parallel disk model* introduced by Vitter and Shriver [VS94] has the following parameters:

N = problem size (in data items),

M = internal memory (in data items),

B = block transfer size (in data items), and

D = number of independent disk drives.

Since we are interested in comparing this model to our other computational models, we only care about the single processor case, so we will ignore the sometimes studied additional parameter of the number of processors.

As mentioned in the introduction, reading sequential data items from a disk achieves a throughput similar to main memory reads. Thus, external memory algorithms that only perform sequential reads or writes are a particularly efficient subclass of EMAs. Further properties of these algorithms are the fact that B is irrelevant for the performance of the algorithm, and that the resulting algorithm is cache-oblivious [FLPR99], and in fact uses the cache hierarchy in an optimal fashion. We therefore define a linear access EMA (LEMA) as an EMA with the following parameter:

P = number of out-of-sequence reads/writes on the disks ("passes")

A read or write operation is considered "out-of-sequence" if it does not act on the data block following the block last read or written on the corresponding disk. We are interested in algorithms for which P and M are small, i.e. poly-logarithmic in N, and $D = O(1)$. The class of functions computable within these restrictions is denoted by **PLog-LEMA**.

6.6.2 Relation to Streaming

While exhibiting certain similarities to the streaming models considered earlier (in that data is read and written sequentially), the relationship between these classes is not entirely obvious. It is summarized by the following Lemma.

Lemma 6.23

(a) **PLog-StrSort** $\subsetneq$ **PLog-LEMA**. *More precisely, any problem in* **StrSort**(p, m) *can be solved by an LEMA with* $D = 4$ *disks, memory* $M = m$ *and* $P = O(p \log n)$ *passes.*

(b) **PLog-LEMA** $=$ **PLog-StrNet**(I). *More precisely, an LEMA with* $D = O(1)$ *can be evaluated in* **StrNet**$(O(P), M, D, I)$*, and a* **StrNet**(p, m, d, I) *network can be evaluated by an LEMA with* $M = m$, $P = O(p)$, $D = d + 2$.

Proof: Part (a) follows from part (b) and Theorem 6.15 (the inequality being demonstrated by Alternating Sequence, cf. Section 6.5.1). So it suffices to prove part (b).

"$\subseteq$": The evaluation of an LEMA in **PLog-StrNet**(I) proceeds in a straightforward manner. The network consists of P phases. The D streams entering a phase represent the content of the D disks at that point (note that we need D constant such that the total size of active streams is $O(n)$). These D streams get replicated and fed to D nodes that each read the streams concurrently to produce the content of one of the D disks at the point of the next out-of-sequence read/write. These streams then go to the next phase.

"$\supseteq$": For the other direction, sort the p-node streaming network topologically, so that we can talk about the first pass, second pass, and so on. Let $\mathcal{S}_i$ be the set of streams (including the input stream) created before the i-th pass, and consumed during or after the i-th pass. Note that since the set $\mathcal{S}_i$ is independent, the streams contained in it take up only $O(n)$ space.

The LEMA operates in p phases. In the i-th phase, it starts out storing the streams of $\mathcal{S}_i$ on one of its disks, and then performs the operation of the i-th node in the network to compute the content of the streams in $\mathcal{S}_{i+1}$. Creating $\mathcal{S}_1$ is simple, since it contains just the input stream. And $\mathcal{S}_{p+1}$ contains the output stream.

To perform the action of the i-th node, the LEMA copies the (up to d) input streams of the node onto different disks; this takes a single pass. Then the output stream of that node of the streaming network is computed by concurrently scanning these d disks. Another pass is sufficient to rearrange data to produce one disk containing all active streams, i.e. $\mathcal{S}_{i+1}$.

Note that the efficiency of this algorithm can be improved by not rearranging all streams of $\mathcal{S}_{i+1}$ onto a single disk, but this would lower the number of passes only by a constant factor. $\square$

6.7 Algorithms

As stated in the introduction, many natural problems are efficiently solvable in the **StrSort**-model. In this section, we will give algorithms in **PLog-StrSort** for several basic problems. Clearly, by Lemma 6.7 many problems from **NC** and **RNC** are in **PLog-StrSort** by a generic simulation argument. We are still giving explicit algorithms for some of these problems, as our **StrSort**-implementations are far more efficient than the generic simulation due to Lemma 6.7.

In general, our algorithms do not have matching lower bounds for the required number of passes, as they often do more *work* that the optimal RAM algorithm. As mentioned in Section 6.9, it therefore is an interesting open problem to provide such lower bounds, or improve the algorithms.

In our presentation, we will also state whether we know of more efficient algorithms for the problems in the $\mathbf{StrNet}(I)$ model, as this might be a desirable way to implement them in practice. By "more efficient", we mean algorithms that replace some sorting passes by fewer than $\Omega(\log n)$ streaming passes.

6.7.1 Frequency Moments and Related Problems

The computation of frequency moments was one of the first problems that sparked the recent interest in the streaming model [AMS99]. Given a stream $\mathcal{S} = x_1 x_2 \dots x_n$, the frequency $f(x)$ of an item $x \in \Sigma$ is defined as $f(x) := |\{i \mid x_i = x\}|$. The k-th frequency moment of a stream is $F^k(\mathcal{S}) := \sum_{x \in \Sigma} f(x)^k$.

We observe that these functions become trivial to compute using just one sorting pass, since the computation of frequencies is easy once identical items are grouped together. Thus, these problems are in $\mathbf{StrSort}(2, O(\log n))$, while in traditional streaming, their exact solution requires linear memory [AMS99].

Other examples for problems that become trivial in the presence of sorting are: computing medians (Equi-Depth Histograms), l_p norms of vectors in the *cash-register* model [Ind00], approximate V-OPT histograms [GKS01], and distinct element counting.

6.7.2 String Problems

String Matching

The standard (sub)string matching problem is the following.

Input: Two strings a and b (with $|a| \geq |b|$) over some alphabet Σ'.

Output: Is b a substring of a?

We assume that the input to the problem is a single stream that contains first a, then a delimiter followed by b. Let $n := |a|$ and $k := |b|$. Obviously, this problem is simple if b fits into memory, i.e. $k \leq m$. In the following, we are interested in solutions that use $o(k)$ memory. We can solve this problem in

- randomized $\mathbf{StrSort}(3, O(\log n))$ (Monte Carlo, i.e. with a polynomially small chance of a false positive), in

- randomized expected $\mathbf{StrSort}(6, O(\log n))$ (Las Vegas, i.e. with correct output, but the number of passes is only expected, and can be higher with polynomially small probability), and in

- randomized $\mathbf{StrNet}(3, O(\log n), 2, I)$ as a Monte Carlo algorithm, and randomized $\mathbf{StrNet}(6, O(\log n), 2, I)$ as a Las Vegas algorithm.

We will use the Karp and Rabin fingerprinting scheme [KR87] to solve this problem. In this algorithm sketch, we will not give details of the fingerprinting scheme, but will simply point out which properties we use for our algorithm.

Our algorithm is as follows. First, we compute a fingerprint (or hash) h of the string b, and store it in memory. Then, we similarly compute fingerprints for all length-k substrings of a. If any of these fingerprints match h, then with high probability this implies that the corresponding substring is identical to b. Since the fingerprint computation is deterministic, outputting the result of these comparisons can only lead to false positives.

The algorithm can be made Las Vegas instead of Monte Carlo by actually comparing the candidate matches to b on a symbol-by-symbol basis.

We now describe the implementation in more detail. The Karp and Rabin fingerprint can be computed using a linear pass on b, and requires $O(\log n)$ bits to store. Computing the fingerprint for every k-symbol substring of a requires a simple trick to do efficiently. We compute the fingerprints in the order in which the substrings appear in a, i.e. by a linear pass over a. When going from one substring to the next by appending an element of a, we can easily update the fingerprint if we have access to the deleted element at the beginning of the old substring. Note that this can easily be accomplished in the interleaved **StrNet**-model.

We achieve the same in the **StrSort**-model by creating two interleaved copies of a, such that the i-th element of the first copy appears next to the $(i-k)$-th element of the second copy. Thus if a_i is the i-th element of a, we create the stream

$$a_1 a_2 \dots a_k a_1 a_{k+1} a_2 a_{k+2} a_3 \dots a_n a_{n-k+1}. \tag{6.2}$$

This stream can easily be computed by using one streaming pass in which for each element a_i we write pairs (a_i, i) and $(a_i, i+k)$, and then sorting the resulting stream on the second component.

Thus, the Monte Carlo variant of the algorithm runs in two streaming passes and one sorting pass. The Las Vegas variant requires one additional streaming and sorting pass each per fingerprint match. This leads to 2 passes for a correct match, and an expected number of $n^{-\Omega(1)}$ passes for false positives, bringing the total expected number of passes to slightly more than 5.

The StrNet(I)-Algorithm: In the interleaved **StrNet**-model, we can replace each sorting pass by a single interleaving pass. To generate the stream (6.2), we interleave two copies of string a in a single pass by reading one at an offset of k elements to the order. In the Las Vegas variant, comparing b to a substring of a can be done in a single pass of reading two streams containing a and b, respectively. We first advance the a stream to the beginning of the substring we want to compare, and read and compare both streams a character at a time.

Suffix Array Computation

We now present an algorithm based on ideas from [MM93] for computing the suffix array of a string. The suffix array is a compact way of representing the ordering of the suffixes of a string, and defined as follows.

Definition 6.24 (Suffix Array)
Let a be a string over an ordered alphabet Σ', and $n := |a|$. For $i = 1, 2, \dots, n$ let $\mathfrak{a}_i$ be the

string $a_i a_{i+1} \dots a_n$ (i.e. the suffix of a starting at the i-th position). Let b_i be the position of $\mathfrak{a}_i$ in the lexicographic ordering of all suffixes $\{ \mathfrak{a}_i \mid 1 \leq i \leq n \}$. The suffix array of a is the list $b_1, b_2, \dots, b_n$. $\square$

Input: A string a over an ordered alphabet Σ'.

Output: The suffix array of a.

We can solve this problem in

- deterministic $\mathbf{StrSort}(O(\log n), O(\log n))$, and in
- randomized $\mathbf{StrNet}(O(\log^2 n), O(\log n), 2, I)$ replacing some sorting operation by single passes.

The naive algorithm for computing an suffix array would consider all suffixes of the string and order them by Quicksort, using string comparison as the comparison operation. Since a string comparison can take time $O(n)$ (where n is the length of string a) time in the worst case, this algorithm runs in $O(n^2 \log n)$ worst case time. A more clever algorithm is presented by Manber and Myers in [MM93]. The idea is to start by sorting according to only the first symbol of each suffix, and then successively refining the order by expanding the considered part of each suffix. We exploit the fact that each proper suffix of the whole string is also the suffix of another suffix to reduce the number of comparisons using a doubling technique.

Let the h-order of the suffixes be their lexicographic order, where only the first h symbols in each suffix are considered. The following observation was made by Manber and Myers [MM93].

Fact 6.25 (Manber and Myers)
Sorting the suffixes using for each suffix $\mathfrak{a}_i$, the position of $\mathfrak{a}_i$ in the h-order as its primary key, and the position of $\mathfrak{a}_{i+h}$ in the h-order as its secondary key, yields the $2h$-order. $\square$

We will now show how the 1-order of the suffixes can be constructed in $O(1)$ passes in the **StrSort**-model, and how using Lemma 6.25, we can in $O(1)$ passes obtain the $2h$-order from the h-order of the suffixes. In summary, this will allow us to compute the suffix array of a string in $O(\log n)$ passes in the **StrSort**-model.

We will store the h-order of a as a stream of pairs (i, b_i), where b_i is the position of $\mathfrak{a}_i$ in the h-order. Let a_i be the i-th element of a. To compute this representation of the 1-order of a, in a linear pass on a, create the stream (a_i, i). We then sort the stream on the first component (breaking ties arbitrarily). This will create a stream

$$(a_{i_1}, i_1)\ (a_{i_2}, i_2)\ (a_{i_3}, i_3) \dots (a_{i_n}, i_n).$$

The numbers $i_1, i_2, \dots, i_n$ are the suffix-indices in the 1-order of the string. In a single streaming pass, we therefore transform this stream into

$$(i_1, 1)\ (i_2, 2)\ (i_3, 3) \dots (i_n, n),$$

the desired representation of the 1-order. This took one sorting pass and two streaming passes.

Now we will see how to transform a representation of an h-order into an $2h$-order representation using Lemma 6.25. Given a stream

$$(1, b_1)\ (2, b_2) \ldots (n, b_n), \tag{6.3}$$

we first in a single pass produce the stream

$$\begin{aligned}(1, b_1)\ (2, b_2) \ldots (h, b_h)\ (h+1, b_{h+1})\ (1, b_{h+1})\ (h+2, b_{h+2})(2, b_{h+2}) \ldots \\ \ldots (j, b_j)\ (j-h, b_j) \ldots (n, b_n)\ (n-h, b_n),\end{aligned}$$

which we sort on the first component to yield

$$\begin{aligned}(1, b_1)\ (1, b_{h+1})\ (2, b_2)\ (2, b_{h+2})\ (3, b_3)\ (3, b_{h+3}) \ldots (j, b_j)\ (j, b_{j+h}) \ldots \\ \ldots (n-h, b_{n-h})\ (n-h, b_n)\ (n-h+1, b_{n-h+1})\ (n-h+2, b_{b-h+2}) \ldots (n, b_n).\end{aligned}$$

Now it is easy to produce a stream that has tuples of i followed by its primary and secondary key according to Lemma 6.25. In one streaming pass, we obtain

$$\begin{aligned}(1, b_1, b_{h+1})\ (2, b_2, b_{h+2})\ (3, b_3, b_{h+3}) \ldots (j, b_j, b_{j+h}) \ldots \\ \ldots (n-h, b_{n-h}, b_n)\ (n-h+1, b_{n-h+1}, 0)\ (n-h+2, b_{b-h+2}, 0) \ldots (n, b_n, 0).\end{aligned} \tag{6.4}$$

Sorting on the second, and in case of a tie on the third component, we obtain a stream where the first components are the indices of the suffixes sorted by their $2h$-order. Like above in the 1-order, we can easily convert this into a representation of the $2h$-order.

In summary, we needed $O(1)$ passes to construct the 1-order and $O(1)$ passes to move from an h-order to a $2h$-order, which we have to do $\lceil \log n \rceil$ times. Thus, the problem is solvable in **StrSort**$(O(\log n), O(\log n))$, since $O(\log n)$ bits suffice to store the indices i.

The StrNet(I)-Algorithm: In the interleaved **StrNet**-model, we can obtain stream (6.4) by interleaving two copies of stream (6.3) in a single pass, thus saving one sorting pass. It is not obvious how to avoid the second sorting pass in each stage, so we have to replace it by a $O(\log n)$-size sorting network to obtain a **StrNet**(I)-algorithm.

6.7.3 Graph Algorithms

For the following, we assume that a graph is given as the list of its edges (u, v) in some arbitrary order. In the **StrSort**-model it is a simple matter to generate a stream of vertices out of this. This can be done by using a streaming pass to output two elements "u" and "v" for each edge (u, v), and then sorting this list of vertex identifiers. In one more streaming pass, we can remove the duplicates in the resulting stream, yielding the desired list of vertices.

Undirected s-t-Connectivity

We gave a **StrSort**$(O(\log n), O(\log n))$ algorithm for this problem in Section 6.3.2. While some sorting operations of this algorithm can be replaced by single passes in **StrNet**(I), a straight-forward implementation is still in randomized **StrNet**$(O(\log^2 n), O(\log n), 2, I)$. Since most algorithms given below reduce to the connectivity algorithm, we do not directly obtain more efficient **StrNet**(I)-algorithms for them, either.

Bipartiteness

Input: An undirected graph G.

Output: Is G bipartite, i.e. can its nodes be colored with two colors such that no vertices of the same color are connected by an edge?

We will describe a simple reduction of this problem to the undirected connectivity problem from Section 6.3.2. This implies that we can decide bipartiteness in randomized **StrSort**$(O(\log n), O(\log n))$.

The reduction is as follows. First, we transform G into $G' = (V', E')$, where

$$V' = V \times \{0, 1\} \quad \text{and} \quad E' = \{((u,0),(v,1)), ((u,1),(v,0)) \mid (u,v) \in E\}.$$

This can easily be done in a single streaming pass. Next, we compute the connected components of G'. The following lemma then provides a simple test for bipartiteness of G'.

Lemma 6.26
The graph G is bipartite iff there is no $v \in V$ such that $(v,0)$ and $(v,1)$ belong to the same connected component of G'.

Proof: We use the fact G is bipartite iff it contains no odd cycles.

"⇒": Suppose the contrary, i.e. $(v,0)$ and $(v,1)$ belong to the same connected component, but G is bipartite. We show a contradiction. Suppose $(v,0)$ and $(v,1)$ are connected through a path $(v,0)(v_2,1)(v_3,0)\dots(v_{2k+1},0)(v,1)$. This path necessarily has to have odd length since the second component of the elements in V' (the 0,1-labels) alternates along every edge in E'. But this implies that $vv_2v_3\dots v_{2k+1}v$ is actually an odd cycle in G, and therefore G is not bipartite.

"⇐": If G is not bipartite, then it contains an odd length cycle. Let $v_1v_2\dots v_{2k+1}v_1$ be such a cycle in G. This implies that there is a path $(v_1,0)(v_2,1)(v_3,0)\dots(v_{2k+1},0)(v_1,1)$ in G', and thus $(v_1,0)$ and $(v_1,1)$ are in the same connected component of G'. □

Deterministic Directed s-t-Connectivity

We next consider a problem that is **LOGSPACE**-complete [CM87], and can be solved in **PLog-StrSort**. This does *not* imply that **LOGSPACE** is contained in **PLog-StrSort**, since the corresponding reductions are not necessarily in **PLog-StrSort**, as they might have outputs of size $\omega(n)$. The problem is the following.

Input: A directed graph $G = (V, E)$, where each node has out-degree at most 1, and two nodes $s, t \in V$.

Output: Is there a path from s to t in G?

We again reduce this problem to the undirected connectivity algorithm from Section 6.3.2, and it therefore is in randomized **StrSort**$(O(\log n), O(\log n))$.

Construct an undirected graph G' from G by first deleting the edge leaving t (if there is one), and then replacing each directed edge by an undirected edge. Then there is a path from s to t iff s and t are in the same connected component of G'. To see this note that each connected component C in G' has at most one maximal element u_C in G (i.e. an element without outgoing edges), and that therefore every node in C has a path to u_C in G.

Since t is maximal after removing its outgoing edge, this shows the correctness of the algorithm.

Minimum Spanning Tree

We now show how to compute a minimum spanning tree (MST) for a graph, using the algorithm for undirected connectivity as a subroutine. The problem can be stated as follows.

Input: A undirected, connected, weighted graph $G = (V, E)$, as a stream of edges with weights.

Output: A subset of edges of E that forms a minimum spanning tree of G, i.e. a set of edges that forms a tree connecting all vertices in V, which has a minimal sum of weights among all such trees.

This problem can be solved in randomized **StrSort**$(O(\log^2 n), O(\log n))$.

For our algorithm, we are going to use a divide and conquer approach. The divide step is as follows.

1. Sort the edges $E = \{e_1, e_2, \ldots, e_m\}$ by increasing weight.
2. Let $E_0 = \{e_1, e_2, \ldots, e_{m/2}\}$ be the "lighter" half of the edges.
3. Compute the connected components of (V, E_0).

In the conquer step, we compute minimum spanning trees for

(a) each connected component in (V, E_0), and

(b) the graph (V', E') where V' is the set of connected components in (V, E_0), where E' contains the edges in $E \setminus E_0$ that connect different components in V'. The edges in E' are labeled according to what components they connect.

It is not hard to see that the union of the edges in the individual spanning trees yields the answer to the minimum spanning tree problem for G. Moreover, the size (number of edges) of each sub-problem in the conquer step reduces by a factor of at least 2. So we need $O(\log n)$ phases, each requiring a computation in **StrSort**$(O(\log n), O(\log n))$, leading to the claimed total number of passes.

Maximal Independent Set

An independent set of an undirected graph $G = (V, E)$ is a set $S \subseteq V$, such that $(S \times S) \cap E = \emptyset$, i.e. no two nodes in S are connected by an edge.

Input: An undirected graph $G = (V, E)$.

Output: An maximal independent set of G, i.e. an independent set S such that $S \cup \{v\}$ is not an independent set for all $v \in V \setminus S$.

Luby [Lub86] gives a simple randomized parallel algorithm to compute a maximal independent set, which can easily be implemented in randomized **StrSort**$(O(\log n), O(\log n))$, even without making use of the general simulation of Lemma 6.7. The core of the algorithm is the following:

1. We randomly pick a set $I \subseteq V$, by choosing each vertex $v \in V$ with probability $1/(2 \cdot \deg(v))$.

2. For each pair $u, v \in I$ such that $(u, v) \in E$, we remove the node with the smaller degree from I.

3. Now, I is an independent set. We add I to our output set S, and remove I and all the neighbors of the elements in I from G.

4. Repeat until G is empty.

Clearly, the output will be a maximal independent set. We note that all steps are easily implementable in the **StrSort**-model. In [Lub86] it is shown that the number of edges in G is expected to drop by a constant factor in each iteration of the above algorithm, thus requiring only $O(\log n)$ iterations with high probability.

Computing Pagerank

"Pagerank" [BMPW98, PBMW99] is an algorithm developed by the inventors of the Google search-engine (`www.google.com`) to order web-pages by their relative importance. The key idea is that a web-page's importance depends on the importance of the pages linking to it, since high quality pages tend to link to other quality pages.

More formally, we are given a directed graph $G = (V, E)$, where nodes correspond to webpages, and a directed edge (u, v) represents a link from page u to page v. Let $\deg(v)$ be the out-degree of a vertex v. Then the pagerank $(R_v)_{v \in V}$ for the pages in V is defined as the solution of

$$R_v = (1 - c)E(v) + c \sum_{(u,v) \in E} \frac{R_u}{\deg(u)}, \tag{6.5}$$

where $E(v)$ is some initial distribution of rank (e.g. assigning rank to popular websites like `www.yahoo.com` or `www.cnn.com`) and $c < 1$ is some constant.

We note that $(R_v)_{v \in V}$ can be computed iteratively, gradually improving the quality of approximation. This can be done in the **StrSort**-model, as explained in [PBMW99]. Essentially, edges are sorted by their end-point and labeled with the out-degree of their starting point. Interleaving this with previously computed approximations of R_v using a sorting pass, we can then compute the next iteration of values R_v by equation (6.5) in a single streaming pass. It has been observed that in practice a small number of passes suffices for convergence of the values R_v (cf. [PBMW99]).

Tree Contraction

"Tree contraction" is a technique that forms the basis of many parallel algorithms (see [KR90], section 2.2.3). Given a rooted tree, it repeatedly contracts all boughs of the tree. A *bough* is a path from a leaf to the first node with more than one child in the tree.

This contraction yields a new tree with at most half the number of leaves of the original tree. The bough-contraction process is then repeated $O(\log n)$ times to finally yield a tree containing a single node.

Depending on the problem to be solved, additional operations can be carried out during the contractions. For example, if all nodes in a tree are labeled with a value, we might want to compute for each node the sum of the values of its descendants (this is also called a "treefix sum"). We will use treefix sums extensively in Section 6.8 to compute minimum cuts in the **StrSort**-model.

We will now give a $\mathbf{StrSort}(O(\log^2 n), O(\log n))$ algorithm for tree-contractions. As a simple application, we will also point out how our contraction algorithm can be used to determine whether an undirected graph contains a cycle.

Contracting Paths. Let us first consider an algorithm that contracts a single undirected path. The basic idea behind our algorithm is to take a pair of edges $(u, v), (v, w)$ and replace them by a single edge (u, w), removing vertex v from the path. This replacement can be done in parallel for many edge pairs, as long as we make sure that we do not contract "overlapping" edge-pairs, e.g. pairs $(u, v), (v, w)$ and $(v, w), (w, x)$, at the same time.

To make contractions disjoint, we employ a simple trick from PRAM algorithms: we randomly label all vertices with a sign $\in \{+, -\}$. We then contract $(u, v), (v, w)$ only if u and w have a '$-$'-sign, and v has a '$+$'-sign. It is easy to see that this ensures that we do not contract overlapping pairs.

Our algorithm is to repeat the following $O(\log n)$ times:

1. Label all vertices v_i at random with a sign $s_i \in \{+, -\}$.

2. Sort the edges such that for all vertices v_i with $s_i =$ '$+$', v_i's incident edges appear together.

3. In a single streaming pass, for each vertex v_i with $s_i =$ '$+$', determine if both its neighbors labels are '$-$', and if so, contract its incident edges.

The probability of a vertex being contracted in one round of this algorithm is 1/8 (the probability that it receives a '$+$'-label, and its neighbors a '$-$'-label). Thus, the number of vertices reduces by an expected factor of $\frac{7}{8}$ in every round, and the path gets reduced to two single vertices in $O(\log n)$ rounds whp. The algorithm therefore is in $\mathbf{StrSort}(O(\log n), O(\log n))$.

Contracting Trees. The above algorithm almost directly yields a tree-contracting algorithm by applying it $O(\log n)$ times to the boughs of the tree. This can be done by marking all vertices on the boughs, and then running the path-contraction algorithm on the marked vertices. Note that the path-contraction algorithm also works if the input is a collection of paths.

There is a problem with this approach, though: we do not initially know which vertices are on the boughs. Our solution is the following: we initially mark all degree-2 vertices in the graph. Some of these are on the boughs, some might be on paths internal to the tree. We then run the path-contraction algorithm on the marked vertices, storing for each contracted path what vertices were contracted into it.

After the contraction, we can easily check which contracted paths started at leaves, since they end in degree-1 vertices (the leaves). We then undo the contraction done for all other paths. Since all boughs were contracted to a single edge, removing the edges incident to the leaves (i.e. to all degree-1 vertices) yields the desired contracted tree.

Contracting a tree requires $O(\log n)$ invocations of the path-contraction algorithm, and can therefore be done in $\mathbf{StrSort}(O(\log^2 n), O(\log n))$.

Detecting Cycles in Undirected Graphs

The tree-contraction algorithm can be used to determine whether an undirected graph contains a cycle. Essentially, we simply run the tree-contraction algorithm. It either finishes, since the graph does not contain a cycle, or it runs into a problem, because the graph does contain a cycle.

Let us be a bit more precise about what "running into a problem" means. The tree-contraction algorithm alternates between two phases: contracting degree-2 vertices, and deleting all degree-1 vertices. Both operations do not change the fact whether the graph contains a cycle.

If the graph contains a cycle consisting only of degree-2 vertices, we will discover that in the "degree-2 vertex contraction" phase. Otherwise, note that the number of degree-1 vertices halves in every iteration of the algorithm. So eventually, we will end up with a graph where all degrees are zero or at least two. The original graph then contains a cycle iff there are any edges remaining in this final graph.

Mincut

Due to its complexity, we defer this algorithm to Section 6.8.

6.7.4 Geometric Problems

Red-Blue Line Segment Intersection

We now consider a geometrical problem, motivated by geometric range queries, called Red-Blue-Intersection (RBI).

Input: List of red and blue line segments in the plane. No two red line segments intersect and no two blue line segments intersect.

Output: Number of intersection points between red and blue line segments.

In many applications of this problem, one is actually interested in a list of intersection points, but that might require an output greater than the input, which our streaming model does not allow.

There are a number of algorithms solving this problem in the RAM model [CEGS95, PS93], parallel models [GSG92, DF93] or using an external memory algorithm [AVV95]. However, most of these algorithms seem to require a topological sorting operation to order the red or blue line segments: if a line segment s_1 is "below" a line segment s_2, then s_1 will appear before s_2 in the ordering. This makes it possible to solve the problem using a sweep-line approach by processing the line segments in that order.

Since we do not know how to do a topological sort in the **StrSort**-model, we will give a modified algorithm that avoids the topological sorting, but requires $O(n \log^3 n)$ total work, as opposed to the optimal $O(n \log n)$.

In the following, we will give a deterministic $\mathbf{StrSort}(O(\log^2 n), O(\log n))$-algorithm solving the RBI problem. Before considering the general case however, we will discuss an algorithm that only works if the slopes of the red line segments are all smaller than the slopes of the blue line segments. We call this problem SLOPE-RESTRICTED RBI. This problem can be solved in $\mathbf{StrSort}(O(\log n), O(\log n))$. Later, we will reduce the general case to this special case.

Wlog, we assume that no two x-coordinates or y-coordinates have the same value; this can be achieved by rotating the whole input by a small angle – this does not change the slope restriction, and can easily be accomplished in a single streaming pass.

Let the *slab* of a line segment $(x_1, y_1) - (x_2, y_2)$ (with $x_1 < x_2$) be the vertical strip $[x_1, x_2] \times \mathbb{R}$ of the plane. Then we have the following.

Proposition 6.27
The number of intersections of red and blue line segments in SLOPE-RESTRICTED RBI *is equal to $r_L - r_R + b_L - b_R$, where*

- r_L := *the number of pairs of red line segments s and left end-points of blue line segments p, such that p is in the slab of s and* below *s,*
- r_R := *the same for right end-points of blue line segments,*
- b_L := *number of pairs of blue line segments s and left end-points of red line segments p, such that p is in the slab of s and* above *s,*
- b_R := *the same for right end-points of red line segments.*

Proof: We show that each pair of red and blue line segments contributes 1 to the above sum iff it intersects, and 0 otherwise. Consider a pair of intersecting segments. Notice that if the red line segment is shrunk by moving both endpoints to the intersection point, then the value of the sum $r_L - r_R + b_L - b_R$ does not change. However, if the segment becomes sufficiently small, then the contribution to b_L becomes 1, while the segment pair does not contribute to any other sum.

For two line segments that do not intersect, again the contribution does not change if we contract the red line segment by moving one endpoint to the other. But then the contribution to r_L and r_R is zero, and the contribution to b_L and b_R is the same (either 1 or 0 each, depending on whether the red segment is above the blue one or not). Thus, the total contribution to the sum is 0. $\square$

So to solve SLOPE-RESTRICTED RBI, it suffices to compute the quantities r_L, r_R, b_L and b_R. For symmetry reasons, it suffices to show how to compute one of them. This is shown in the following Lemma, which immediately implies that SLOPE-RESTRICTED RBI is solvable in $\mathbf{StrSort}(O(\log n), O(\log n))$.

Lemma 6.28
The quantity r_L can be computed in $\mathbf{StrSort}(O(\log n), O(\log n))$ for a SLOPE-RESTRICTED RBI *instance.*

Proof: First, we preprocess the input to create a stream that contains the red line segments and the left end-points of the blue line segments. We use a divide-and-conquer approach. In each sub-problem, we are only concerned with a slab $[x_{\min}, x_{\max}[\times \mathbb{R}$, so initially $x_{\min} = -\infty$ and $x_{\max} = +\infty$.

The divide step is as follows:

1. Compute the median x_{mid} among the x-coordinates of the blue points in the problem. This can be done in one sorting and one streaming pass.

2. Split each red line segment that crosses the x_{mid}-coordinate into two, a left half ending at x_{mid}, and a right half starting at that position. This can be accomplished in a single streaming pass.

3. We then have two sub-problems, the *left* sub-problem, and the *right* sub-problem.

   Left sub-problem: Consists of all blue points with x-coordinate less than x_{mid} and all red line segments in the slab $[x_{\min}, x_{\text{mid}}[\times\mathbb{R}$ that do *not* cross the whole slab, i.e. whose projection onto the x-axis does *not* cover $[x_{\min}, x_{\text{mid}}[$.

   Right sub-problem: Consists of all blue points with x-coordinate $\geq x_{\text{mid}}$ and all red line segments in the slab $[x_{\text{mid}}, x_{\max}[[\times\mathbb{R}$ that do *not* cross the whole slab.

Since we halve the number of blue points in a sub-problem in every step, after $O(\log n)$ recursions, we are left with trivial sub-problems with just a single blue point, for which we can easily compute r_L.

For the conquer-step, suppose that we solved the left and right sub-problems, i.e. counted the number r_L of red line segments above blue points for these two sub-problems. Their sum is not yet equal to the value r_L for the whole problem since we did not consider the line segments that cross the left or right slab completely. So it remains to compute the number of these line segments that are above blue points. This can be done as follows.

1. Create a list of the blue points and all red line segments that cross either the left or right slab completely (i.e. precisely the line segments that were not passed down to the left or right sub-problems).

2. Sort this list by increasing y-coordinate. More precisely sort its elements by y-coordinate for the blue points, and by the y-coordinate at the midpoint x_{mid} for the red line segments.

3. Now read this stream in order, maintaining two counters ℓ and r for the number of blue points seen in the left and right slabs so far, respectively. When encountering a blue point, we increase ℓ or r (depending on what slab the point is in) by 1. If we encounter a red line segment in the left slab, we increase r_L by ℓ, for a red line segment in the right slab, we increase r_L by r.

Clearly, this counts for every red line segment that completely covers a slab the number of blue points below it. Thus, the returned value of r_L is the correct answer for the subproblem. Since each divide and conquer step takes $O(1)$ streaming and sorting passes, this shows that r_L can be computer in **StrSort**$(O(\log n), O(\log n))$. $\square$

It remains to be shown how to reduce Red-Blue Intersection to Slope-Restricted RBI. This can also be accomplished using a divide and conquer approach. Suppose we are given a set R of red line segments, and a set B of blue line segments. The reduction is as follows:

1. Pick a slope α such that half the elements of R have a slope $\leq \alpha$, and half have a larger slope.

2. Let $R_{\leq}$ and $R_{>}$ be the elements of R with slope $\leq \alpha$ and $> \alpha$, respectively. Similarly, define $B_{\leq}$ and $B_{>}$.

3. Use the Slope-Restricted RBI algorithm to compute the number of intersections in $R_{\leq} \cup B_{>}$ and $R_{>} \cup B_{\leq}$, noting that for symmetry reasons that algorithm can also be applied if the slopes of all red line segments are *larger* than the slopes of all blue line segments.

4. Recursively compute the number of intersections in $R_{\leq} \cup B_{\leq}$ and in $R_{>} \cup B_{>}$.

Note that in the last two steps, any pair of red and blue line segments is tested for an intersection exactly once, thus the algorithm returns the correct result.

Since the number of red line segments in a problem is halved in every step of the recursion, there are only $O(\log n)$ levels to the recursion. As every level can be solved in $\mathbf{StrSort}(O(\log n), O(\log n))$, the complete algorithm is in $\mathbf{StrSort}(O(\log^2 n), O(\log n))$.

6.8 Mincut in StrSort

6.8.1 The Problem

We will now turn to a **P**-complete problem that is solvable in **PLog-StrSort**. Again, this does not imply that **P** is in **PLog-StrSort**, since the commonly used reductions are not in **PLog-StrSort**.

The problem is to find a minimum cut in a graph.

Input: An undirected graph $G = (V, E)$.

Output: A cut $S \subseteq V$ that minimizes the number of edges crossing the cut, i.e. the size of the set $E \cap S \times (V \setminus S)$.

We base our streaming algorithm on a result by Karger [Kar00], who shows that this problem can be solved in almost linear time. His algorithm can be implemented in **PLog-StrSort** by reordering the computations to occur in a more parallel fashion. In the following, we discuss the necessary changes to Karger's algorithm. The focus is on the streaming implementation of the algorithms, and not its correctness for which we refer to Karger's paper. We will therefore only re-state his algorithm as far as necessary for describing its implementation in the streaming model, and will assume that the reader is familiar with the journal paper [Kar00].

Outline of the Algorithm

In the following we will assume that G is connected. If G were not connected (which we can easily check with the connectivity algorithm from Section 6.3.2), then the minimum cut has size 0.

Given a spanning tree T of G, and a cut S, we will say that S k-respects T, if at most k edges of T cross the cut S. Note that if we are given a spanning tree T and the set of its edges that cross the cut, then this uniquely defines the cut.

With this terminology, Karger's algorithm can be summarized as follows:

1. Compute a packing of spanning trees in G that whp contains a tree that is 2-respected by a minimum cut.

2. For each tree in the packing, find the smallest cut that 2-respects the tree.

We will now describe the algorithm in more detail, following Karger's presentation closely, with frequent references to [Kar00].

6.8.2 Computing a Tree-Packing

Before computing the tree packing, we first compute a skeleton version of the graph. This is done by removing each edge independently with probability $1-\Theta((\log n)/(\varepsilon^2 c))$ (where c is the minimum cut size). For details see section 4.2 in [Kar00] and [Kar99]. The resulting graph then whp has a weighted spanning tree packing of size $O(\log n)$ that has the desired property.

Note that we do not initially know c, the size of the minimum cut. But since for the above we only have to know c within a constant factor, we can simply run the algorithm for $O(\log n)$ different values of c, e.g. all powers of two. This method of determining c is the only reason why we cannot simply use our algorithm for weighted graphs. It would therefore be interesting to see whether approximation algorithms for the minimum cut size such as [Mat93] can be implemented in the **StrSort**-model to overcome this problem.

To compute the weighted tree packing itself, we use the packing algorithm by Plotkin, Shmoys and Tardos [PST95]. This algorithm reduces to a sequence of $O(\log n)$ minimum spanning tree computations, where the costs on the edges depend on the previously computed spanning trees. We already know how to compute minimum spanning trees in **PLog-StrSort** (cf. Section 6.7.3), and updating the costs can easily be done in linear passes over the edges. Thus computing the whole packing is possible in **PLog-StrSort**.

There is one small caveat: we cannot compute and store all $O(\log n)$ trees at the same time, since this would require more than $O(n)$ storage. We therefore compute the trees one at a time, and run the algorithm described below to determine the minimal cut that 2-respects the tree. The only information we have to maintain between tree computations are the costs on the edges (or equivalently, the sum of the trees packed so far), which needs only linear storage.

6.8.3 Minimum Cuts that 1-Respect a Tree

For each of the $O(\log n)$ trees in the packing, we now determine the smallest cut that 1-respects or 2-respects the tree. By construction of the tree packing, the smallest cut among these will whp be a minimum cut of the graph G.

We begin with finding the smallest cut that 1-respects a tree. This easier case will serve as an opportunity to introduce notation, and to describe the techniques used to implement the algorithm in **PLog-StrSort**. First, some definitions.

Definition 6.29

$\mathcal{C}(V_1), \mathcal{C}(V_1, V_2)$ **:** *For node set $V_1, V_2 \subseteq V$, we denote by $\mathcal{C}(V_1, V_2)$ the number of edges from V_1 to V_2 (double-counting edges in $(V_1 \cap V_2) \times (V_1 \cap V_2)$), and set $\mathcal{C}(V_1) := \mathcal{C}(V_1, \overline{V_1})$.*

$v^\uparrow$**:** *For a node v in a rooted tree, $v^\uparrow$ denotes the set of all ancestors of v in the tree, including v.*

$v^\downarrow$**:** *For a node v in a rooted tree, $v^\downarrow$ denotes the set of all descendants of v in the tree, including v.*

$f^{\downarrow}(v)$**:** *For a function f defined on all vertices of a rooted tree, the* treefix sum *of f, denoted by $f^{\downarrow}$ is defined as*

$$f^{\downarrow}(v) = \sum_{w \in v^{\downarrow}} f(w). \quad \square$$

Karger's algorithm reduces the computation of a minimum cut to the computation of a series of treefix sums $f^{\downarrow}$ for "simple" functions f. The key insight is that these computations can be carried out in the **StrSort**-model.

Note that all cuts that 1-respect a tree are of the form $\mathcal{C}(v^{\downarrow})$. So it suffices to compute these values for all $v \in V$.

Lemma 6.30 ([Kar00], Lemma 5.9)
We have $\mathcal{C}(v^{\downarrow}) = \delta^{\downarrow}(v) - 2\rho^{\downarrow}(v)$, where $\delta(v)$ is the (weighted) degree of vertex v, and $\rho(v)$ is the total weight of edges whose endpoints' least common ancestor is v. $\square$

It will be instructive to consider how to compute $\delta^{\downarrow}(v)$ in the **StrSort**-model. Clearly, we can compute $\delta(v)$ for each vertex, by first sorting the edges by vertex, and then using an additional streaming pass to determine the degree of each vertex. So the problem is how to sum up the values $\delta(v)$ to yield $\delta^{\downarrow}(v)$. It turns out that we apply the tree-contraction algorithm from Section 6.7.3.

Computing Treefix Sums

Treefix Sums for Paths. First consider the simple case where the tree is actually a path $v_1, v_2, \ldots, v_n$, where v_1 is the leaf and v_n the root. In **PLog-StrSort**, we can sort the vertices in increasing distance from the leaf, yielding a stream $v_1 v_2 \ldots v_n$. Computing the values $\delta^{\downarrow}(v_i)$ is then the matter of a single streaming passes, assuming that each vertex v_i is labeled with the value of $\delta(v_i)$.

How do we compute the ordering of the vertices on a path? This can be done by a variation of the path-contraction algorithm from Section 6.7.3 (page 140). In that algorithm we repeatedly replace degree-2 vertices by an edge connecting their neighboring vertices. While doing so, for each edge, we keep a (sorted) sequence of the vertices contracted into it. After $O(\log n)$ contractions, a path gets reduced to a single edge, which then contains the ordered sequence of vertices on the path.

Treefix Sums for Trees. For arbitrary trees, we use a variant of the tree-contraction algorithm of Section 6.7.3 to compute the treefix sum in $O(\log n)$ phases. In each phase, we compute the treefix sum for all nodes that are on boughs of the tree, using the path treefix algorithm described above. For the parent node of each bough, we add the treefix sum of its descendant to that node's δ-value. We then drop all boughs from the tree and repeat. As in the tree-contraction algorithm, $O(\log n)$ phases suffice to compute treefix sums for all vertices.

All the computations on trees given in the following will be (sometimes non-trivial) modifications of this basic paradigm for computing treefix sums by repeatedly computing them on boughs. An example is the following result.

Corollary 6.31
$\mathcal{C}(v^{\downarrow})$ can be computed for all v in **PLog-StrSort**. *Thus, we can find the minimum cut than 1-respects a tree in* **PLog-StrSort**.

Proof: We use Lemma 6.30. Since δ can be computed on a node-by-node basis in **PLog-StrSort**, we can use the treefix sum algorithm described above to compute $\delta^{\downarrow}(v)$ for all v. Let us now consider $\rho(v)$.

For this, we first label each vertex v with the edges incident to v. Note that this just rearranges the information about G, and therefore does not use more space than the description of G itself. We will compute $\rho(v)$ by moving these labels upward in the tree, starting at the leaves. Since each edge is stored twice in the tree, eventually these two copies will meet. By construction (see below), this will be at the least common ancestor of the end-points of the edge. In that case, we will increase ρ by one for that vertex, and delete the two copies of the edge.

More precisely, we operate in phases just as for the treefix sum computation. For each bough, we order its vertices starting at the leaf. If any edge-label appears twice on the bough (which we can detect by sorting the edge-labels by what boughs they appear on), the vertex closer to the root is the least common ancestor of the edge's endpoints. The remaining edge labels, i.e. the ones just appearing once on the bough, are moved to the parent node of the bough. We then remove the boughs, and repeat.

This allows us to compute ρ in **PLog-StrSort**, and therefore $\rho^{\downarrow}$. Actually, since the computation of ρ so closely follows the treefix computation, both can be carried out simultaneously. $\square$

6.8.4 Minimum Cuts that 2-Respect a Tree

We distinguish two cases for the tree-edges that determine the cut: one is an ancestor of the other, or not. We will write $v \perp w$ to mean that $v \notin w^{\downarrow}$ and $w \notin v^{\downarrow}$. So if the edges determining the cut are the tree-parent edges of v and w, then we distinguish the cases $v \perp w$ and $v \in w^{\downarrow}$ (wlog).

Case 1: $v \perp w$

It is shown in Lemma 7.5 of [Kar00] that the minimum cut in the case $v \perp w$ is $\min_v(\mathcal{C}(v^{\downarrow}) + \mathcal{C}_v)$. We already know how to compute $\mathcal{C}(v^{\downarrow})$ (see the previous section), so it suffices to compute $\mathcal{C}_v$. We will not define these values $\mathcal{C}_v$ directly, but will instead simply state Karger's algorithm for computing them.

In this algorithm, Karger uses the dynamic tree data structure of Sleator and Tarjan [ST83] that allows to efficiently perform the following operations on a rooted tree where every node v stores a value $\texttt{val}[v]$.

`AddPath`(v, x)**:** add x to $\texttt{val}[u]$ for every $u \in v^{\uparrow}$.

`MinPath`(v)**:** return $\min_{u \in v^{\uparrow}} \texttt{val}[u]$ as well as the u achieving this minimum.

The following algorithm is then used to compute $\mathcal{C}_v$ for a leaf v (cf. Lemma 7.7, [Kar00]):

1. Set $\texttt{val}[w] = \mathcal{C}(w^{\downarrow})$ for all w.
2. Return `LocalUpdate`(v).

We already know how to perform the first operation in **PLog-StrSort**. The algorithm `LocalUpdate` is defined as follows.

Algorithm 6.32 (`LocalUpdate`(v))

1. Call `AddPath`(v, ∞).
2. For each edge (v, u), call `AddPath`($u, -2\mathcal{C}(v, u)$).
3. For each edge (v, u), call `MinPath`(u).
4. Return the minimum result of Step 3.

It is not too hard to see that `LocalUpdate` can be implemented in **PLog-StrSort** by repeated contraction of the tree, while passing the `AddPath`-values upward, and updating the `MinPath`-value along the way. We will now describe an algorithm that computes $\mathcal{C}_v$ for *all* leaves of the tree in parallel in **PLog-StrSort**.

Computing $\mathcal{C}_v$ For All Leaves v. First, for each edge (v, u) where v is a leaf, our algorithm attaches a label $(v, -2\mathcal{C}(v, u))$ to node u. Also, for every leaf v, we add a label (v, ∞) to node v. Note that nodes might receive more than one label this way.

We are now going to move ("bubble") the labels upwards in the tree (toward the root). Whenever two labels (v, c_1) and (v, c_2) meet at a node, we replace them by a single label $(v, c_1 + c_2)$. In this way, we keep track of the effect of the `AddPath`-operations in steps 1 and 2 of the above algorithm, i.e. we can easily determine their influence on the `val`[$\cdot$]-values of nodes.

While moving the labels, we also perform the `MinPath`-computations from step 3. Note that for a node v, only the nodes through which labels of the form (v, c) move, can be candidates for the `MinPath` queries for node v (which yield $\mathcal{C}_v$). So we make sure to consider all nodes w through which labels (v, c) pass, and check whether $\mathcal{C}(w^{\downarrow}) + c$ is less than the previously recorded minimum for $\mathcal{C}_v$.

As with our treefix sum computations, we repeatedly move labels upwards on each bough. So let us first consider what we have to compute for a path $v_1, v_2, \ldots, v_k$.

First consider one leaf v. It might have labels at one or more of the nodes $v_1, v_2, \ldots, v_k$. We sort these labels by their position on the path (starting at the leaf), and compute their cumulative sum. By this we mean, if the labels are $(v, c_1), (v, c_2), \ldots, (v, c_j)$, then we replace them by $(v, c_1), (v, c_1 + c_2), \ldots, (v, \sum_{i=1}^{j} c_i)$.

If we just cared about $\mathcal{C}_v$ for this particular node v, we could now do a linear pass over the path, always remembering the last seen label $(v, \sum_{i=1}^{i'} c_i$, and check for the encountered nodes w whether $\mathcal{C}(w^{\downarrow}) + \sum_{i=1}^{i'} c_i)$ is less than the minimum estimate for $\mathcal{C}_v$ seen so far, and updating the estimate if it is. However, this approach does not generalize to computing $\mathcal{C}_v$ for all leaves simultaneously, since we do not have enough memory to store the current labels (v, c) for all leaves v.

We therefore employ a divide-and-conquer strategy. Note that for each leaf v, the labels (v, c) occur only at certain places on the path, and the contribution c stays constant in the intervals between these places. For each v, compute labels (v, i, j, c) describing these intervals, i.e. that v's label-value is c from node v_i to node v_j on the path ($i \leq j$).

The divide-and-conquer strategy is similar to the one employed for red-blue line intersections in Section 6.7.4. We split the path in the middle, and cut all intervals (v, i, j, c) that cross the mid-point into two. We also compute the minimum value of $\mathcal{C}(w^{\downarrow})$ on the left and right halves of the path, let us call them $\mathcal{C}_{\text{left}}$ and $\mathcal{C}_{\text{right}}$, respectively. For all intervals

(v, i, j, c) that entirely cross the left or the right half of the path, we do the following. Suppose (v, i, j, c) completely crosses the left half, then we output $c + \mathcal{C}_{\text{left}}$ as a new estimate for $\mathcal{C}_v$. Since v's contribution c does not change in that half of the path, the output value clearly is the minimum value attained in that half. We do the same for the right half.

After outputting all the estimates, they are sorted by v, and the new current minimum estimate of $\mathcal{C}_v$ is computed for all leaves v. We then continue to the next level of the divide-and-conquer algorithm, recursing on the left and the right halves of the path, after dropping from consideration all intervals that completely cover either half.

Similar to the analysis for the red-blue line intersection problem, it is not hard to see that $O(\log n)$ recursion levels are sufficient, and that the storage requirement remains linear.

To generalize the algorithm from paths to trees, we just have to aggregate all labels (v, c) on a bough on the parent of that bough. This can easily be done by first sorting the labels by v, and then summing up their values. Thus, $\mathcal{C}_v$ can be computed in **PLog-StrSort** for all leaves v in the tree.

Computing $\mathcal{C}_v$ For Nodes in a Bough. Let $v_1, v_2, \ldots, v_k$ be the nodes on the bough, with v_1 being the leaf. Then the following algorithm computes the values $\mathcal{C}_{v_i}$ (cf. [Kar00], section 7.3).

Algorithm 6.33 (Compute $\mathcal{C}_v$ for a Bough)

1. Set `val`$[w] = \mathcal{C}(w^{\downarrow})$ for all w.
2. Set $\mathcal{C}_{v_1} =$ `LocalUpdate`(v_1).
3. For $i = 2, 3, \ldots, k$ do:
   (a) Set $c_1 = \mathcal{C}_{v_{i-1}}$.
   (b) Set $c_2 =$ `LocalUpdate`(v_i).
   (c) Set $\mathcal{C}_{v_i} = \min(c_1, c_2)$.

Note that in the above algorithm, every call to `LocalUpdate` changes some `val`$[\cdot]$ entries, i.e. it is *not* side-effect free.

We will first only compute the `LocalUpdate`(v_i) values from step 3b of the algorithm. The true $\mathcal{C}_{v_i}$ values can then be computed by sorting the `LocalUpdate`(v_i)-values by v_i, and computing a running minimum as in step 3c of the above algorithm.

We can modify our **PLog-StrSort**-algorithm given above for the "leaf-case" to this bough-case. As previously, for all bough nodes v_i and edges (v_i, u) we first put labels $(v_i, -2\mathcal{C}(v_i, u))$ on the node u, and a label (v_i, ∞) on node v_i. For the following it will be important that we can always figure out which v_i's belong to the same bough. So for clarity, let us write $b_1, \ldots, b_k$ to refer to the vertices on the same bough b, ordered from leaf (b_1) to top (b_k).

To compute the `LocalUpdate`-values, we will again bubble labels up the tree, while keeping track of estimates for the values $\mathcal{C}_v$. Let us consider the case of bubbling values up a path. We employ the same divide-and-conquer strategy as before. The only change is that if there are labels (b_i, c_i) at the beginning of a path segment, then the contribution of b_i is $\sum_{j=1}^{i} c_j$, and not c_j.

This complication can be dealt with in variety of ways. Either we can always maintain these aggregated values $\sum_{j=1}^{i} c_j$ in the labels, as opposed to just the c_i. Or we can aggregate them as needed in our divide-and-conquer algorithm. Either way, it is not too hard to see that this again yields a **PLog-StrSort**-algorithm to compute the `LocalUpdate`-values along a path. As usual, this can be extended to an algorithm for the whole tree.

Computing $\mathcal{C}_v$ For All Nodes in the Tree. Karger observes ([Kar00], section 7.4) that a whole tree can be processed bottom up. First, we compute $\mathcal{C}_v$ for all nodes in the boughs. Then, all boughs get contracted with their parent nodes into a single node. We then repeat the algorithm. Clearly, $O(\log n)$ such contractions will leave us with a trivial single-node tree. Since each bough-computation can be carried out in **PLog-StrSort**, the whole computation can be carried out in **PLog-StrSort** as well.

Case 2: $v \in w^{\downarrow}$

As described in [Kar00], section 7.5, the case $v \in w^{\downarrow}$ can be solved by a straight-forward modification (even simplification) of the previous case $v \perp w$. The algorithm differs mostly in what values are "bubbled" up during our computations. We therefore omit the details.

6.9 Open Problems

In this work, we have studied several computational models for massive data sets. The weakest one, the traditional streaming model augmented with a sorting primitive (**StrSort**), still allows for the efficient solution of a variety of natural problems, such as undirected connectivity, MST, red-blue line intersections, computing a minimum cut, and several others. Since this class is a quite natural extension of the traditional streaming model, and efficiently implementable in practice, it deserves further study. This leads to several open problems.

New Algorithms in StrSort: Can the following problems be solved in **PLog-StrSort**: computing a breadth first or depth first traversal of a graph, topologically sorting a directed acyclic graph, determining directed connectivity, or evaluating an arbitrary circuit with $O(n)$ gates?

Lower Bounds in StrSort: Are the numbers of sorting passes used in our **StrSort**-algorithms tight? Since sorting is a powerful primitive, it would be interesting to obtain matching lower bounds, or reduce the number of sorting passes further.

Our second model, "streaming networks" models the computational power of distributed applications that communicate via streams, or equivalently, the power of tape-based computations. Allowed to arbitrarily interleave input streams, this model is strictly more powerful than the "streaming and sorting" model. The separation result between the two models leaves open several interesting questions.

Natural Separation: It would be interesting to find more natural problems than Alternating Sequence that separate the two classes. A candidate problem could be Subsequence, in which given two strings, the problem is to decide whether one is a subsequence of the other.

Number of Interleaved Streams: We saw a polynomial increase in processing power of **PLog-StrNet**(I) if the maximal in-degree is increased from 1 to 2 (evidenced by ALTERNATING SEQUENCE). Is there a similar increase of computational power if the maximal in-degree grows from k to $k+1$ for any number k?

Proper Inclusions: Over the course of this chapter, we proved numerous inclusions between complexity classes. It would be interesting to determine which of these are proper inclusions.

Sorting in Streaming Networks: Is is possible to implement deterministic comparison sorting in a $O(\log n)$-size serialized access streaming network? For Lemma 6.16, we gave a network using $O(\log^2 n)$ nodes.

The last model studied in this work is a restriction of external memory algorithms to mostly linear accesses of external storage. This model has been considered by other researchers before, and we for the first time relate its computational power to the streaming model.

We showed equivalences between two pairs of models:

$$\textbf{PLog-StrSort} = \textbf{PLog-StrNet}(S) \quad \text{and} \quad \textbf{PLog-LEMA} = \textbf{PLog-StrNet}(I).$$

This independent characterization emphasizes the fact that the two classes are somehow "natural". It also shows that the key advantage of external memory algorithms over "streaming and sorting" algorithms is the ability to interleave the access to several streams.

Chapter 7

Conclusion

7.1 Computational Models

In this thesis, we have considered three very different areas of computer science: processors with multiple instruction units, peer-to-peer networks, and computations for massive datasets. Each of these areas leads to a very different notion of what computation, and particularly "efficient computation", means for them.

For processors with multiple instruction units, we gave an algorithm that generates optimal execution schedules for programs with tree-based precedence constraints. Efficiency means making optimal use of the parallelism allowed by the hardware.

For peer-to-peer networks, we gave protocols for load-balancing and finding close copies of data items. Efficiency here means the equal distribution of load (in terms of storage, computation, and network traffic) among the nodes and the ability to quickly recover from network failures.

Regarding massive data sets, we compared several computational models, and gave a number of algorithms in a particularly weak model among these. Algorithms are efficient for massive data sets if they access their data mostly sequential.

Studying these and other practically motivated computational models is interesting from the point of view of theoretical computer science for several reasons.

1. In the application areas motivating the models, there are usually many problems for which no provably good solutions exist. So the models come with a healthy supply of new algorithmic problems.

2. Solving these practically motivated problems can be useful for "real" applications, leading to a transfer of theoretical results into practice.

3. Even old problems that have been solved optimally in the RAM model become interesting again if one tries to solve them in new computational models.

4. Creating meaningful computational models that capture real systems can be a challenge in itself.

We therefore think that this is a highly rewarding approach to take in theoretical computer science, which has benefits for the applied areas of computer science, too.

7.2 Future Work

At the ends of the previous chapters, we have already given lists of possible future research problems. On a higher level, there are two kinds of research problems motivated by our research. First, one can work on the problems considered here from a theoretical point of view. Algorithms and protocols can be made simpler, more general, more efficient, and more fault-tolerant. Analyses can be simplified or strengthened. But while doing this, one should not forget about the motivation of our research.

The second direction of future research therefore is the practical evaluation of the proposed schemes. Do they really perform as well in practice as the theoretical bounds suggest? Are the constants hidden in the O-notation too large for practical purposes? It is quite possible that the asymptotic bounds we prove are not a good indication of practical behavior, since the problem size n is not that large in real applications. How do unexpected failures affect the implementation? All these questions can be answered truly only by implementing the algorithms.

And the cycle continues. Implementations lead to a better understanding and refinements of theoretical computational models, which lead to new, provably efficient algorithms, and better implementations. In the end, both computer systems and theoretical computer science benefit from this relationship.

Bibliography

[ABW02] James Abello, Adam L. Buchsbaum, and Jeffery R. Westbrook. A Functional Approach to External Graph Algorithms. *Algorithmica*, 32:437–458, 2002.

[ADADC+97] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-Performance Sorting on Networks of Workstations. In *Proceedings SIGMOD*, pages 243–254, 1997.

[ADRR03] Gagan Aggarwal, Mayur Datar, Sridhar Rajagopalan, and Matthias Ruhl. Beyond Streaming: Models for Massive Data Set Computations. Manuscript, 2003.

[Aga96] Ramesh C. Agarwal. A Super Scalar Sort Algorithm for RISC Processors. In *Proceedings SIGMOD*, pages 240–246, 1996.

[AHKV03] Micah Adler, Eran Halperin, Richard M. Karp, and Vijay V. Vazirani. A Stochastic Process on the Hypercube with Applications to Peer-to-Peer Networks. In *Proceedings STOC*, pages 575–584, June 2003.

[AKLM02] Saman Amaransinghe, David R. Karger, Walter Lee, and Vahab S. Mirrokni. A Theoretical and Practical Approach to Instruction Scheduling on Spatial Architectures. Technical Report MIT-LCS-TM-635, MIT, 2002.

[AKMV03] Pankaj Agarwal, Shankar Krishnan, Nabil Mustafa, and Suresh Venkatasubramanian. Streaming Geometric Optimization Using Graphics Hardware. Technical Report TD-5HL2NX, AT&T, 2003.

[AMS99] Noga Alon, Yossi Matias, and Mario Szegedy. The Space Complexity of Approximating the Frequency Moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[AS03] James Aspnes and Gauri Shah. Skip Graphs. In *Proceedings SODA*, pages 384–393, January 2003.

[ATI03] ATI Technologies. RADEON 9800 Series Product Website, 2003. Available online at `www.ati.com/products/radeon9800`.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(8):1116–1127, 1988.

[AVV95] Lars Arge, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-Memory Algorithms for Processing Line Segments in Geographic Information Systems. In *Proceedings ESA*, pages 295–310, September 1995.

[BCDFC02] Michael A. Bender, Richard Cole, Erik D. Demaine, and Martin Farach-Colton. Scanning and Traversing: Maintaining Data for Traversals in a Memory Hierarchy. In *Proceedings ESA*, pages 139–151, September 2002.

[Ben75] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[BG89] David Bernstein and Izidor Gertner. Scheduling expressions on a pipelined processor with a maximal delay of one cycle. *ACM Transactions on Programming Languages and Systems*, 11(1):57–66, January 1989.

[BH03] Ian Buck and Pat Hanrahan. Data Parallel Computation on Graphics Hardware. Manuscript, 2003.

[BMPW98] Sergey Brin, Rajeev Motwani, Lawrence Page, and Terry Winograd. What can you do with a Web in your Pocket? *Bulletin of the IEEE Computer Society Technical Committe on Data Engineering*, 21(2):37–47, June 1998.

[Bri95] Sergey Brin. Near neighbor search in large metric spaces. In *Proceedings VLDB*, pages 574–584, 1995.

[BWY80] J. Bentley, B. Weide, and A. Yao. Optimal expected-time algorithms for closest point problems. *ACM Transactions of Mathematical Software*, 6(4):563–580, 1980.

[BYJKS02] Ziv Bar-Yossef, T.S. Jayram, Ravi Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. In *Proceedings FOCS*, 2002.

[CEGS95] Bernard Chazelle, Herbert Edelsbrunner, Leonidas J. Guibas, and Micha Sharir. Algorithms for Bichromatic Line-Segment Problems Polyhedral Terrains. *Algorithmica*, 11(2):116–132, February 1995.

[CG72] E.G. Coffman, Jr. and R.L. Graham. Optimal sequencing for two-processor systems. *Acta Informatica*, 1:200–213, 1972.

[CGG$^+$95] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-Memory Graph Algorithms. In *Proceedings SODA*, pages 139–149, 1995.

[Cha02] Laurent Chavet. FSORT, 2002. Available online at `www.fsort.com`.

[Chu41] Alonzo Church. The calculi of lambda-conversion. *Annals of Mathematical Studies*, 6, 1941.

[Cla99] K. Clarkson. Nearest neighbor queries in metric spaces. *Discrete Computational Geometry*, 22(1):63–93, 1999.

[CM87] Stephen A. Cook and Pierre McKenzie. Problems complete for deterministic logarithmic space. *Journal of Algorithms*, 8(3):385–394, 1987.

[CNBYM01] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in Metric Spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[CP95] P. Chrétienne and C. Picouleau. Scheduling with communication delays: A survey. In P. Chrétienne, Jr. E. G. Coffman, J. K. Lenstra, and Z. Liu, editors, *Scheduling Theory and its Applications*, pages 65–90. John Wiley & Sons Ltd, 1995.

[dBSvKO00] Mark de Berg, Otfried Schwarzkopf, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer Verlag, 2000.

[DF93] Olivier Devillers and Andreas Fabri. Scalable Algorithms for Bichromatic Line Segment Intersection Problems on Coarse Grained Multicomputers. In *Proceedings WADS*, pages 277–288, August 1993.

[Dun01] John D. Dunagan. Personal communication, November 2001.

[EFKR01] Daniel W. Engels, Jon Feldman, David R. Karger, and Matthias Ruhl. Parallel Processor Scheduling with Delay Constraints. In *Proceedings ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 2001.

[Eng00] Daniel W. Engels. *Scheduling for Hardware-Software Partitioning in Embedded System Design*. PhD thesis, Massachusetts Institute of Technology, 2000.

[FCFM00] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the Sorting-Complexity of Suffix Tree Construction. *Journal of the ACM*, 47(6):987–1011, 2000.

[FFM98] Martin Farach, Paolo Ferragina, and S. Muthukrishnan. Overcoming the Memory Bottleneck in Suffix Tree Construction. In *Proceedings FOCS*, pages 174–183, November 1998.

[FL96] Lucian Finta and Zhen Liu. Single machine scheduling subject to precedence delays. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 70, 1996.

[FLMB96] Lucian Finta, Zhen Liu, Ioannis Milis, and Evripidis Bampis. Scheduling UET–UCT series-parallel graphs on two processors. *Theoretical Computer Science*, 162(2):323–340, August 1996.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. In *Proceedings FOCS*, pages 285–297, 1999.

[FR99] Jon Feldman and Matthias Ruhl. The Directed Steiner Network problem is tractable for a constant number of terminals. In *Proceedings FOCS*, October 1999.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[GK01] Michael Greenwald and Sanjeev Khanna. Space-Efficient Online Computation of Quantile Summaries. In *Proceedings SIGMOD*, pages 58–66, 2001.

[GKL03] Anupam Gupta, Robert Krauthgamer, and James R. Lee. Bounded geometries, fractals, and low-distortion embeddings. In *Proceedings FOCS*, October 2003.

[GKMV03] Sudipto Guha, Shankar Krishnan, Kamesh Mungala, and Suresh Venkatasubramanian. Application of the Two-Sided Depth Test to CSG Rendering. In *Proceedings of SIGGRAPH Symposium on Interactive 3D Graphics*, 2003.

[GKS01] Sudipto Guha, Nick Koudas, and Kyuseok Shim. Data-Streams and Histograms. In *Proceedings STOC*, pages 471–475, 2001.

[GLLR79] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A Survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[GPW00] Center for International Earth Science Information Network (CIESIN), Columbia University, International Food Policy Research Institute (IFPRI) and World Resources Institute (WRI). Gridded Population of the World (GPW), Version 2, 2000. Available online at `sedac.ciesin.org/plue/gpw`.

[GSG92] Michael T. Goodrich, Steven B. Shauck, and Sumanta Guha. Parallel Methods for Visibility and Shortest-Path Problems in Simple Polygons. *Algorithmica*, 8(5&6), 1992.

[Gut00] Jimmy Guterman. Gnutella No Ideal Fix for Napster Fiends. *The Industry Standard*, September 2000. Available online at `www.thestandard.com/article/display/0,1151,18632,00.html`.

[HEB+01] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordon Stoll, Matthew Everett, and Pat Hanrahan. WireGL: A Scalable Graphics System for Clusters. In *Proceedings of SIGGRAPH*, pages 129–140, 2001.

[HHH+02] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon T. Loo, Scott Shenker, and Ion Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In *Proceedings 2nd International Workshop on Peer-to-Peer Systems (IPTPS '02)*, pages 242–250, March 2002.

[HHN+02] Greg Humphreys, Mike Houston, Yi-Ren Ng, Randall Frank, Sean Ahern, Peter Kirchner, and James T. Klosowski. Chromium: A Stream Processing Framework for Interactive Graphics on Clusters. In *Proceedings of SIGGRAPH*, pages 693–702, 2002.

[HRR99] Monika Rauch Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan. Computing on data streams. In *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, volume 50, pages 107–118, 1999.

[IMRV97] Piotr Indyk, Rajeev Motwani, Prabhakar Raghavan, and Santosh Vempala. Locality-Preserving Hashing in Multidimensional Spaces. In *Proceedings STOC*, pages 618–625, May 1997.

[Ind00] Piotr Indyk. Stable Distributions, Pseudorandom Generators, Embeddings and Data Stream Computation. In *Proceedings FOCS*, pages 189–197, 2000.

[Ind01] Piotr Indyk. Algorithmic Aspects of Geometric Embeddings, August 2001. Available online at `theory.lcs.mit.edu/~indyk/tut.html`.

[Int00] Intel Corporation. *The IA-64 Architecture Software Developer's Manual*, January 2000.

[JKS89] Hermann Jung, Lefteris Kirousis, and Paul Spirakis. Lower bounds and efficient algorithms for multiprocessor scheduling of dags with communication delays. In *Proceedings of SPAA*, pages 254–264, 1989.

[Kar99] David R. Karger. Random Sampling in Cut, Flow, and Network Design Problems. *Mathematics of Operations Research*, 24(2):383–413, 1999.

[Kar00] David R. Karger. Minimum Cuts in Near-Linear Time. *Journal of the ACM*, 47(1):46–76, 2000.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pages 190–201, November 2000.

[KK03] Frans Kaashoek and David R. Karger. Koorde: A Simple Degree-optimal Hash Table. In *Proceedings 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, February 2003.

[KL03] Robert Krauthgamer and James R. Lee. Navigating nets: Simple algorithms for proximity search. Submitted, 2003.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent Hashing and Random Trees: Tools for Relieving Hot Spots on the World Wide Web. In *Proceedings STOC*, pages 654–663, May 1997.

[Knu98] Donald E. Knuth. *The Art of Computer Programming: Volume 3, Sorting and Searching*. Addison-Wesley, 1998.

[KR87] Richard M. Karp and Michael O. Rabin. Efficient Randomized Pattern-Matching Algorithms. *IBM Journal of Research and Development*, 31:249–260, 1987.

[KR90] Richard M. Karp and V. Ramachandran. *Parallel algorithms fro shared-memory mchines*, pages 870–941. MIT Press, Cambridge, MA, 1990.

[KR02] David R. Karger and Matthias Ruhl. Finding Nearest Neighbors in Growth-restricted Metrics. In *Proceedings ACM Symposium on Theory of Computing (STOC)*, May 2002.

[Lew98] Daniel M. Lewin. Consistent Hashing and Random Trees: Algorithms for Caching in Distributed Networks. Master's thesis, Massachusetts Institute of Technology, May 1998.

[Lop00] Asbel Lopez. The South Goes Mobile. *The Unesco Courier*, pages 65–67, July/August 2000. Available online at `www.unesco.org/courier/2000_07/uk/connex.htm`.

[Lub85] Michael Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. In *Proceedings STOC*, pages 1–10, 1985.

[Lub86] Michael Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986.

[LVV96] Jan Karel Lenstra, Marinus Veldhorst, and Bart Veltman. The complexity of scheduling trees with communication delays. *Journal of Algorithms*, 20(1):157–173, January 1996.

[Man02] Farhad Manjoo. Gnutella bandwidth bandits. *Salon.com*, August 8, 2002. Available online at `www.salon.com/tech/feature/2002/08/08/gnutella_developers/`.

[Mat93] David W. Matula. A Linear Time $2+\varepsilon$ Approximation Algorithm for Edge Connectivity. In *Proceedings SODA*, pages 500–504, January 1993.

[Mat02] Jiří Matoušek. *Lectures on discrete geometry*, volume 212 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 2002.

[MM93] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22:935–948, 1993.

[MM02] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *Proceedings 2nd International Workshop on Peer-to-Peer Systems (IPTPS '02)*, pages 53–65, March 2002.

[MNR02] Dalia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proceedings PODC*, pages 183–192, July 2002.

[MP80] J. Ian Munro and Michael S. Paterson. Selection and Sorting with Limited Storage. *Theoretical Computer Science*, 12:315–323, 1980.

[MS95] Rolf H. Möhring and Markus W. Schäffter. A simple approximation algorithm for scheduling forests with unit processing times and zero-one communication delays. Technical Report 506, Technische Universität Berlin, Germany, 1995.

[NVI03] NVIDIA Corporation. Quadro FX Product Website, 2003. Available online at `www.nvidia.com/view.asp?PAGE=quadrofx`.

[NW03] Moni Naor and Udi Wieder. Novel Architectures for P2P Applications: the Continuous-Discrete Approach. In *Proceedings SPAA*, pages 50–59, June 2003.

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray Tracing on Programmable Graphics Hardware. In *Proceedings of SIGGRAPH*, pages 703–712, 2002.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford Digital Libraries, November 1999. Available online at `http://dbpubs.stanford.edu/pub/1999-66`.

[Pic92] C. Picouleau. *Etude de problèmes les systèmes distrubés*. PhD thesis, Univ. Pierre et Madame Curie, Paris, France, 1992.

[PRR99] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32:241–280, 1999.

[PS93] Larry Palazzi and Jack Snoeyink. Counting and Reporting Red/Blue Segment Intersections. In *Proceedings WADS*, pages 530–540, August 1993.

[PST95] Serge A. Plotkin, David B. Shmoys, and Eva Tardos. Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research*, 20:257–301, 1995.

[PY88] Christos H. Papadimitriou and Mihalis Yannakakis. Optimization, Approximation, and Complexity Classes (Extended Abstract). In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 229–234, May 1988.

[RADR03] Matthias Ruhl, Gagan Aggarwal, Mayur Datar, and Sridhar Rajagopalan. Extending the Streaming Model: Sorting and Streaming Networks. In *DIMACS Working Group Meeting on Streaming Data Analysis*, March 2003.

[Raj02] Sridhar Rajagopalan. Personal communication, November 2002.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *Proceedings ACM SIGCOMM*, pages 161–172, August 2001.

[RLS+03] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load Balancing in Structured P2P Systems. In *Proceedings 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, February 2003.

[Rom00] Simon Romero. Cell Phone Surge Among World's Poor. *The New York Times*, December 19, 2000. Available online at `www.nytimes.com/2000/12/19/technology/19CELL.html`.

[SMK+01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings ACM SIGCOMM*, pages 149–160, August 2001.

[SSS93] Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-Hoeffding bounds for applications with limited independence. In *Proceedings SODA*, pages 331–340, January 1993.

[ST83] Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, June 1983.

[TdSL00] Joshua B. Tenenbaum, Vin de Silva, and John C. Langford. A Global Geometric Framework for Nonlinear Dimensionality Reduction. *Science*, 290(5500):2319–2323, December 2000. See also `isomap.stanford.edu`.

[Ten98] Joshua B. Tenenbaum. Mapping a Manifold of Perceptual Observations. In *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.

[Tex00] Texas Instruments. *TMS320C6000 Programmer's Guide*, March 2000.

[TKM+02] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Jae-Wook Lee, Paul Johnson, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro*, 22(2):25–35, March/April 2002.

[Tur36] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[Uhl91] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.

[Ull75] J. D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, June 1975.

[vEB90] P. van Emde Boas. *Machine models and simulations*, pages 1–61. MIT Press, Cambridge, MA, 1990.

[Vit01] Jeffrey Scott Vitter. External Memory Algorithms and Data Structures: Dealing with Massive Data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[VRKL96] Theodora A. Varvarigou, Vwani P. Roychowdhury, Thomas Kailath, and Eugene Lawler. Scheduling in and out forests in the presence of communication delays. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1065–1074, October 1996.

[VS94] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2-3):110–147, 1994.

[Wyl99] Jim Wyllie. SPsort: How to Sort a Terabyte Quickly. Technical report, IBM Almaden Research Center, 1999. Available online at `www.almaden.ibm.com/cs/gpfs-spsort.html`.

[Yia93] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings SODA*, pages 311–321, 1993.

[ZHS+03] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 21, November 2003.