ALGORITHMS FOR NETWORK ROUTING, MULTICASTING, SWITCHING, AND DESIGN

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Ashish Goel
July 1999

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Serge Plotkin
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Monika R. Henzinger

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Rajeev Motwani

Approved for the University Committee on Graduate Studies:

Abstract

The phenomenal success of modern communication networks is due primarily to the hands on approach adopted by the developers of the early ARPAnet and Internet. Unfortunately, this hands on approach led, at times, to the deployment of *ad hoc* strategies and algorithms which do not always perform as desired. These *ad hoc* solutions to networking problems are less likely to succeed as modern communication networks grow larger and as networking applications become increasingly diverse and demanding. There is a clear need for more principled approaches to designing algorithms and strategies for these problems.

This thesis identifies several such problems at different levels, or layers, in the network and presents algorithmic techniques to address them. The layers discussed in this thesis are the switching layer, the protocol layer, and the network design layer.

The switching layer concerns itself with moving packets from their source to their destination along a prescribed route in an efficient fashion. One major challenge at this layer is to design fast packet forwarding switches that offer strong Quality of Service (QoS) guarantees. The solutions must be fast and extremely localized; a switch in the system must decide which packets to forward without recourse to global knowledge or distributed control. We propose packet scheduling algorithms which allow the switch to work at high speeds and still offer strong quality of service guarantees. More formally, we show that a switch operating at most twice as slowly as the fastest possible switch can give the same kind of quality of service guarantees as an idealized switch which does not need to bother about efficiency. We hope that our result will significantly alter the way switches are designed. We also study the question of whether a given packet forwarding protocol results in stable behaviour on a given network. We give a partial answer to this question by characterizing networks for which all protocols are stable. As an important special case, we obtain a characterization of all networks for which FIFO is stable.

The protocol layer controls the injection and travel of packets in the network. This

control is done in several ways – by admission control of connections, by determining the route to be followed by a packet (on either a per connection or per destination basis), and by controlling the rate of injection of packets (on either a per connection or per aggregated flow basis). This layer is where most research in the field of networking has been concentrated. Solutions to problems at this layer must still be efficient; however some amount of global knowledge, centralized computation, and distributed control are acceptable. The problems in this layer are too numerous to admit a single monolithic solution. But some of the important considerations are QoS guarantees (the most prominent being end to end delay), congestion control, admission control, and application level guarantees (as opposed to packet level guarantees). At the protocol layer, this thesis first presents algorithms for the problems of delay sensitive routing and delay sensitive multicasting, which are important for reducing end to end delay. We then explore multicast admission control, followed by a look at multicast routing in directed networks. As an example of application level guarantees, we study the problem of scheduling data transfers in a network, where the user is concerned not with minimizing per packet delays but the delay for the entire file transfer. For each of the five problems discussed above, we assume the existence of a malicious adversary and do worst caae analysis of our algorithms.

The performance of the switching and protocol layers is constrained by the network topology. If the total network capacity is too low, or the total distance between two nodes in the network is very high, then the protocol and switching layers can not possibly guarantee high throughput or low end to end delays. It is therefore important to design the network carefully to satisfy predicted traffic requirements. Network design is done offline; efficiency is less of a concern. The solutions are inherently centralized and use all available global information. We study the problems of network design in both undirected and directed settings.

One major goal of this thesis has been to study algorithmic problems that have relevance to the field of computer networking, and to solve them with an eye towards the implementability of the solution. The exposition in this thesis is mostly theoretical.

Acknowledgements

This thesis owes a great debt to the encouragement, advice, and support of many individuals. First among them is my advisor, Serge Plotkin. Over the last five years he has been a source of numerous fruitful research directions. My technical conversations with Serge remain some of my most cherished moments at Stanford. His advice, while sparing, was always propitious. He has played the role of guide, advisor, mentor, and colleague during the various stages of my PhD, changing smoothly from one to the other as I matured professionally. I am particularly grateful for his generousity with his time during the first few years of my PhD when I needed it most.

It has been my good fortune to be associated with three seasoned young researchers – Monika Henzinger, Nick McKeown, and Balaji Prabhakar. Each has been a coauthor, friend, and mentor, each a source of valuable technical information and career advice. Monika taught me the importance of persistence in doing research; difficult problems unravel of their own accord in conversations with Monika. Nick and Balaji showed me that it is possible to do beautiful theory and apply it to real life problems at the same time. Balaji has been like a second advisor to me. I hope to emulate his passion for research and his breadth of knowledge.

My two internships at Bell Labs have been instrumental in shaping my interest in networking. I would like to thank K.G. Ramakrishnan (Ram) and Debasis Mitra for the opportunity of working with them. Special thanks are due to Ram for being a good landlord, a great friend, a kind boss, and a personal role model.

The Stanford algorithms group has a formidable set of students; I have been lucky in covering a large subset as coauthors. It has been a pleasure (and sometimes a lesson in humility) to work with Moses Charikar, Chandra Chekuri, Sudipto Guha, Piotr Indyk, Adam Meyerson, and Kamesh Munagala. And while I haven't written any papers with Suresh

Venkatasubramanian, Narayanan Shivakumar, or Jeffrey Oldham, my technical conversations with them have been tremendously helpful. I am truly sad at cutting the cord with the Stanford theory group.

Of all my friends, Sanjay and Suresh are by far the dearest. When I entered Stanford, I never imagined that my friendship with them could grow any deeper; I am happy that I have been proved wrong. I wish to thank my *bhaisahab* and *bhabhi*, Vikas and Reena, for their support during the last year of my PhD, and my friends, Chandra, Piotr, Shiva, and Sudipto, for countless shared pleasures.

I would like to dedicate this thesis to my parents for teaching me the importance of hard work and humility.

The research in this thesis was supported by ARO Grants DAAG55-98-1-0170, DAAG55-97-1-0221, and DAAH04-95-1-0121; by the Center for Telecommunications at Stanford; by NSF Grants CCR9304971/CCR9307045; and by a Stanford School of Engineering Fellowship.

Contents

Chapter 1

Introduction

Computer networks are now ubiquitous. The utility of a computer system is determined, to a large extent, by its network connectivity. There are few applications of computer systems which do not use the network in some fashion. This phenomenal success of modern communication networks is due largely to the hands on approach adopted by the developers of the early ARPAnet and Internet [40]. Unfortunately, this hands on approach led, at times, to the deployment of *ad hoc* strategies and algorithms which do not always perform as desired. For example, the TCP congestion control strategy required a radical redesign when the traffic over the Internet increased [75].

Ad hoc solutions to networking problems are less likely to succeed as modern communication networks grow larger and as the applications that run on top of these networks become increasingly diverse and demanding, both in terms of network resources as well as quality of service guarantees. While these problems are easiest to visualize in terms of existing IP and ATM networks, the Internet being a salient example, they are fundamental to the design of any large communication network. This thesis identifies several such problems at different levels, or layers, in the network and presents algorithmic techniques to address them. The layers discussed in this thesis are the switching layer, the protocol layer, and the network design layer; these layers are an abstraction and do not necessarily correspond to real layers in a network such as the Internet.

The switching layer is responsible for forwarding cells or packets through a network. The local network topology as well as the route that each packet must follow is known, and the problem is to move the packets from the source to the destination in an efficient and controlled fashion. The two classical problems that have concerned researchers in this field are route lookup and packet scheduling; the former involves computing the next-hop for a packet, and the latter involves computing which packet must travel forward if more than one packet is contending for a node/link in the network. This thesis studies the latter problem. Both these problems need to be solved at time scales which are too small for distributed or centralized algorithms. The solutions must be fast and extremely localized; a switch in the system must decide which packets to forward without recourse to global knowledge or distributed control. Packet scheduling algorithms go hand in hand with switch design; certain switch architectures are more suited to fast packet forwarding. Section 1.1, and later Chapters 2 and 3 discuss this problem in detail and explain the contributions of this thesis.

The protocol layer controls the injection and travel of packets in the network. This control is done in several ways – by admission control of connections, by determining the route to be followed by a packet (on either a per connection or per destination basis), and by controlling the rate of injection of packets (on either a per connection or per aggregated flow basis). The objectives of the protocol layer are to maximize throughput, minimize congestion in the network, and guarantee good Quality of Service to the end users. This layer is where most research in the field of networking has been concentrated. Solutions to problems at this layer must still be efficient; however some amount of global knowledge, centralized computation, and distributed control are acceptable. The problems in this layer are too numerous to admit a single monolithic solution. Section 1.2 and Chapters 4, 5, 6, 7 and 8 detail the problems studied and the solutions obtained.

The performance of the switching and protocol layers is constrained by the network topology. If the total network capacity is too low, or the total distance between two nodes in the network is very high, then the protocol and switching layers can not possibly guarantee high throughput or low end to end delays. It is therefore important to design the network carefully to satisfy predicted traffic requirements. Network design is done offline; efficiency is less of a concern. The solutions are inherently centralized and use all available global information. Section 1.3 and Chapters 9 and 10 present an approximation algorithms framework for a large class of network design problems.

The rest of the thesis is structured into three parts, one corresponding to each of the three layers described above.

1.1 The Switching Layer: Switch Scheduling and Network Stability

A *switch* is the packet forwarding engine in a network. Packets arrive at the input ports of a switch, and get forwarded on to the output ports. To resolve contention when several packets need to go on to the same output port, switches need to run packet forwarding protocols for arbitration. The two questions of interest here are to decide which packet forwarding protocols are good, and how they may be implemented efficiently in a network switch. Example packet forwarding protocols are FIFO (First In First Out), where the packet which arrived at the switch the earliest has preference; LIS (Longest In System), where the packet which was injected into the network the earliest has preference; SIS (Shortest In System), where the packet which was injected into the network the latest has preference; and WFQ (Weighted Fair Queueing), which runs a weighted round robin scheme over different packet flows sharing the same link.

It is perhaps best to first address the issue of implementability head on. Most of the popular switch architectures today use a crossbar fabric or a shared memory to transfer packets from the input ports to the output ports. These switches may have buffers only at the input ports, only at the output ports, or both. Accordingly, the switch is said to possess an input queueing (IQ), output queueing (OQ), or combined input output queueing (CIOQ) architecture, respectively. Ideally, a switch must serve two purposes. It must transfer packets from the input to the output side at a high rate, guaranteeing high throughput. Also, it must be able to carefully schedule packets so as to follow a packet forwarding protocol (such as WFQ) which can provide a good Quality of Service guarantee to individual packet flows. The former requires raw speed while the latter requires careful scheduling. Therefore it is not surprising that the two objectives are mutually conflicting and we must look for a tradeoff.

The concept of speedup is useful for studying the efficiency (i.e. raw bandwidth) of a switch. The speedup of a switch is the ratio of the internal data transfer rate of the switch to the speed of the external lines which are feeding and draining the switch. A switch architecture that needs high speedup is undesirable as it either requires a faster packet transfer fabric or can support lower external line rates.

In an OQ switch, up to N packets which need to go to the same output port may arrive during the same switch cycle, one at each input port. Since there are no buffers on the input side, all these N packets need to be copied to the *same* output port and hence can not be transferred in parallel. Therefore, the switch has to perform N serial memory operations in the worst case for each operation by the external lines. This implies that the speedup of an OQ switch is N in the worst case. This is particularly bad because of the high (and increasing) processor memory gap. While processor speeds have been increasing exponentially (Moore's law), memory speeds have been growing only linearly. Therefore, an architecture that uses less memory operations at the expense of a larger number of processor operations is more likely to scale as line rates and switch sizes increase. In OQ switches all the packets which need to depart on the same output port are available in a single data structure at the corresponding output buffer. This permits fine grained control of the departure times of packets from the queue, and schemes like WFQ, FIFO, and LIS become very simple to implement. Thus OQ switches have low throughput but can offer good QoS guarantees.

In an IQ switch, during one switch cycle only one packet is allowed to leave each input port, and only one is allowed to reach each output port. Thus IQ switches have a speedup of one. Karol et al. [81] showed that the throughput of IQ switches is restricted to 58%, but McKeown et al. [105] showed how this bound can be circumvented by the use of Virtual Output Queues to obtain 100% throughput at a speedup of one. But an IQ switch can not guarantee good fairness properties; in fact an IQ switch is not even *work conserving*[1].

A dream tradeoff for the above problem would be an algorithm for scheduling packet transfers across a CIOQ switch such that the speedup of the switch is small (comparable to that of an IQ switch) and the switch can offer the same QoS guarantees as an OQ switch. Further, the algorithm should be implementationally simple. Such a solution is indeed obtainable. Chapter 2 presents an algorithm that *exactly* emulates a large class of OQ switches (including WFQ, FIFO, LIS, and all other natural protocols that the authors know of) with a speedup of two. The number of processor operations needed per switch cycle is $O(N)$; thus the scalability bottleneck shifts from memory to processor. Further, the number of processor iterations required is $O(\log N)$ both in the expected sense and with high probability, given certain weak assumptions on the arrival process. An almost matching lower bound of $2 - 1/N$ is presented on the speedup of any CIOQ switch that emulates FIFO. One key idea behind the algorithms presented in Chapter 2 is to determine a stable

[1] A switch is said to be work conserving if no output line can idle as long as there is a packet in the switch destined for that output.

matching of inputs to outputs; this idea was first used by Prabhakar and McKeown [117] to demonstrate that a speedup of four suffices to emulate FIFO.

The choice of the local packet forwarding protocol (such as LIS, FIFO) can impact the global performance of the network. This phenomenon can be studied using the framework of adversarial queueing theory, proposed recently by Borodin et al. [27]. Adversarial queueing theory is a recently developed paradigm which tries to analyze the performance of network protocols under the minimal assumption that for each link in the network, the rate of injection of packets which need this resource is less than one. A packet forwarding protocol is said to be stable for a network if the queue lengths remain bounded. A protocol is said to be universally stable if it is stable for all networks; a network is said to be universally stable if all greedy protocols are stable for this network. Andrews et al. [5] showed that not all greedy protocols are universally stable; specifically, FIFO is not universally stable. Consequently, not all networks are universally stable. They proved the existence of an algorithm that, given a network, decides whether it is universally stable. But they did not give a constructive characterization of universally stable networks. Chapter 3 in this thesis presents such a characterization; an undirected network is universally stable iff no connected component has more links than nodes, and a directed network is universally stable iff each strongly connected component is a simple cycle. Borodin et al. [27] also introduced the notion of stochastic adversaries and proved some results about the stability of protocols on simple cycles against such adversaries. Chapter 3 shows that SIS (Shortest In System) is stable against a large class of stochastic adversaries for all networks.

1.2 Protocol Layer: Routing, Rate Allocation, and Admission Control

Routing in a network has twin objectives. Routes should be chosen to minimize the overall cost incurred by the network as well as maximize end user satisfaction. The simplest problem that can be formulated in this setting is the following. Given a network with a cost and a delay assigned to each link, find the cheapest route between two nodes that satisfies a given end to end delay guarantee. This problem is known to be NP-hard. An excellent heuristic due to Hassin [70] finds routes that satisfy the delay guarantee, and have cost at most a factor $1 + \epsilon$ higher than the cost of the cheapest path which satisfies the delay guarantee. The running time of this algorithm is $\tilde{O}(mn/\epsilon)$. Such algorithms are called Polynomial

Time Approximation Schemes (PTAS); the $\tilde{O}$ notation hides constants and polylogarithmic factors. Hassin's algorithm finds routes from one source to a single destination. Further it is not progressive i.e. terminating the algorithm in the middle does not result in good routes. This thesis presents a slightly different kind of PTAS. Let the delay threshold be D; the algorithm DSA in Chapter 4 finds a route that satisfies a delay threshold of no more than $D(1+\epsilon)$ and is no more expensive than the cheapest route that satisfies the delay bound D. Unlike Hassin's PTAS, DSA is progressive; the quality of the routes it produces improves as it runs longer. Hence DSA can be interrupted before it terminates, and still generate reasonable routes. Further, DSA can generate routes from one source to all destinations. The running time of DSA is $O(mL/\epsilon)$ where L is the maximum hopcount of any route produced by DSA as a solution. This is a significant saving over Hassin's algorithm in, for example, expander graphs where L is likely to be $O(\log n)$. A detailed description of DSA is presented in Chapter 4. This framework also applies to finding the cheapest path satisfying a given end to end packet drop probability threshold, under the link independence assumption.

When posed in a multicast framework, the objective of delay based routing is the following: given a cost and a delay metric defined on edges in a graph, a source s and a set S of receivers, find the cheapest tree T which contains each node in $\{s\} \cup S$ and where the delay from the source to each receiver satisfies a delay bound D. Further, this problem can be posed as an online problem, where receivers request to join the multicast group one receiver at a time, and the routing algorithm must add each receiver to the multicast tree as soon as it requests. The performance of an online algorithms is typically quantified using its competitive ratio [126]. For this problem, the competitive ratio of an algorithm would be the maximum possible value of the cost of the tree generated by the algorithm divided by the cost of the optimal tree; the maximum is taken over all instances with n nodes and k receivers, and is expressed as a function of n and k. In the offline setting this problem becomes the Shallow Light Steiner Tree problem. Chapter 7 details an approximation algorithm for this problem; the algorithm runs in quasi-polynomial time and guarantees a tree where the delay from the source to each receiver is no more than a constant factor times larger than D, and the cost is no more than $O(\log^2 n)$ times more than the cost of the cheapest tree where the delay from the sender to each receiver is D. This algorithm is not very efficient and does not extend to an online (or dynamic) setting. Chapter 5 studies the same problem under the additional assumption that the delay and the cost of an edge are identical. The problem becomes much more tractable under this assumption. It is possible to obtain an efficient online algorithm which guarantees that the cost of the multicast tree is no more than $O(\log k)$ times the cost of the cheapest tree and the delay along the tree from the source s to any receiver r is $O(d(s,r))$, where $d(s,r)$ represents the delay from s to r along the fastest route in the network. The algorithm employs some rerouting; each receiver gets rerouted at most once. Details of the algorithm, along with some simulation results, are presented in Chapter 5.

From a network service provider's point of view, it is important to satisfy as many customers as possible. Consider a capacitated network, where multicast sessions are created in an online fashion. Each group consists of several participants; the participants are revealed in an online fashion. Each multicast session requires a reserved bandwidth on each link in the tree that is used by this session. Whenever a node requests to participate in a session, this request must either be rejected outright, or a path must be constructed that joins the requesting node to the existing tree for this session and then bandwidth must be reserved along every link on this path. Since the network is capacitated (i.e. each link in the network has a finite capacity), it may not be possible to accept all requests. The goal is to accept as many requests as possible. Chapter 6 presents the first polylogarithmic competitive algorithm for this problem in the general case where requests to join a multicast tree can be interleaved in any arbitrary order; this is an extension of a result by Awerbuch et al. [13].

The above results assume an undirected network. For a directed network, finding even one good multicast tree (given a cost function on the links) is hard. This problem is known as the Directed Steiner Tree problem. Chapter 7 describes a polylogarithmic approximation algorithm for the Directed Steiner tree problem that runs in quasi-polynomial time.

Chapter 8 considers the *online ftp problem*. The goal is to service a sequence of file transfer requests given bandwidth constraints of the underlying communication network. We can control the time when a ftp request starts to be serviced and the rate at which it gets serviced. The main result of Chapter 8 is a technique that leads to algorithms that optimize several natural metrics, such as max-stretch, total flow time, max flow time, and total completion time. In particular, it is possible to achieve optimum total flow time and optimum max-stretch if the capacity of the underlying network is increased by a logarithmic factor; such capacity increase is called *resource augmentation*. This chapter also presents polynomial lower bounds on the competitive ratio of any online algorithm that attempts to minimize either max-stretch or total flow time but does not employ resource augmentation, and proves poly-logarithmic lower bounds on the resource augmentation factor necessary in order to keep the total flow time and max-stretch within a constant factor of optimum.

1.3 Network Design Layer

Perhaps the most basic network design problem is the Generalized Steiner Tree problem, where the goal is to construct the cheapest subnetwork from a given network topology such that the subnet satisfies certain connectivity constraints (specified as part of the problem input) between node pairs. More formally, given a multigraph $G(V, E)$, a cost function c defined on the edges and a set $S = \{(s_i, t_i, r_i)\}$ of (source, destination, requirement) triplets, the problem is to find the cheapest subgraph H of G such that there are r_i disjoint paths from s_i to t_i in H. This problem has been extensively studied when G is an undirected (symmetric) network [1, 134, 76], culminating in a two approximation [76]. But the corresponding problem for directed networks is much harder. Chapter 9 studies this problem in a directed network, and presents an $\tilde{O}(k^{2/3})$-approximation algorithm, where k is the number of node pairs which need to be connected. Dodis et al. [44] recently presented a strong hardness result for this problem; they showed that this problem is as hard to approximate as Label Cover. Notice that k can be as large as n^2, and therefore obtaining even an $o(n)$ approximation algorithm for this problem remains open.

Since the Generalized Steiner Tree problem has been well studied for undirected networks and good approximation algorithms are known for this problem, it is natural to turn our attention to more complicated network design problems on undirected networks. Often, a network service provider needs to design a network topology, or construct a virtual private network over an existing network topology. This network (or virtual network) must satisfy connectivity and traffic requirements, and it is desirable to minimize the cost of laying down a network. Most problems of this nature are NP-hard and also hard to approximate. Bartal [17, 18] presented a framework which allows the network to be represented as a tree, making these problems easier to solve. Specifically, he gave a randomized polynomial time algorithm that given any n point metric G, constructs a tree T such that the expected stretch (distortion) of any edge is at most $O(\log n \log \log n)$. His result has found several applications and in particular has resulted in approximation algorithms for many network design problems. However approximation algorithms based on his result are inherently randomized. Chapter 10 presents two frameworks for derandomizing the use of Bartal's algorithm in the design of approximation algorithms, and gives improved results for the case when the network can be embedded in a real normed space. The first framework is suited for graph optimization problems which can be expressed as an integer program with 0-1 variables $\bar{x}(e)$ for each edge and with an objective function expressible as $\sum_{e \in E} \bar{x}(e) \cdot c(e)$. Given an optimal solution to LP relaxation of the integer program, the solution can be mapped to a tree approximation of the graph with at most an $O(\log n \log \log n)$ blow-up in the value of the objective function. This can then be coupled with a deterministic LP rounding procedure on the tree to obtain a completely *deterministic* approximation algorithm for the original problem. This framework can be applied to the k-median problem, the group Steiner tree problem, and the minimum communication cost spanning tree problem; Chapter 10 presents details for the latter two problems. The rounding procedure on trees for the group Steiner tree problem is of independent interest. The second framework, which subsumes the first, consists of an efficient polynomial time algorithm that given a finite n point metric G, constructs $O(n \log n)$ trees and a probability distribution μ on them such that the expected stretch of any edge of G in a tree chosen according to μ is at most $O(\log n \log \log n)$. The above result establishes that finite metrics can be probabilistically approximated by a *small* number of tree metrics; the first deterministic approximation algorithms for buy-at-bulk network design [8] and vehicle routing [33] can be obtained as a consequence. The main result is obtained by a novel view of probabilistic approximation of metric spaces as a deterministic optimization problem via linear programming, using the packing framework due to Plotkin et al. [116]. This view also provides a new proof of the result in [18] that might be easier to generalize.

Part I

Switching Layer: Switch Scheduling and Network Stability

Chapter 2

Emulating Output Queued Switches Using Combined Input Output Queued Switches[1]

2.1 Introduction

A switch in a network accepts packets on several incoming lines (input ports), and forwards them to the appropriate outgoing lines (output ports). In this chapter we assume that all incoming packets are in the form of fixed length cells. Such switches are directly applicable to ATM, and can also function as the lower level mechanisms over which IP routers may be built. In this chapter, switches will be assumed to have N input ports and N output ports. An output queued switch has buffers only at the output ports, while an input queued switch has buffers only at the input ports.

Many commercial switches and routers today employ output-queueing. When a packet arrives at an output-queued (OQ) switch, it is immediately placed in a queue that is dedicated to its outgoing line, where it waits until departing from the switch. This approach is known to maximize the throughput of the switch: so long as no input or output is oversubscribed, the switch is able to support the traffic and the occupancies of queues remain bounded. Furthermore, by carefully scheduling the time a packet is placed onto the

[1]This chapter presents joint work with S. Chuang, N. McKeown, and B. Prabhakar, and was published as [39].

outgoing line, a switch or router can control the packet's latency, and hence provide quality-of-service (QoS) guarantees. But output queueing is impractical for switches with high line rates and/or with a large number of ports, since the fabric and memory of an $N \times N$ switch must run N times as fast as the line rate. Unfortunately, at high line rates, memories with sufficient bandwidth are simply not available.

On the other hand, the fabric and the memory of an input queued (IQ) switch need only run as fast as the line rate. This makes input queueing very appealing for switches with fast line rates, or with a large number of ports. For this reason, the highest performance switches and routers use input-queued crossbar switches [110, 106]. But IQ switches can suffer from head-of-line (HOL) blocking, which can have a severe effect on throughput. It is well-known that if each input maintains a single FIFO, then HOL blocking can limit the throughput to just 58.6% [81]. McKeown et al. [105] showed how Virtual Output Queues (VOQs) can be used to obtain a 100% throughput in the long run from an input queued switch.

One method that has been proposed to reduce HOL blocking is to increase the "speedup" of a switch. A switch with a speedup of s can remove up to s packets from each input and deliver up to s packets to each output within a time slot, where a time slot is the time between packet arrivals at input ports. Hence, an OQ switch has a speedup of N while an IQ switch has a speedup of one. For values of s between 1 and N, packets need to be buffered at the inputs before switching as well as at the outputs after switching. Such an architecture is called a combined input and output queued (CIOQ) switch.

Both analytical and simulation studies of a CIOQ switch which maintains a single FIFO queue at each input have been conducted for various values of speedup [73, 67, 108, 38]. A common conclusion of these studies is that with a speedup of 4 or 5 one can achieve about 99% throughput when arrivals are independent and identically distributed at each input, and the distribution of packet destinations is uniform across the outputs. Whereas these studies consider average delay (and simplistic input traffic patterns), they make no guarantees about the delay of individual packets. This is particularly important if a switch or router is to offer QoS guarantees.

A well-designed network switch should perform predictably in the face of all types of arrival process and allow the delay of individual packets to be controlled. Hence the approach presented in this chapter is quite different: rather than find values of speedup that work well on average, or with simplistic and unrealistic traffic models, find the minimum speedup such that a CIOQ switch behaves identically to an OQ switch for all types of traffic. (Here, "behave identically" means that when the same inputs are applied to both the OQ switch and to the CIOQ switch, the corresponding output processes from the two switches are completely indistinguishable.) This approach was first formulated in the recent work of Prabhakar and McKeown [117]. They showed that a CIOQ switch with a speedup of four can behave identically to a FIFO OQ switch for arbitrary input traffic patterns and switch sizes. In this sense, this chapter builds upon and extends the results in [117]. A number of researchers have recently considered various aspects of the speedup problem, most notably [35] which obtains packet delay bounds and [95] which finds sufficient conditions for maximizing throughput through work conservation and mimicking of output queueing. Charny observed that speedup could also be used to provide QoS guarantees [34].

This chapter shows that a CIOQ switch with a speedup of two can behave identically to an OQ switch. The result holds for switches with an arbitrary number of ports, and for any traffic arrival pattern. It is also found to be true for a broad class of widely used output link scheduling algorithms such as weighted fair queueing, strict priorities, and FIFO. This chapter introduces some specific scheduling algorithms that achieve this result. Further, a speedup of $2-1/N$ is both necessary and sufficient for a CIOQ switch to behave identically to a FIFO OQ switch.

It is worth briefly considering the implications of this result. It demonstrates that it is possible to emulate an OQ switch using buffer memory operating at only twice the speed of the external line. Previously, an OQ switch could only be implemented with memories operating at N times the speed of the external line. However, the advantages do not come for free. In essence, the memory bandwidth is reduced at the expense of a fast cell scheduling algorithm that is required to configure the crossbar. The scheduling algorithms presented here are complex, the best known-to-date having a running-time complexity of N. (The issue of implementation complexity is discussed in some detail in Section 2.6). This means that it is not yet practicable to emulate fast OQ switches with a large number of ports. While some strategies are presented in this paper, this is a topic for further research.

2.2 Background

Consider the single stage $N \times N$ switch shown in Figure 2.1. Throughout this chapter it is assumed that packets begin to arrive at the switch from time 0, the switch having

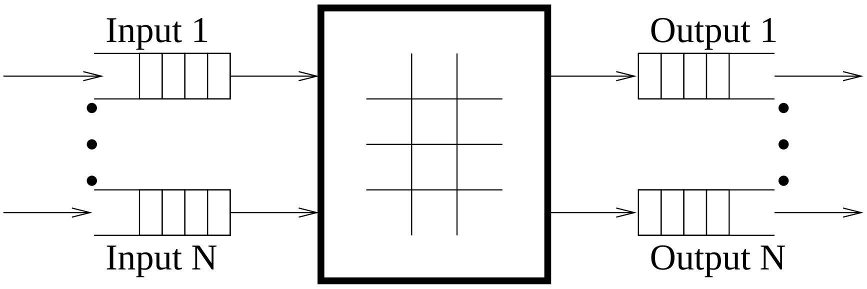

Figure 2.1: A General Combined Input and Output Queued (CIOQ) switch.

been empty before that time. Although packets arriving to the switch or router may have variable length, it is assumed that they are treated internally as fixed length "cells". This is common practice in high performance LAN switches and routers; variable length packets are segmented into cells as they arrive, carried across the switch as cells, and reassembled back into packets again before they depart [106, 110]. The arrival time between cells is considered as the basic time unit and referred to as a time slot. The switch is said to have a speedup of s, for $1 \leq s \leq N$, if it can remove up to s cells from each input and transfer at most s cells to each output in a time slot. A speedup of s requires the switch fabric to run s times as fast as the input or output line rate. For $1 < s < N$, buffering is required both at the inputs and at the outputs, and leads to a combined input and output queued (CIOQ) architecture. The following is the problem addressed in this chapter.

The speedup problem:

Determine the smallest value of s and an appropriate cell scheduling algorithm A that

1. allows a CIOQ switch to exactly mimic the performance of an output-queued switch (in a sense that will be made precise),
2. achieves this for any arbitrary input traffic pattern, and
3. is independent of switch size.

In an OQ switch, arriving cells are immediately forwarded to their corresponding outputs. This (a) ensures that the switch is work-conserving, i.e. an output never idles so long as there is a cell destined for it in the system, and (b) allows the departure of cells to be scheduled to meet latency constraints. It is important that any solution of the speedup problem possess these two desirable features; that is, a CIOQ switch must behave identically to an OQ switch in the following sense:

Definition 2.1 Identical Behavior: *A CIOQ switch is said to behave identically to an OQ switch if, given identical inputs, the departure time of every cell from both switches is identical.*

As a benchmark with which to compare a CIOQ switch, assume there exists a shadow OQ switch that is fed the same input traffic pattern as the CIOQ switch. The goal is to arrange for each cell to depart from the CIOQ switch at exactly the same time as its counterpart cell departs from the OQ switch. In the CIOQ switch, the sequence in which cells are transferred from their input queues to the output queue is determined by a scheduling algorithm. In each time slot, the scheduling algorithm matches each non-empty input with at most one output and, conversely, each output is matched with at most one input. The matching is used to configure the crossbar fabric before cells are transferred from the input side to the output side. A CIOQ switch with a speedup of s is able to make s such transfers during each time slot.

Selecting the appropriate scheduling algorithm is the key to achieving identical behavior between the CIOQ switch and its shadow OQ switch.

2.2.1 Push-in Queues

Throughout this chapter, repeated use will be made of the concept of "push-in" queues. Similar to a discrete-event queue, a push-in queue is one in which an arriving cell is inserted at an arbitrary location in the queue based on some criterion. For example, each cell may carry with it a departure time, and is placed in the queue ahead of all cells with a later departure time, yet behind cells with an earlier departure time. The only property that defines a push-in queue is that once placed in the queue, cells may not switch places with other cells. In other words, their relative ordering remains unchanged. In general, there are two types of push-in queues: (1) "Push-In First-Out" (PIFO) queues, in which arriving cells are placed at an arbitrary location, and the cell at the head of the queue is always the next to depart. PIFO queues are quite general – for example, a WFQ scheduling discipline operating at an output queued switch is a special case of a PIFO queue. (2) "Push-In Arbitrary-Out" (PIAO) queues, in which cells are removed from the queue in an arbitrary order. i.e. it is not necessarily the case that the next cell to depart is the one currently at the head of the queue.

It is assumed that each input of the CIOQ switch maintains a queue, which can be thought of as an ordered set of cells waiting at the input port. In general, the CIOQ switches considered here can all be described using PIAO input queues. Many orderings of the cells are possible - each ordering leading to a different switch scheduling algorithm, as we shall see.

Each output maintains a queue for the cells waiting to depart from the switch. In addition, each output also maintains an output priority list: an ordered list of cells at the inputs waiting to be transferred to this particular output. The output priority list is drawn in the order in which the cells would depart from the OQ switch being emulated (i.e. the shadow OQ switch). This priority list will depend on the queueing policy followed by the OQ switch (FIFO, WFQ, strict priorities etc.).

2.2.2 Definitions

Definition 2.2 Time to Leave: *The "time to leave" for cell c, TL(c), is the time slot at which c will leave the shadow OQ switch. Note that it is possible for TL(c) to increase. This happens if new cells arrive to the switch, destined for c's output, and have a higher priority than c*[2]*.*

Definition 2.3 Output Cushion: *At any time, the "output cushion" of cell c, OC(c), is the number of cells waiting in the output buffer at cell c's output port with a smaller time to leave value than cell c.*

Notice that if a cell is still on the input side and has a small (or zero) output cushion, the scheduling algorithm must urgently deliver the cell to its output so that it may depart on time. Since the switch is work-conserving, a cell's output cushion decreases by one during every time slot.

Definition 2.4 Input Thread: *At any time, the "input thread of" cell c, IT(c), is the number of cells ahead of cell c in its input priority list..*

[2]Of course, TL(c) is also the time slot in which c must leave from the CIOQ switch for identical behavior to be achieved.

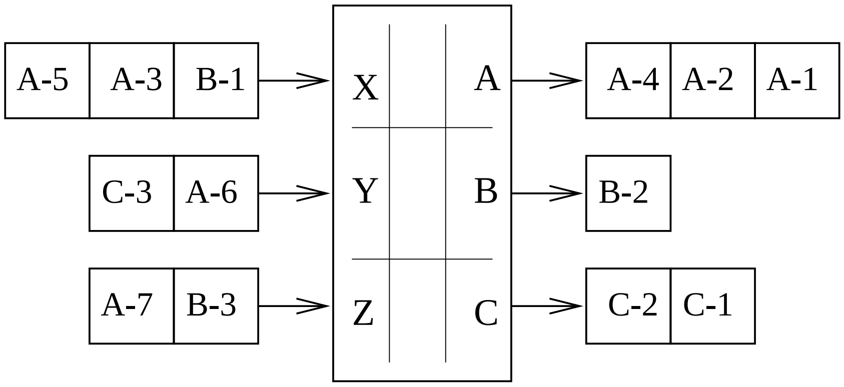

Figure 2.2: A snapshot of a CIOQ switch.

In other words, $\mathrm{IT}(c)$ represents the number of cells currently at the input that need to be transferred to their outputs more urgently than cell c. A cell's input thread is decremented only when a cell ahead of it is transferred from the input, and is possibly incremented by newly arriving cells. Notice that it would be undesirable for a cell to simultaneously have a large input thread and a small output cushion – the cells ahead of it at the input may prevent it from reaching its output before its time to leave. This motivates our definition of slackness.

Definition 2.5 Slackness: *At any time, the "slackness" of cell c, $L(c)$, equals the output cushion of cell c minus its input thread.*

Slackness is a measure of how large a cell's output cushion is with respect to its input thread. If a cell's slackness is small, then it urgently needs to be transferred to its output. Conversely, if a cell has a large slackness, then it may languish at the input without fear of missing its time to leave.

Figure 2.2 shows a snapshot of a CIOQ switch with a number of cells waiting at its inputs and outputs. For convenience we assume the time the snapshot was taken to be 1. Let P-t denote a cell that, in the shadow switch, will depart from output port P at time t. Consider, for example, the cell c denoted in the figure by A-3. For the CIOQ switch to mimic the shadow OQ switch, the cell must depart from port A at time 3. Its input thread is 1, since B-1 is the only cell ahead of A-3 in the input priority list. Its output cushion is 2, since out of the three cells queued at A's output buffer, only two cells, A-1 and A-2 will depart before it. Further, the slackness of cell A-3 is given by $2 - 1 = 1$.

2.2.3 The general structure of CIOQ scheduling algorithms

This chapter mostly deals with CIOQ switches that have a speedup of two. Hence, each time slot can be broken into into four phases:

1. **The Arrival Phase:** All arrivals of new cells to the input ports take place during this phase.

2. **The First Scheduling Phase:** The scheduling algorithm selects cells to transfer from inputs to outputs, and then transfers them across the crossbar.

3. **The Departure Phase:** All departures of cells from the output ports take place during this phase.

4. **The Second Scheduling Phase:** Again, the scheduling algorithm selects cells to transfer from inputs to outputs and transfers them across the crossbar.

The order in which the four phases occur is not crucial to the algorithms presented in this paper; however this ordering makes the proofs simpler.

A matching of input ports to output ports is a (not necessarily maximal) set of cells waiting on the input side such that all these cells can be sent across the crossbar in a single transfer (i.e. are free of input and output contention). During each scheduling phase the scheduler finds a stable matching between the input ports and the output ports.

Definition 2.6 Stable Matching: *A matching of input ports to output ports is said to be stable if for each cell c waiting in an input queue, one of the following holds:*

1. *Cell c is part of the matching, i.e. c will be transferred from the input side to the output side during this phase.*

2. *A cell that is ahead of c in its input priority list is part of the matching.*

3. *A cell that is ahead of c in its output priority list is part of the matching.*

Notice that conditions 2 and 3 above may be simultaneously satisfied, but condition 1 excludes the other two. The conditions for a stable matching can be achieved using the so-called stable marriage problem. Solutions to the stable marriage problem are called stable matchings and were first studied by Gale and Shapely [49]. They gave an algorithm that finds a stable matching in at most M iterations, where M is the sum of the lengths of all the input priority lists.

The specification of the scheduling algorithm for a CIOQ switch is almost complete: the only thing that remains is to specify how the input queues are maintained. Different ways of maintaining the input queues result in different scheduling algorithms. In fact, the various scheduling algorithms presented later differ only in the ordering of their input queues. For reasons that will become apparent, this chapter only deals with a particular class of orderings, which is defined as follows.

Definition 2.7 PIAO Input Queue Ordering: *When a cell arrives, it is given a priority number which dictates its position in the queue. i.e. a cell with priority number x is placed at location $(x+1)$ from the head of the list. A cell is placed in an input priority list according to the following rules:*

1. *Arriving cells are placed at (or, "push-in" to) an arbitrary location in the queue,*
2. *The relative ordering of cells in the queue does not change once cells are in the queue, i.e. cells in the queue cannot switch places, and*
3. *Cells may be selected to depart from the queue from any location.*

Now all that is needed to complete the description of the scheduling algorithms is an insertion policy which determines where an arriving cell gets placed in its input queue.

On the output side, the CIOQ switch keeps track of the time to leave of each waiting cell. During each time slot the cell that departs from an output and is placed onto the outgoing line is the one with the smallest time to leave. For the CIOQ switch to successfully mimic the shadow OQ switch, each cell must cross over to the output side before it is time for the cell to leave.

Even before the algorithm description is complete, it is clear that the algorithm must maintain a large amount of state. More importantly, the algorithm must keep track of a large amount of global state, taking into account information about the queues at all the inputs and all the outputs. Section 2.6 discusses the information complexity of these algorithms, and the difficulty of implementing them at high speed.

2.3 Necessity and Sufficiency of a Speedup of $2 - 1/N$

Having defined speedup, the next natural question is the following: what is the minimum required speedup, s, for a CIOQ switch to emulate an OQ switch. The following theorem answers this question.

Theorem 2.8 *[Necessity] An $N \times N$ CIOQ switch needs a speedup of at least $2 - 1/N$ to exactly emulate a FIFO OQ switch.*

Remark: Since FIFO is a special case of a variety of output queueing disciplines (Weighted Fair Queueing, Strict Priorities etc.), the lower bound applies to these queueing disciplines as well.

The proof is deferred to Appendix A.

Theorem 2.9 *[Sufficiency] An $N \times N$ CIOQ switch with a speedup of $2 - 1/N$ can exactly emulate a FIFO OQ switch. An $N \times N$ CIOQ switch with a speedup of 2 can exactly emulate any OQ switch that implements a PIFO queueing discipline.*

The proof for FIFO is based on the insertion policy "Last In Highest Priority" (LIHP) and is presented in Appendix A. The proof for arbitrary queueing disciplines is presented in the next two sections.

2.4 A Simple Input Queue Insertion Policy for a Speedup of two

This section presents a simple insertion policy that, with a speedup of two, mimics an OQ switch with a FIFO queueing discipline. This insertion policy is termed "Critical Cells First" (CCF)[3]. CCF inserts an arriving cell as far from the head of its input queue as possible, such that the input thread of the cell is not larger than its output cushion. More formally:

The CCF Insertion Policy: Suppose cell c arrives at input port P. Let x be the output cushion of c. Insert cell c into the $(x+1)$-th position from the front of the input queue at P. Hence, upon arrival cell c has a slackness of zero. If the size of this list is less

[3]Recall that to specify a scheduling algorithm for a CIOQ switch, all that is needed is an insertion policy for the input queues.

than x cells, then place c at the end of the input priority list at P. Hence, in this case, c has a positive slackness.

A consequence of this is that a cell's slackness is non-negative upon arrival. The intuition behind this insertion policy is that a cell with a small output cushion needs to leave soon (i.e. it is "more critical"), and therefore needs to be delivered to its output sooner than a cell with a larger output cushion. In other words, a cell with a large output cushion can safely reside further from the head of its input queue.

The rest of this section is devoted to proving that CCF, with a speedup of two, mimics an OQ switch. A key property of the CCF algorithm is established first: a cell never has a negative slackness, i.e. a cell's input thread never exceeds its output cushion. It is then shown how this ensures that a cell always reaches the output side in time to leave.

Lemma 2.10 *The slackness, L, of a cell c waiting on the input side is non-decreasing from time slot to time slot.*

Proof. Let the slackness of c be l at the beginning of a time slot. During the arrival phase, the input thread of c can increase by at most one because an arriving cell might be inserted ahead of c in its input priority list. During the departure phase, the output cushion of c decreases by one. If c is scheduled in any one of the scheduling phases, then it is delivered to its output and is no longer of concern. Otherwise, during each of the two scheduling phases, either the input thread of c decreases by one, or the output cushion of c increases by one (by the property of stable matchings). Therefore the slackness of c increases by at least one during each scheduling phase. Counting the changes in each of the four phases (arrival, departure, and two scheduling phases), it follows that the slackness of cell c can not decrease from time slot to time slot. $\square$

Remark: Because the slackness of an arriving cell is non-negative, it follows from the above lemma that the slackness of a cell is always non-negative.

Theorem 2.11 *Regardless of the incoming traffic pattern, a CIOQ switch that uses CCF with a speedup of 2 exactly mimics a FIFO OQ switch.*

Proof. Suppose that the CIOQ switch has successfully mimicked the OQ switch up until time slot $t-1$, and consider the beginning (first phase) of time slot t. The proof is by showing that any cell reaching its time to leave is either: (1) already at the output side of the switch, or (2) will be transferred to the output during time slot t. From Lemma 2.10, it follows that a cell always has a non-negative slackness. Therefore, when a cell reaches its time to leave (i.e. its output cushion has reached zero), the cell's input thread must also equal zero. This means either: (1) that the cell is already at its output, and may depart on time, or (2) that the cell is simultaneously at the head of its input priority list (because its input thread is zero), and at the head of its output priority list (because it has reached its time to leave). In this case, the stable matching algorithm is guaranteed to transfer it to its output during the time slot, and therefore the cell departs on time. □

2.5 Providing QoS guarantees

As pointed out in the introduction, the goal of this work is to control the delay of cells in a CIOQ switch in the same way that is possible in an OQ switch. The previous section concerned itself only with the emulation of an OQ switch in which cells depart in FIFO order. This section proves that with a speedup of two, CCF successfully emulates an OQ switch that uses the broad class of PIFO (Push-In First-Out) queueing policies; a class that includes widely-used queueing policies such as WFQ [42] and Strict Priority queueing. Notice that with an arbitrary PIFO policy, the TL of a cell never decreases, but may increase as a result of arrival of higher priority cells.

The description of CCF remains unchanged; however the shadow OQ switch used to calculate the output cushion and the output priority lists is not the FIFO switch but the OQ switch being emulated.

Theorem 2.12 *Regardless of the incoming traffic pattern, a CIOQ switch that uses CCF with a speedup of 2 exactly mimics an OQ switch that adheres to a PIFO queueing policy.*

The proof of this theorem is identical to that of Theorem 2.11.

2.5.1 Universally Stable Protocols

Andrews et al. [5] proved that the protocols LIS (Longest In System) and SIS (Shortest In System) are universally stable. It turns out that both these policies are push-in policies. Hence, CIOQ switches running at a speedup of 2 can be used to ensure stability in the network. A more detailed description of universal stability and the protocols LIS and SIS is presented in the next chapter.

2.6 Towards making CCF practical

CCF as presented above suffers from two main disadvantages. First, the stable matching that is needed in each scheduling phase can take as many as $\Omega(N^2)$ iterations to compute. Second, the algorithm has a high information complexity – CCF needs to know both the output cushion and time to leave of each cell at the inputs. This information is not locally available at each input, but depends on the state of all the switch outputs. These disadvantages are addressed in this section. The Delay Till Critical (DTC) strategy reduces the number of iterations needed to compute a stable matching to N (from $\Omega(N^2)$); and the Group By Virtual Output Queue (GBVOQ) algorithm can emulate FIFO and WFQ OQ switches without using global information. Unfortunately, the two solutions do not compose i.e. either the number of iterations to or the information complexity can be reduced, but not both at the same time. Such a composition is left as an open problem.

2.6.1 The Delay Till Critical (DTC) strategy:

The "Delay Till Critical" strategy is as follows: During each scheduling phase, mark as "active" all cells with a slackness of zero, and mark all other cells inactive. The stable matching algorithm now considers only active cells. Intuitively, cells with zero slackness are the most critical and should be considered for immediate transfer across the fabric. Since the slackness of a cell can never become negative, CCF combined with DTC strategy can emulate any OQ switch that follows a PIFO queueing policy.

It remains to show that this simple strategy reduces the number of iterations required to compute a stable matching from $\Omega(N^2)$ to $O(N)$. Before proving this fact, it is worthwhile to examine the problem that DTC attempts to remove. At any time instant, define the dependency graph G to be a directed graph with a vertex corresponding to each active cell that is waiting on the input side of the CIOQ switch. Let X and Y be two cells waiting at the input side. There is a directed edge from X to Y if and only if cell Y is ahead of X either in an input queue or in an output priority list. Clearly two cells have to share either the same input port or the same output port if there is to be an edge between them. If CCF is used as defined in Section 3, there may be cycles in this dependency graph. These cycles are the main cause of inefficiency in finding stable matchings, and the DTC strategy is designed to remove these cycles.

Lemma 2.13 *If DTC is used in conjunction with CCF then, during any scheduling phase, the dependency graph is acyclic.*

Proof. The proof is by contradiction. Assume there does exist a cycle in the dependency graph on active cells. Pick a smallest cycle in this graph. If there is an edge from cell X to cell Y, then Y must be ahead of X either in the input queue ordering or in the output queue ordering. Call the edge an "input" edge in the former case and an "output" edge in the latter; ambiguities are resolved arbitrarily. The smallest cycle must have alternating input and output edges, because two successive input or output edges could be collapsed into one resulting in a smaller cycle. If there is an output edge from X to Y, then the output cushion of Y is at most as large as that of X. But X and Y are both active, and the input thread of an active cell must equal its output cushion. Therefore, the input thread of Y is no larger than the input thread of X. Also, if there is an input edge from X to Y then the input thread of Y must be strictly smaller than that of X; that is, X appears in Y's input thread. The smallest cycle must have at least two edges, as there can be no self loops in the dependency graph. Consequently, the cycle must contain at least one input edge. But this implies that during one traversal of the cycle, the input thread of the cell where the traversal started must be larger than the input thread of the cell where it ends. Since the start and end are at the same cell, this implies that the input thread of this cell must be less than itself. This is clearly impossible. Hence the assumption that there exists a cycle in the graph cannot be true, and the lemma is proved. □

Now consider how the first cell in the match is found. Since there are no cycles in G, there has to be at least one "sink" (i.e. a cell with no outgoing edges). Let cell c be the sink. Since there are no active cells ahead of c in either its input queue or its output priority list, cell c has to be part of any stable matching of active cells. Hence c is guaranteed to be transferred to the output side, and therefore all cells which have the same input or output port as c can be removed from G; they clearly can not be part of the match. The resulting graph is again acyclic, and the above procedure can be repeated $N-1$ more times to obtain a stable matching. Notice that each iteration of the above N iteration algorithm is quite straightforward.

The next subsection addresses the second disadvantage of CCF, i.e. that of high information complexity.

2.6.2 The Group By Virtual Output Queue (GBVOQ) algorithm

With CCF, the stable matching algorithm needs to calculate both the time to leave and output cushion of each cell in the input queues. These quantities require centralized information about the state of all the queues in the system, making CCF (as described) unsuitable for a distributed implementation. However, for emulating a FIFO OQ switch, incoming cells can be grouped into Virtual Output Queues resulting in reduced information complexity. The algorithm, GBVOQ, which achieves this bound is described below.

At each input, GBVOQ maintains a single priority list as before, as well as a VOQ for each output port. All cells belong to a VOQ and the single input priority list. When a new cell arrives, it is always placed at the tail of the corresponding VOQ. If the VOQ was empty, the new cell is placed at the head of the input priority list. If, on the other hand, the VOQ is non-empty, the new cell is inserted in the input priority list just behind the last cell belonging to the same VOQ. i.e. all cells that are in the same VOQ occupy contiguous positions in the input priority list. Therefore, to make a scheduling decision, it is sufficient to just keep track of the relative priority ordering of VOQs. Since there are at most N VOQs at an input port in a FIFO switch, the size of the input priority list is bounded. Since GBVOQ assigns a non-negative slackness to an incoming cell, a CIOQ switch that uses GBVOQ with a speedup of two successfully emulates a FIFO OQ switch.

GBVOQ has several desirable properties. First, the decision of where an incoming cell needs to be inserted is much simpler for GBVOQ than CCF – each input port can maintain its local priority queue without any access to global information. Second, during the stable matching phase, to determine which of two cells has a higher output priority, it is sufficient to compare the arrival timestamps of the two cells; the cell which arrived earlier will have a smaller TL (and hence a higher output priority) because of the First In First Out property.

Like CCF, GBVOQ can be used in conjunction with the DTC strategy to reduce the number of iterations needed to compute a stable matching. In fact, DTC is made much simpler when used in conjunction with GBVOQ because of the following property: if the cell at the head of a VOQ is marked inactive during a scheduling phase, the entire VOQ can be marked inactive, reducing the number of cells that need to be marked active/inactive. However, and unfortunately, to determine which VOQs need to be marked active, the scheduling algorithm again needs access to global state, namely the output cushion of each cell at the head of a VOQ. Finding a solution which simultaneously has low information complexity and low number of iterations is an interesting open problem.

GBVOQ with per-flow VOQs can emulate WFQ with low information complexity. However, even a bound of N^2 on the number of iterations for this case is not known.

2.7 Extensions to Multicasting

This section briefly discusses the speedup required for the exact emulation of an OQ switch by a CIOQ switch when the incoming traffic is multicast. Suppose that the maximum "fanout" of an incoming cell (i.e., the maximum number of outputs it wishes to go to) is F. Then replicating this cell when it arrives at the input results in at most F new arrivals in each time slot at that input. (This contrasts with the unicast case, in which at most 1 new cell arrives). Thus, to extend Theorem 2.11 to include multicast traffic, up to F new arrivals per time slot per input port must be allowed for. During a departure phase, the slackness of a cell can go down by at most 1. During an arrival phase, the slackness of a cell already in the system can go down by at most F since a new cell with fanout F may get inserted ahead of it. Recall, from the proof of Lemma 2.10, that the slackness of a cell must go up by at least 1 during each scheduling phase. Therefore an equivalent of Lemma 2.10 (and hence Theorem 2.11) holds for multicast if the speedup is $F + 1$.

2.8 Conclusions

With the continued demand for faster and faster switches, it is increasingly difficult to implement switches that use output queueing or centralized shared memory. Before long, it will become impractical to build the highest performance switches and routers using these techniques. It has been argued for some time that most of the advantages of output-queuing (OQ) can be achieved using combined input and output queueing (CIOQ). While this has been argued for very specific, benign traffic patterns there has always been a suspicion that the advantages would diminish in a more realistic operating environment.

This chapter showed that a CIOQ switch with a speedup of just two can behave identically to an OQ switch which employs a wide variety of packet scheduling algorithms, such as WFQ, strict priorities, etc. Perhaps more importantly, this was shown to be true for any traffic arrival pattern and for arbitrary switch sizes.

However, while this result makes possible a significant reduction in memory bandwidth, it comes at the expense of a scheduling algorithm. The scheduling algorithm is required to configure the crossbar, operating at least twice as fast as cells can arrive. While the algorithms that are described here are quite simple, they have a running time complexity of at least N making them unsuitable for fast switches with a large number or ports. But the result does not preclude algorithms that are more readily implemented at higher speed; this is an important area for future research.

Chapter 3

Stability of Networks and Protocols in the Adversarial Model

3.1 The Adversarial Model for Packet Injection

In traditional queueing theory, the source which generates network traffic is typically assumed to be stochastic. Adversarial Queueing Theory developed out of a recent need for more robust models for these sources. The growing complexity of network traffic makes it increasingly unrealistic to model traffic as, say, a Poisson stream. It is therefore desirable to have a general robust framework which makes as few assumptions about the network traffic as possible. Such a framework was developed by Borodin et al. [27] in the context of packet routing and several very interesting results in this model were proved by Andrews et al. [5]. We refer the reader to [27, 5] for a more thorough motivation of the adversarial model. In this model, packets are injected into the network by an adversary. To keep things simple, it is assumed that the route of each packet is given along with the packet itself. The adversary is limited in the following way: over any w consecutive steps, the adversary can inject at most $\lceil w(1-\epsilon)\rceil$ path packings into the network, where a path packing is an edge disjoint collection of simple paths in a network. Each path can be looked upon as a route that a packet needs to follow. A single path packing may, therefore, correspond to many packets. The parameter w is called the burst size, and $1-\epsilon$ is the injection rate. Such adversaries are called (w, ϵ)-adversaries.

Once injected, packets follow their routes one edge at a time till they reach their destination. Only one packet may cross an edge during one time step, so a protocol is needed to decide which packet should cross an edge if more than one packets are waiting. A protocol is said to be greedy if it always forwards a packet on an edge if there are packets waiting to cross that edge. Any non-greedy protocol can be simulated by a (centralized) greedy protocol. Distributed greedy protocols are the simplest to implement and reason about, and we limit ourselves to such protocols.

Definition 3.1 *A protocol P is stable on network G (or equivalently, the pair (P, G) is stable) if for all (w, ϵ)-adversaries, the maximum number of packets in the system, as well as the maximum delay a packet experiences, is bounded.*

Definition 3.2 *A protocol P is universally stable if (P, G) is stable for all networks G. Similarly, a network G is universally stable if (P, G) is stable for all greedy protocols P.*

Some common greedy protocols are defined below. To define a greedy protocol, it is sufficient to specify how contention is resolved when more than one packets want to traverse the same edge.

FIFO: First In First Out.

SIS: Shortest In System – the packet which was injected last wins.

LIS: Longest In System – the packet which was injected first wins.

NTG: Nearest To Go – the packet nearest its final destination wins.

FTG: Furthest to Go – the packet furthest away from its final destination wins.

One of the most interesting results of Andrews et al. is that several natural greedy protocols for packet routing are not universally stable. Specifically, FIFO is not stable! But fortunately, several natural greedy protocols are stable on all networks – for example, LIS and SIS, which give priority to the oldest and latest packet, respectively. Andrews et al. show a similar dichotomy for networks. They show that all greedy protocols are stable on rings, trees, and DAGs. But they do not characterize all universally stable networks. We give a complete characterization of such networks in Section 3.2. Specifically, we show that a connected undirected graph is universally stable if and only if it has at most one cycle. For directed graphs, we give a simple decision procedure to determine universal stability.

Borodin et al. [27] introduced the notion of stochastic adversaries. The number of packet packings injected by a stochastic adversary during time step i is X_i, where X_is are i.i.d. random variables with mean less than 1. However, the path packings chosen by the adversary need not come from a probability distribution. Hence stochastic adversaries are more powerful than traditional source models, where packet injection is probabilistic for each source-destination pair. Borodin et al. give some results for stability of protocols against stochastic adversaries on directed cycles. In Section 3.3, SIS is shown to be stable on all networks against all 0-1 adversaries.

Subsequent to our work, Gamarnik [50] gave an equivalence between a fluid model and the adversarial model of stability. One of the motivations behind his work was to characterize universally stable networks. However, that work did not result in such a characterization. Recently, Gamarnik [51] extended the results of this chapter to the case of an undirected network where an edge can be crossed only in one direction during one time step.

3.2 Universal Stability of Networks

Andrews et al. [5] prove that not all graphs are universally stable, by giving a counter-example. They also attempt to characterize graphs that are universally stable. They claim that for undirected graphs, the property of universal stability of a graph is closed under minor inclusion. Using the results of Robertson and Seymour [122], it follows that universal stability can be decided in polynomial time. While this approach does offer a lot of insight, it is unsatisfactory for several reasons. First, the results of Robertson and Seymour are non-constructive. We are guaranteed that there are only finitely many minor-minimal graphs which are not universally stable, but there is no general technique for finding these forbidden minors. Moreover, the results of Andrews et al. hold only for undirected graphs. Andrews et al. do observe that large classes of commonly used networks are unstable.

A simple and constructive characterization of universally stable graphs is given in this section. An undirected connected graph G with n vertices is shown to be universally stable iff it has at most n edges (Theorem 3.14). For directed graphs, the set of forbidden minors is given (Theorem 3.11) along with a simple decision procedure (Theorem 3.13).

Definition 3.3 *For any directed graph G, the line graph $L(G)$ of G is defined in the natural way: the edge set of G forms the vertex set of $L(G)$ and there is an edge in $L(G)$ from e_1 to e_2 if, in the graph G, the head of e_1 coincides with the tail of e_2.*

Lemma 3.4 *If digraphs G_1 and G_2 are universally stable, then so is any graph G formed by joining them with edges that go only from G_1 to G_2.*

Proof. Assume that the adversary we are working against has rate $1-\epsilon$ and burst size w. Since G_1 is stable, any packets originating in G_1 get out of G_1 within T_1 time steps, where T_1 is some function of w and ϵ. Some of these packets may then enter G_2. Now, during a time window T_2 units long, any new packets that enter G_2 must have been introduced during $T_1 + T_2$ contiguous units. The number of path packings introduced during this interval can be at most $(T_1 + T_2 + w)(1-\epsilon)$. Choose any ϵ' such that $0 < \epsilon' < \epsilon$. Now, set $T_2 = (T_1 + w)(1-\epsilon)/(\epsilon - \epsilon')$. During any window of size $t \geq T_2$ at most $t(1-\epsilon') - (T_2(\epsilon - \epsilon') - (T_1 + w)(1-\epsilon)) = t(1-\epsilon')$ path packings are introduced into G_2. This implies that these packets could have been introduced by an adversary of burst size T_2 and rate ϵ'. But by definition of universal stability, G_2 is stable against such an adversary. Therefore, the traversal time for a packet as well as the queue lengths are bounded in G. □
Lemma 3.4 implies that all acyclic digraphs are universally stable. 3.4.

Corollary 3.5 *A digraph is universally stable iff all of its strongly connected components are universally stable.*

We now focus on graphs that are strongly connected. The following result by Andrews et al. [5] will be useful:

Lemma 3.6 *(Andrews et al.[5]) The directed cycle on any number of vertices is universally stable.*

The same result holds for the undirected cycle, as the undirected cycle can be modeled as two disconnected directed cycles. The next logical question to ask is the following: Does there exist a strongly connected digraph that is more complicated than a cycle and is still universally stable? The answer is no. Consider the following two simple digraphs, H_1 and H_2, also drawn in Figure 3.1.

$\underline{H_1}$: There are just two vertices u and v with two parallel edges e_1 and e_2 from u to v and an edge f from v to u.

$\underline{H_2}$: There are three vertices u, v and w with the following edges: $e_1 = (uv)$, $e_2 = (vu)$, $f_1 = (uw)$, $f_2 = (wu)$.

Neither of these two simple graphs is universally stable. For now, we ignore the fact that the paths specified have to be simple. We will come back to it later in the section.

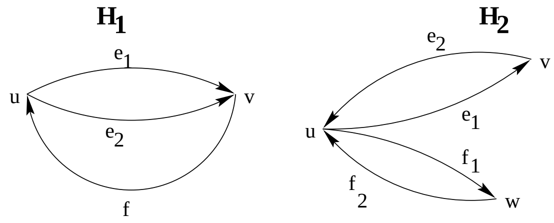

Figure 3.1: The graphs H_1 and H_2.

Lemma 3.7 *H_1 is not universally stable.*

Proof. To prove instability, the adversary and the greedy protocol are allowed to work in collusion. Suppose there are s packets waiting to cross edge f, where s is larger than some large enough constant. The adversary introduces packets at a rate $r < 1$, and operates in four rounds. At the end of the fourth round, there will be more than s packets waiting to cross edge f. Our proof goes along the lines of a similar proof in [5] for a larger graph.

The first round lasts for s steps: the adversary introduces sr packets of the form (fe_2) and delays all of these. At the beginning of the second round, there are sr packets of type (fe_2) waiting at edge f. During the second round (duration sr), sr^2 packets of type (e_2) and another sr^2 packets of type (fe_1) are introduced. Further, all these packets are delayed. At the beginning of round 3, there are sr^2 packets each of type (e_2) and (fe_1). The adversary now introduces (in sr^2 steps) sr^3 packets each of types (e_2) and (e_1f), delaying each of these. At the beginning of the fourth round, there are sr^3 packets each of type (e_2) and (e_1f). During the fourth round (duration sr^3) the adversary introduces sr^4 packets each of types (e_2f) and (e_1). The packets of type (e_2f) are delayed, whereas packets of type (e_1) are allowed to pass through, delaying packets of type (e_1f) from the previous round. Notice that this is the only step when the greedy protocol differs from LIS. At the end of the fourth round, there are sr^4 packets each of types (e_1f) and (e_2f) waiting. All these packets need to cross edge f.

During the above procedure, the number of packets waiting at edge f went from s to $2sr^4$. If $r > 0.5^{1/4} \approx 0.84$, then the number of packets goes up, and the adversary can use this procedure over and again to make the number of packets in the system unbounded.

In the above proof, the number of packets at the end of round 4 will actually be $2sr^4$ minus some constant. But since s was chosen to be larger than a large enough constant, this is not a problem. To initially generate a constant number of packets at edge f, we can attach a large acyclic graph at u, where the newly added acyclic portion is used to

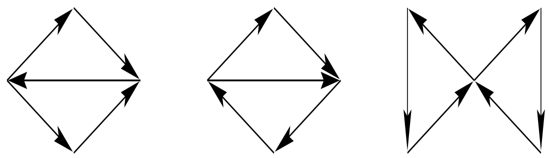

Figure 3.2: The forbidden minors for a universally stable digraph.

generate the initial packets. The new graph (ie. H_1 with an acyclic graph attached) is not universally stable. Corollary 3.5 now implies that H_1 is not universally stable. □

Simple modifications to the above proof result in the following corollary.

Corollary 3.8 *Any graph obtained by replacing the edges e_1, e_2 and f in H_1 by disjoint directed paths is not universally stable. In particular, such a graph is not stable for FIFO.*

Lemma 3.9 *H_2 is not universally stable.*

Proof. The proof is similar to that for Lemma 3.7. The adversary again operates in four rounds. Suppose there are s packets waiting to cross edge e_1. In the first round (duration s), the adversary introduces sr packets of type $(e_1e_2f_1f_2)$ and delays them. In the second round (duration sr), the adversary injects sr^2 packets each of types (e_2) and (f_2) and delays them. In the third round (duration sr^2), the adversary injects sr^3 packets each of types (e_2e_1) and (f_2) and again delays the newly introduced packets. In the fourth round (duration sr^3), the adversary introduces sr^4 packets each of types (e_2) and (f_2e_1). During this round, the adversary delays packets of type (e_2e_1) from the previous round and (f_2e_1) from the current round. Therefore, at the end of the fourth round, there are $2sr^4$ packets that need to cross edge e_1. □

Corollary 3.10 *Any graph obtained by replacing the edges e_1, e_2, f_1 and f_2 in H_2 by disjoint directed paths is not universally stable.*

Any strongly connected digraph must either be a cycle, or it must consist of at least two cycles which either share an edge or a vertex. If these cycles share an edge, this digraph would be unstable by Lemma 3.7. If they share a vertex, this digraph would be unstable by Lemma 3.9. Imposing the restriction that each packet must follow a simple path, the forbidden minors for universally stable digraphs are given in Figure 3.2.

Theorem 3.11 *A digraph G is universally stable iff it does not contain any of the forbidden minors drawn in Figure 3.2.*

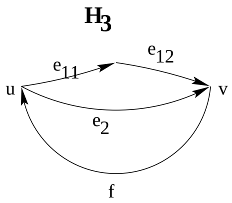

Figure 3.3: The graph H_3.

The above theorem does not indicate whether there exist any natural protocols for which the above characterization holds. We rectify that in the following theorem.

Theorem 3.12 *The pair (FIFO, G), where G is a digraph is stable iff G does not contain any of the forbidden minors drawn in Figure 3.2.*

The proof of the theorem is a simple modification of the proofs of Lemmas 3.7 and 3.9. The main differences are briefly sketched below for completeness.

Proof. We first prove that the graph H_3 drawn in Figure 3.3 is unstable for FIFO. We use essentially the same construction as in the proof of Lemma 3.7 with the following differences. Wherever the edge e_1 occurs in the construction, we replace it by the path $e_{11}e_{12}$. The duration of the third round is increased to $sr^2(1+r)$. and that of the fourth round to sr^4. During the fourth round, instead of introducing paths of the form e_1 we introduce paths of the form e_{12}. And there is an extra fifth round of duration sr^5 during which paths of the form f are introduced. It is straightforward to check that the graph is unstable if $r^6(2+r)/(1+r) > 1$ which holds for any $r \geq (2/3)^{1/6}$.

The above construction takes care of the first of the two minors in Figure 3.2. The third minor can be proved to be unstable for FIFO by modifying the proof of Lemma 3.9 using similar ideas to the above sketch. □

There is an even simpler procedure to check universal stability of a digraph G. Define a feasible line graph $L_F(G)$ of a digraph G as follows: the edge set of G forms the vertex set of $L(G)$ and there is an edge in $L(G)$ from e_1 to e_2 if, in the graph G, the head of e_1 coincides with the tail of e_2, and e_1 and e_2 do not form a directed cycle. The last condition captures the restriction that a packet cannot traverse the same edge in two directions consecutively.

Theorem 3.13 *A digraph G is universally stable iff in its feasible line graph $L_F(G)$, all strongly connected components are simple cycles.*

The above theorems completely characterize universally stable graphs, and give an efficient ($O(mn)$) procedure for deciding universal stability.

For undirected graphs, the characterization is even simpler.

Theorem 3.14 *A connected undirected graph with n vertices is universally stable iff it has no more than n edges.*

3.3 Against Stochastic Adversaries

Borodin et al. [27] introduce the notion of stochastic adversaries. They show that several commonly used protocols such as LIS and FTG are stable on a directed cycle against stochastic adversaries with exponentially vanishing tails[1]. We show that SIS is stable on all networks against all 0-1 adversaries.

Definition 3.15 *A 0-1 stochastic adversary generates 0 path packings with probability ϵ and 1 path packing with probability $1 - \epsilon$ during any single time step, for some constant $\epsilon > 0$.*

We first define a non-greedy protocol, SIS-NG, which takes at least as much time to route any packet as SIS. SIS-NG is then shown to be stable against a 0-1 stochastic adversary.

Definition 3.16 *The non greedy protocol SIS-NG does the following: a packet p is blocked at each edge in its path till the number of path packings introduced after p becomes less than the duration for which p has been waiting at this edge.*

Informally, SIS-NG assumes that each packet gets blocked once at each edge along its path by each of the path packings introduced after it.

Lemma 3.17 *Given the same sequence of requests, SIS takes at most as long to route any packet as SIS-NG.*

Theorem 3.18 *SIS-NG is stable against all 0-1 stochastic adversaries on arbitrary graphs.*

[1] These results were originally reported for adversaries with bounded variance, but the restriction that the input process have an exponential tail is essential.

Proof. In this proof, the protocol will implicitly be assumed to be SIS-NG, and the mean number of path packings injected by the adversary during a single step will be $1-\epsilon, 0 < \epsilon < 1$.

Let $P_i(\alpha)$ represent the probability of α path packings being generated by the 0-1 adversary between the time when packet p was introduced and when it crosses the ith edge on its path. Clearly, $P_0(0) = 1$, and $P_0(\alpha) = 0$ for all $\alpha > 0$. If there are β packets in the system when p crosses its ith edge, and there were α packets when p crossed its $(i-1)$th edge, then during the previous β steps, exactly $\beta - \alpha$ path packings must have been introduced, and no packet must have been introduced during the current step. This argument can be formalized to obtain the following recursive equation:

$$P_i(\beta) = \sum_{\alpha \le \beta} \binom{\beta}{\alpha} P_{i-1}(\alpha)\epsilon^{\alpha+1}(1-\epsilon)^{\beta-\alpha}$$

The solution to the above equation is $P_i(\beta) = \epsilon^i(1-\epsilon^i)^\beta$. This can be verified inductively. Clearly, this solution is valid for $i = 1$. Suppose the solution is valid for all $i \le j$. Now,

$$\begin{aligned}
P_{j+1}(\beta) &= \sum_{\alpha \le \beta} \binom{\beta}{\alpha} \epsilon^j(1-\epsilon^j)^\alpha \epsilon^{\alpha+1}(1-\epsilon)^{\beta-\alpha} \\
&= \epsilon^{j+1} \sum_{\alpha \le \beta} \binom{\beta}{\alpha} \left(\epsilon(1-\epsilon^j)\right)^\alpha (1-\epsilon)^{\beta-\alpha} \\
&= \epsilon^{j+1} \left(1 - \epsilon + \epsilon(1-\epsilon^j)\right)^\beta \\
&= \epsilon^{j+1}(1-\epsilon^{j+1})^\beta
\end{aligned}$$

, which completes the inductive proof. Let d be the maximum distance that a packet p needs to cover (clearly, $d \le m$). Let β_d represent the number of path packings introduced after p but before p completes its journey. Expected delay for packet p is at most

$$\begin{aligned}
\mathbf{E}(d + d\beta_d) &= d\left(1 + \sum_{\beta \ge 0} \beta\epsilon^d(1-\epsilon^d)^\beta\right) \\
&= d\left(1 + \frac{1-\epsilon^d}{\epsilon^d}\right) \\
&= \frac{d}{\epsilon^d}
\end{aligned}$$

This completes the proof of stability of SIS-NG against 0-1 adversaries. □

Corollary 3.19 *If SIS-NG is used against 0-1 adversaries, the distributions of queue lengths and packet delays have exponentially vanishing tails.*

Proof. Prob(delay $> d + kd$) $\leq$ Prob($\beta_d > k$) $\leq \epsilon^d \sum_{\beta > k}(1-\epsilon^d)^\beta = (1-\epsilon^d)^{k+1}$, which proves that packet delays have exponentially vanishing tails.

Prob(number of packets $> dkm + dm$) $<$ Prob(oldest packet is at least $dk + d$ units old). Now, there can be at most m packets per time step, and incrementing k by one corresponds to d time steps. Using the bound on delay obtained earlier in this proof, the above probability is at most $md \sum_{\beta > k}(1-\epsilon^d)^\beta = (dm/\epsilon^d)(1-\epsilon^d)^k$, which has an exponentially vanishing tail. □

The expected delay is exponential in d. But for a large class of networks, d can be logarithmic in the network size, which would result in polynomial delays and queue sizes. Also, by Lemma 3.17, all the above results hold for SIS as well.

Part II

Protocol Layer: Routing, Rate Allocation, and Admission Control

Chapter 4

Delay Sensitive Unicast Routing[1]

4.1 Introduction

QoS-sensitive routing is the first step towards achieving end-to-end QoS guarantees for connections with stringent QoS requirements. The goal of QoS-sensitive routing is to find a path that satisfies multiple QoS constraints while achieving overall network resource efficiency. We consider the important case where there is a single QoS constraint, namely end-to-end delay. The problem formulation and the algorithmic framework presented are quite general; they apply to any source-based or sink-based routing protocol.

The QoS parameters for a connection comprise quality-related parameters, such as packet loss ratio and end-to-end delay as well as capacity-related parameters such as peak rate, acceptable burst rate and mean rate. Capacity related parameters are easy to handle using link filters; pretend that any link which does not have the required spare capacity does not exist in the network. End-to-end delay is by far the most important of the remaining QoS constraints. In addition, Administrative Weight (AW) is the hook available to the network administrator to indicate how expensive each link is, in terms of utilization of network resources. The challenge of QoS-sensitive routing then becomes to find a path that satisfies the end to end delay requirement while simultaneously maximizing the utilization of network resources, i.e., finding the path that is the cheapest in terms of the AWs among all paths that satisfy the delay requirement. This problem is NP-hard [54]. Nevertheless, routing subject to a delay constraint remains a desirable feature despite its intractability.

[1]This chapter represents joint work with D. Kataria, D. Logothetis, and K.G.Ramakrishnan [61].

In this chapter we present an efficient algorithm that lends itself to real-time as well as off-line constrained shortest path computations. Our algorithm is based on delay-scaling and unlike the length-scaling algorithm due to Hassin [70], it generates paths from one source to multiple destinations. Since this problem is NP-hard, any efficient algorithm can only produce approximate solutions. Both our algorithm and that of Hassin are approximate. The approximation results due to discretization errors. The solution produced by length-scaling algorithm of Hassin can only guarantee optimality in path length up to an error tolerance of ϵ but satisfies the delay threshold; our algorithm produces optimal shortest paths but can violate path delay threshold by at most ϵ, where ϵ is a user-given error tolerance. The running times of both algorithms are proportional to $\frac{1}{\epsilon}$, giving an accuracy/computation-time tradeoff. Such algorithms are called Fully Polynomial Time Approximation Schemes (FPTAS). Both algorithms invoke the corresponding pseudo-polynomial dynamic programming algorithm on scaled input. The details of running times are discussed in Section 3.

Another useful contribution of this chapter is to demonstrate quantitatively the advantage of using both delay and AW simultaneously in path calculation as opposed to using only AW to compute the shortest path and subsequently checking whether this path also satisfies the delay constraints.

Our algorithm finds explicit routes from a source to multiple destinations. In IP networks, traditional link-state protocols, such as OSPF, do not support explicit routes. Explicit routing can be supported through use of Label Switched Paths(LSPs). In ATM networks, explicit routing is already well supported by PNNI. For explicit routing, it is beneficial to pre-calculate some routes and keep them in a route cache in order to avoid real time route computation overheads. While our new algorithm works well for both real time computation and pre-calculation of paths, it is particularly well suited to the latter.

Even though we concentrate on end-to-end delay in this chapter, the same techniques work when we need to find the cheapest (in terms of AW) path with no more than a given packet loss ratio, if we make the link independence assumption. The link independence assumption is that the packet drops on different links are independent events. Since the packet drop probabilities are small for all interesting applications, the packet drop probability along a path is approximated well by the sum of the packet drop probabilities on all the links along the path. Thus the problem of satisfying a packet loss threshold becomes combinatorially equivalent to that of satisfying a certain delay threshold. In general, we can substitute any additive metric for delay.

The outline of the chapter is as follows. In Section 4.2 we provide the relevant background on the subject of QoS-sensitive routing and address the various issues involved. Section 4.3 presents the enhanced constrained shortest path algorithms using the length scaling and delay scaling methods and also quantifies the advantage of using delay scaling in the path selection. Section 4.4 details the conclusions of our study.

4.2 Background

Typical routing algorithms such as Dijkstra's shortest path routing algorithm provide an optimal solution if the required routing decision is based on a single metric (QoS parameter). For the case of multiple metrics (QoS parameters), the routing problem becomes NP-hard [54]. An example of such a problem is cost and delay requirements, where the end-to-end cost has to be lower than the target cost value, and the end-to-end delay has to be lower than the target delay value. Since the number of QoS parameters that can be specified in a data network includes delay, delay jitter, loss probability, cost and bandwidth, the problem of finding a path that satisfies all these constraints is also NP-hard. As mentioned in [131] the only feasible combinations are bandwidth and one of the four (delay, loss probability, cost and delay jitter). This is because bandwidth is a link constraint. A link constraint is a restriction on the use of links for path selection, that is, bandwidth on a link must be greater than or equal to that required by the connection. Link constraints are easy to handle because they can be used to remove links from the topology for path computation. It is the path constraints that make the routing problem intractable. As was mentioned before, even a routing problem with only one path constraint is NP-hard. Under certain assumptions regarding input traffic profiles and scheduling disciplines it is possible to relate bandwidth, delay, delay jitter and loss parameters with each other [100] and reduce the multi-metric QoS routing problem to a single metric shortest path. Further, in many applications only one of the path constraints is of interest, typically end-to-end delay or packet loss ratio. However, whenever cost (or administrative weight) is considered it is harder to associate it with any of the QoS parameters and in the best case we obtain a two metric shortest path problem. To support QoS-sensitive routing, therefore, new routing paradigms are needed.

One possible approach is to define a function which generates a single metric from multiple QoS parameters and the cost. For example, the routing algorithm could be based on a linear combination of weighted QoS parameters and the cost. This reduces the routing problem to a single-criterion shortest path problem. However, the results of such a routing algorithm are sensitive to the selected weights. Also there is no guarantee that a feasible path will always be found quickly, if one exists, since the approach is enumerative.

Another approach is to find a possible path by calculating the AW-based shortest path from the network resource utilization point of view. The obtained path has to be further evaluated as to whether it can meet the other QoS requirements. If such a path is found, then that path can be regarded as the best path for both the users and networks. If such a path is not found, then a criterion other than AW is used, to find a path, even if the path is not the AW-based shortest path. The advantage of this approach is that only one optimization criterion is used in the routing algorithm every time. However, it cannot always be guaranteed that a path satisfying the all the QoS requirements will always be found.

A third approach, applicable to off-line path calculations, is to incorporate the delay constraints in the optimization problem that determines the AWs and then solve, for example, a large integer program. This again becomes computationally expensive for large dense networks.

The approach advocated by us is based on solving the constrained shortest path problem by first scaling and discretizing the problem to obtain a bounded domain for the delay values, and then using dynamic programming procedures. This method always converges and produces near optimal solutions efficiently (see Section 4.3).

The PNNI routing and signaling protocol specification [107] describes a framework for implementing path selection algorithms in two stages. The first stage involves finding a path from a list of precalculated routes, and the second stage involves finding a path by on-demand calculation excluding the blocked links detected during the first stage from whole known link state information. We expect IP labeled routing to work in a similar fashion.

The path precalculation scheme initially prepares a list of paths from source to all destinations all of which satisfy fixed values of multiple QoS criterion. Several such lists are prepared each corresponding to different fixed values for multiple QoS parameters. During connection-setup phase, an appropriate path is chosen from among the applicable lists. Since the path precalculation does not need any on-demand path calculation at that time, it can reduce connection set-up delay significantly. If no appropriate paths can be found among precalculated paths, then on-demand path calculation scheme needs to be used to find another possible path. The on-demand path calculation scheme is performed among reduced sets of possible paths after pruning unnecessary links, considering both user QoS requirements and the actual blocked link information. Although more processing is involved than the path precalculation scheme, on-demand path calculation helps to reduce the call blocking probability.

Since path precalculation reduces connection set-up delay and on-demand path calculation reduces connection blocking probability a combination of precalculation and on-demand calculation scheme seems to be the right solution. The delay-scaling algorithm we propose in this chapter is well suited for this hybrid approach.

4.3 Path Selection Algorithms

In this section we present four constrained shortest path algorithms that we will investigate; two pseudo-polynomial dynamic programming algorithms DAD and DAL, a length scaling algorithm, and a delay-scaling algorithm. The first three of these algorithms were presented by Hassin [70]. The delay-scaling algorithm is a new proposal. We also compare the time complexity of all these algorithms. We first formulate the constrained shortest path problem. Let $G = (V, E)$ be a directed graph with a vertex (or node) set $V = \{1, 2, \cdots, n\}$ and an edge (or link) set E with $|E| = m$. With each edge $(i, j) \in E$ we associate a length l_{ij} and a delay t_{ij}. Our objective is to find the shortest path (i.e., the path with the smallest length) from a source node s to a destination node t, such that the path's total delay does not exceed a given threshold T. In another variant, the objective is to find the shortest path satisfying the delay constraint for all other nodes in G. The problem in its most general form, is NP-hard [54]. However, under certain assumptions i.e, integer values for lengths and delays and upper bounds for delay or length of any path T or L, respectively, pseudo-polynomial algorithms can be constructed.

The dynamic algorithm gives an exact solution and runs, as any exact algorithm must, unless P = NP, in pseudo-polynomial time. The length scaling and delay scaling algorithms run in polynomial time and give approximate solutions.

In the rest of the chapter, we will assume, without any loss of generality, that the source node s is 1, and the destination node t is n. Also, we will interchangeably use the terms *link* and *edge* to refer to the edges of the graph.

4.3.1 The dynamic programming algorithm

Let $L(i,t)$ the length of a shortest path from source node 1 to node i with delay less than or equal to t. The dynamic programming algorithm that iterates on delay, $DAD(G,T)$, takes as input a graph G with both length and delay metrics specified on edges, and a delay constraint value T. It then builds up the table $L[v,t]$, where $L[v,t]$ is the minimum distance from 1 to v with delay no more than t.

ALGORITHM DAD

Inputs:

Graph G with edge weights l_{ij} and delays t_{ij}

Delay threshold T

Outputs:

Table $L(v,T), 1 \le v \le n$. The entry $L(v,T)$ is the length of the shortest path from 1 to v whose delay is no more than T.

Table $P(v,T); 1 \le v \le n;$. The entry $P(v,T)$ encodes the shortest path from 1 to v, whose delay is no more than T

1. Initialize $L(1,t) = 0, \qquad t = 0, \cdots, T$
2. Initialize $L(j,0) = \infty, \qquad j = 2, \cdots, n$
3. Compute $L(j,t) = \min\left\{L(j,t-1), \min_{k|t_{kj} \le t \text{ and} (k,j)\in E} \{L(k,t-t_{kj}) + l_{kj}\}\right\}$ $j = 2, \cdots, N$, $t = 1, \cdots, T$

The time complexity of the algorithm is $O(mT)$, where m is the number of links in the network. Another recursion is also possible. Let $D(i,l)$ denote the delay of "fastest" path whose length is at most l. The dynamic programming algorithm that iterates on length, $DAL(G,L)$ takes as input a graph G with both length and delay metrics specified on edges, and a length constraint value L. It then builds up the table $D[v,l]$, where $D[v,l]$ is the minimum delay from 1 to v with length no more than l.

ALGORITHM DAL

Inputs:

Graph G with edge weights l_{ij} and delays t_{ij}

Length threshold L

Outputs:

Table $D(v, l), 1 \leq v \leq n; \quad 1 \leq l \leq L$. The entry $D(v, l)$ is the delay of fastest path from 1 to v whose length is no more than l.

Table $P(v, L); 1 \leq v \leq n$. The entry $P(v, L)$ encodes the fastest path from 1 to v, whose length is no more than L

1. Initialize $D(s, l) = 0, \qquad l = 0, \cdots, L$
2. Initialize $D(j, 0) = \infty, \qquad j = 2, \cdots, N$
3. Compute $D(j, l) = \min\left\{D(j, l-1), \min_{k|l_{kj} \leq l \text{ and} (k,j) \in E} \{D(k, l - l_{kj}) + t_{kj}\}\right\}$ $j = 2, \cdots, N$, $l = 1, \cdots, L$

The time complexity of this algorithm is $O(mL)$.

Both DAD and DAL also keep track of the paths $P(v, t)$ and $P(v, l)$, respectively, which achieve the values $L(v, t)$ and $D(v, l)$ provided these values are finite. As can be seen from their corresponding time complexities both algorithms do not scale well either with the delay or the length constraint. The next two algorithms attempt to fix this problem at the expense of some loss of accuracy that in most practical cases will be insignificant.

4.3.2 The length scaling algorithm

The length-scaling algorithm, $LSA(G, T, d, \epsilon)$, was presented in [70]. It finds a path from the source 1 to a single destination d with the following two properties:

1. It satisfies the delay constraint T.
2. The length of the path is at most $(1+\epsilon)$ times the length of the shortest path satisfying the delay constraint.

The running time of this algorithm is $O(\frac{mn}{\epsilon} \log \log \frac{\text{UB}}{\text{LB}})$, where UB and LB are the lengths of the fastest and the shortest path from s to d, respectively. Since this algorithm finds paths for a single source destination pair, the time required to compute paths to all destinations from a single source is $O(\frac{mn^2}{\epsilon} \log \log \frac{\text{UB}}{\text{LB}})$. This scales well with both the delay and length values. However, this running time is much worse than the running time of the delay scaling algorithm described next.

We make some modifications to the length scaling algorithm described in [70], which result in a slightly improved running time. The length scaling algorithm is completely described in [70] and will not be repeated here.

4.3.3 The delay-scaling algorithm

The delay scaling algorithm attempts to balance the tradeoff between scaling down the delay requirement and the inaccuracy introduced by rounding scaled delays to integers. As in the length scaling case, whenever the delay metric is scaled by multiplying the delay of each link by a factor of $\frac{\tau}{T}$ and then truncating to an integer, a shortest path of delay no greater than τ is equivalent to delay of no greater than $T(1 + (n-1)/\tau)$ on the original graph. The introduction of the $(n-1)/\tau$ is due to the rounding of up to $n-1$ links.

The delay-scaling algorithm, $DSA(G, T, \epsilon)$ finds an approximate solution to the constrained shortest path problem. More specifically, it returns a tree rooted at node 1 with the following properties:

1. The length of the path from 1 to any other node v on this tree is no more than the shortest path from 1 to v satisfying the delay constraint T.

2. The delay of the path from 1 to v is no more than $T(1+\epsilon)$.

A τ-scaling G_τ of graph G is obtained by multiplying the delay on each edge in G by τ/T and then truncating the new delay to an integer. The corresponding delay constraint for the new scaled graph is τ. The algorithm $DSA(G, T, \epsilon)$ works as follows:

ALGORITHM DSA

Inputs:

Graph G with edge weights l_{ij} and delays t_{ij}

Delay constraint T

Relative error ϵ.

Outputs:

Table $L(v), 1 \leq v \leq n$;. The entry $L(v)$ is the length of the shortest path from 1 to v whose delay is no more than $T(1+\epsilon)$.

Table $P(v)$; $1 \leq v \leq n$. The entry $P[v]$ encodes the actual shortest path, from 1 to v, whose delay is no more than $T(1+\epsilon)$ and whose length is given by $L[v]$.

1. Set $\tau = \tau_0$ (a small number $<< T$)

2. Call $DAD(G_\tau, \tau)$ to compute the $L(v, \tau)$ and $P(v, \tau)$ table

3. Compute the delays in the original graph G, for each of paths $P(v, \tau)$ and store the delays in $D(v)$.

4. If $\exists v \in G$: such that $D(v) > T(1+\epsilon)$ and $2\tau < \frac{n}{\epsilon}$ set $\tau = 2\tau$ and go to Step 2.

The table calculated during the last iteration of DSA contains the constrained shortest paths.

Theorem 4.1 *The worst case complexity of the algorithm is* $\frac{4mD}{\epsilon}$*, where D is the maximum length in hops of any optimal constrained shortest path in the graph G.*

Proof. The worst case running time of $DSA(G, T, \epsilon)$ is $O(mD/\epsilon)$, where D is the maximum number of hops in any optimal constrained shortest path in the graph satisfying the delay constraint $T(1+\epsilon)$. To obtain this result we need the following lemma, which we state without proof, since it is straightforward.

Lemma 4.2 *Let $P_\tau(v)$ denote the shortest path from the source to any vertex v in G_τ satisfying the time constraint τ. The length of $P_\tau(v)$ is no more than the length of the shortest path between s and v satisfying the time constraint T in G. Further, the delay of $P_\tau(v)$ in G is at most $T(1+D/\tau)$ where D is the maximum length, in hops, of any shortest path satisfying the delay constraint τ.*

If $\tau \geq D/\epsilon$ then the delay of $P_\tau(v)$ in G is at most $T(1+\epsilon)$, and the algorithm $DSA(G, T, s, \epsilon)$ terminates during step 3. Thus, the running time of DSA is at most of order

$$m\tau_0 + 2 \cdot m\tau_0 + \ldots + 2^k \cdot m\tau_0$$

, where $k = \left\lceil \log \frac{D/\epsilon}{\tau_0} \right\rceil$. Summing up, the running time is at most of order

$$4 \cdot 2^{k-1}\tau_0 < 4mD/\epsilon$$

. Notice that the algorithm guarantees a running time of $O(mD/\epsilon)$ without needing to know D. The constant hidden inside the O-notation is just 4. $\square$

Clearly $D < n$. The worst case running time of this algorithm is independent of the delay/length values and hence, this algorithm scales well as these values change. Further, a single invocation of DSA finds the constrained shortest path from the source to all destinations.

Several observations are worth noting about DSA:

- The time complexity result has a value D, but the algorithm does not need to know this value.

- As mentioned earlier, a slightly weaker time complexity result is $\frac{4mn}{\epsilon}$, but in most real-life cases when the paths have a hop-count restriction (for example 5), a tighter complexity result is order(s) of magnitude better than the weaker result.

- DSA has a monotonicity property, that is useful in practice: the accuracy of the solution monotonically improves with successive iterations. DSA can be terminated in the middle and still produce reasonable answers. For instance, when there is a call processing time budget during call setup, we can terminate the execution when the time runs out, and still have a reasonably accurate solution. This monotonicity property does not hold for the dynamic programming algorithms, DAD and DAL.

- DSA is strongly polynomial, i.e. the running time bound does not depend on the actual delay and length values on the edges. Neither of DAD, DAL, and LSA are strongly polynomial. In fact DAD and DAL are pseudo polynomial. Strongly polynomial algorithms are traditionally considered more robust.

4.3.4 Comparison of Algorithms and Computational Results

In this section we compare the algorithms in terms of executions times and performance gain. We measured their execution times on a 150MHz SGI Unix machine. For all our experiments the error tolerance parameter ϵ was set to 5%.

For our experiments we used random graphs of a given node size n and connectivity p which we defined as the probability of a link existence between nodes i and j. The link length and delay are also random numbers uniformly distributed in the open interval [1, 1000]. We also scale the mean of the delay distribution by a factor $0 < r < \infty$ to achieve the effect of light/heavy network load. *The execution times for the length scaling algorithm are for a single-source destination pair. All other running times are for one source, all destinations.*

We study the efficacy of the enhanced path selection algorithms, by comparing their performance with the performance of the classical Dijkstra unconstrained shortest path algorithm (DSP) on the same topologies and different loading conditions. DSP is the classical shortest path algorithm [2] that finds shortest paths to all destinations. DSP ignores the delay metrics on each link and finds the shortest path to all nodes using only the length metric. Let $\mathcal{S}$ denote the set of nodes for which there exists a feasible path satisfying the delay constraint, but DSP found an infeasible path. In an attempt to characterize the gain when considering enhanced path selection algorithms we define a gain factor g defined as:

$$g = \frac{\sum_{i \in \mathcal{S}} \frac{D_i - D_{min_i}}{D_{min_i}}}{N_{delay} - N_{dijkstra}} \tag{4.1}$$

where D_i, D_{min_i} are the path delays to destination node i using DSP and constrained shortest path algorithm, respectively. D_i, the delay of the path thus obtained by DSP is in general larger than D_{min_i} N_{delay} denotes the number of destinations that satisfy the delay constraint for the constrained shortest path algorithm, and $N_{dijkstra}$ is the total number of destinations that satisfy the same constraint when DSP is used. Notice that, if the delay constrained shortest path is a feasible problem, then N_{delay} will be n. $N_{dijkstra}$ denotes the number of destinations, whose shortest paths using DSP happen to satisfy the delay constraint.

The intuitive explanation of g, is as follows. Thus, $|\mathcal{S}|$ is $N_{delay} - N_{dijkstra}$. The total relative amount of violation is

$$\sum_{i \in \mathcal{S}} (D_i - D_{min_i})/D_{min_i} \tag{4.2}$$

The gain factor g is the average relative increase in delay and is obtained by dividing the above expression by $|\mathcal{S}|$. Thus larger the value of g, more severe is the violation by DSP, and thus more need for constrained shortest path algorithm. When the network is under heavy loading, the gain factor can be quite high.

Tables 4.1, 4.2, 4.3, 4.4, 4.5 compare the execution times of the various algorithms for different network sizes, connectivities and scale delay factors r. The column DAD shows the running time for the dynamic programming algorithm DAD. The column labeled LSA is the length scaling algorithm of R. Hassin (with slight modifications) [70]. The next column shows the times for DSA. As stated above, observe that DAD and DSA

simultaneously compute the paths for all destinations, unlike LSA. The last column in the table, labeled *Dijkstra* shows the running time for unconstrained Dijkstra. The numbers shown in parentheses are $N_{dijkstra}$, N_{delay}, and the gain factor g, respectively. From the tables, it is clear that for lightly loaded networks, DSP does as well as DSA, but when the loading is severe, the benefits of DSA are readily evident. For example, for the 35 node network, the gain factors are 0.0, 21.46, 11.27 5.624, and 3.625 for r= 0.1, 0.5, 1.0, 1.5, and 2.0 respectively. the value r = 1.0 corresponds to a moderately loaded network. In our experiments, we reflect the loading by scaling the edge delays by the factor r.

No. of Nodes	Connectivity	Time (μ seconds)			
		DAD	Length Scal. (single destination)	Delay Scal.	Dijkstra
10	0.2	4662	269	1558	23 (7/7/0.000)
15	0.2	18335	402	2425	53 (14/14/0.000)
20	0.2	11676	665	3389	88 (18/18/0.000)
25	0.2	19505	377	3651	136 (24/24/0.000)
30	0.2	23823	396	3373	174 (29/29/0.000)
35	0.2	31161	458	4000	230 (34/34/0.000)
10	0.4	2581	253	1662	26 (8/8/0.000)
15	0.4	6041	292	2378	50 (14/14/0.000)
20	0.4	10984	285	2507	91 (19/19/0.000)
25	0.4	16014	697	3024	125 (24/24/0.000)
30	0.4	23188	413	3656	175 (29/29/0.000)
35	0.4	59449	467	4357	240 (34/34/0.000)
10	0.6	2472	240	1738	27 (9/9/0.000)
15	0.6	6557	317	2431	52 (14/14/0.000)
20	0.6	8900	368	2685	85 (19/19/0.000)
25	0.6	13471	438	3235	138 (24/24/0.000)
30	0.6	23821	433	4565	211 (29/29/0.000)
35	0.6	28921	518	5132	229 (34/34/0.000)

Table 4.1: Algorithm comparison summary (Very lightly loaded network ($r = 0.1$))

Figure 4.1 shows the effectiveness of unconstrained Dijkstra shortest path (DSP) algorithm in meeting the delay constraints. Observe that in DSP we ignore the delay constraint, and solve for the shortest path from one source to all destinations. Then we check and see how many of these shortest paths satisfied the delay constraints. The Y-axis in the figure shows the fraction of destination nodes that violated the delay constraint. The X-axis is

No. of Nodes	Connectivity	Time (μ seconds)			
		DAD	Length Scal. (single destination)	Delay Scal.	Dijkstra
10	0.2	4809	810	1400	22 (7/7/0.000)
15	0.2	8154	296	1981	50 (14/14/0.000)
20	0.2	21727	305	2722	83 (18/18/0.000)
25	0.2	18410	353	4225	138 (23/24/6.860)
30	0.2	23336	396	3165	173 (29/29/0.000)
35	0.2	37338	454	3354	240 (33/34/8.073)
10	0.4	2647	234	1534	28 (8/8/0.000)
15	0.4	5693	277	2175	52 (14/14/0.000)
20	0.4	14657	318	3656	95 (18/19/5.521)
25	0.4	20489	360	2833	129 (24/24/0.000)
30	0.4	21116	396	3275	193 (20/29/12.788)
35	0.4	36298	472	8564	245 (32/34/8.553)
10	0.6	2475	265	1642	28 (9/9/0.000)
15	0.6	5524	284	2132	53 (13/14/8.118)
20	0.6	8922	348	2576	81 (19/19/0.000)
25	0.6	15365	374	6683	135 (22/24/8.641)
30	0.6	21629	425	8320	196 (25/29/9.290)
35	0.6	35590	487	22160	250 (25/34/21.466)

Table 4.2: Algorithm comparison summary (Lightly loaded network ($r = 0.5$))

the congestion factor r. The family of curves is for different size networks, all with the connectivity factor of 0.2. The plots clearly show that as congestion increases, the fraction of violated nodes monotonically increases. This is especially true if the network is sparse, as the numbers in the tables show. Thus when networks are in critical loading, the need for a constrained shortest path, seems of paramount importance, in connection setups.

4.4 Conclusions

QoS-sensitive routing is the first step towards achieving end-to-end QoS guarantees for connections with stringent QoS requirements. Perhaps the most important special case of the QoS routing problem is to find the cheapest (in terms of Administrative weights) route that satisfies a certain end to end delay guarantee. This chapter has proposed the use of two

No. of Nodes	Connectivity	Time (μ seconds)			
		DAD	Length Scal. (single destination)	Delay Scal.	Dijkstra
10	0.2	5304	269	1599	26 (7/7/0.000)
15	0.2	9875	314	2068	55 (14/14/0.000)
20	0.2	14509	484	6703	91 (15/18/1.138)
25	0.2	16734	46785	4216	134 (9/24/7.047)
30	0.2	28259	439	6828	177 (21/29/2.839)
35	0.2	40107	1233	8574	233 (18/34/6.250)
10	0.4	3616	312	1387	30 (8/8/0.000)
15	0.4	5706	350	2153	54 (13/14/0.869)
20	0.4	16593	308	3371	93 (11/19/3.067)
25	0.4	19144	370	2570	140 (14/24/2.349)
30	0.4	24622	440	5926	180 (13/29/9.893)
35	0.4	32930	2981	9315	233 (21/34/5.422)
10	0.6	2683	939	1649	28 (7/9/1.974)
15	0.6	9261	1445	2016	51 (6/14/6.374)
20	0.6	9987	306	3796	86 (7/19/3.628)
25	0.6	14156	355	5486	133 (11/24/9.172)
30	0.6	23869	2341	13282	189 (8/29/8.171)
35	0.6	83995	507	14170	246 (10/34/11.217)

Table 4.3: Algorithm comparison summary (Moderately loaded network ($r = 1.0$))

methods: length scaling [70] and delay scaling which provide an order of magnitude speed-up improvement in computation times compared to the exhaustive dynamic programming procedure. Our delay scaling algorithm is better suited to precalculation of routes; for real time calculation either the length scaling or the delay scaling approach may be better suited.

The length scaling and delay scaling are approximate constrained shortest path algorithms that are computationally efficient and lend themselves to implementation in real-time. Given that the exact problem is NP-hard, our solution offers a good practical alternative to support delay-sensitive routing. In the case of delay scaling, paths satisfying the imposed constraint are found from the source to all destinations and therefore can also be used to populate the route cache off-line when the number of nodes in the peer group is large and the update interval is short. Lagrangian relaxation is not as efficient as our scaling approach, and also does not provide any optimality (or near-optimality).

We have shown that in heavy load scenarios, the unconstrained Dijkstra fails to find

No. of Nodes	Connectivity	Time (μ seconds)			
		DAD	Length Scal. (single destination)	Delay Scal.	Dijkstra
10	0.2	4899	244	1496	23 (7/7/0.000)
15	0.2	8077	271	2302	54 (13/14/0.128)
20	0.2	10193	303	2631	85 (10/15/2.072)
25	0.2	23701	2048	5713	128 (5/24/6.779)
30	0.2	22266	2327	5893	187 (13/29/2.039)
35	0.2	34823	58077	7642	237 (12/34/5.041)
10	0.4	2459	260	1391	27 (8/8/0.000)
15	0.4	5103	345	2463	51 (8/12/0.690)
20	0.4	23814	468	3104	89 (7/19/3.504)
25	0.4	13110	3505	3469	126 (9/24/1.703)
30	0.4	37126	2222	4950	188 (9/29/4.412)
35	0.4	24476	487	10197	244 (14/34/4.860)
10	0.6	2270	289	1501	28 (8/9/1.165)
15	0.6	5170	1170	3719	56 (4/14/4.640)
20	0.6	9004	1486	3662	85 (7/19/2.607)
25	0.6	15394	370	7237	128 (9/24/8.136)
30	0.6	16466	1983	9262	184 (7/29/5.677)
35	0.6	24028	477	11663	265 (13/34/5.624)

Table 4.4: Algorithm comparison summary (Heavily loaded network ($r = 1.5$))

paths that satisfy the delay constraints resulting in crankbacks. Crankbacks are wasteful of network resources. By using computationally efficient approaches such as length scaling and delay scaling we considerably reduce the problem of crankbacks and allow the network the capability to satisfy call requests on first attempt.

The most important open problem is to devise similar computationally efficient and practical strategies for multiple QoS parameters.

No. of Nodes	Connectivity	Time (μ seconds)			
		DAD	Length Scal. (single destination)	Delay Scal.	Dijkstra
10	0.2	4530	244	1334	24 (6/6/0.000)
15	0.2	7079	362	1990	55 (9/10/0.165)
20	0.2	10279	285	2928	83 (6/13/1.815)
25	0.2	15761	2145	3613	133 (4/24/6.335)
30	0.2	26055	703	3467	167 (11/25/2.022)
35	0.2	27373	3189	4030	224 (12/34/3.182)
10	0.4	2457	239	1302	26 (7/7/0.000)
15	0.4	4940	291	2418	54 (8/9/1.167)
20	0.4	7407	1213	2015	89 (7/17/3.125)
25	0.4	11710	2698	3078	130 (6/21/1.293)
30	0.4	16259	393	3575	190 (10/28/3.408)
35	0.4	23214	457	7471	227 (14/33/2.940)
10	0.6	1905	242	1413	28 (6/8/0.778)
15	0.6	8973	494	1774	55 (9/12/1.511)
20	0.6	12085	1515	2579	90 (7/19/1.674)
25	0.6	11046	2347	2161	133 (7/24/5.273)
30	0.6	15825	3807	4005	197 (3/29/5.209)
35	0.6	26978	2174	6256	233 (8/34/3.625)

Table 4.5: Algorithm comparison summary (Very heavily loaded network ($r = 2.0$))

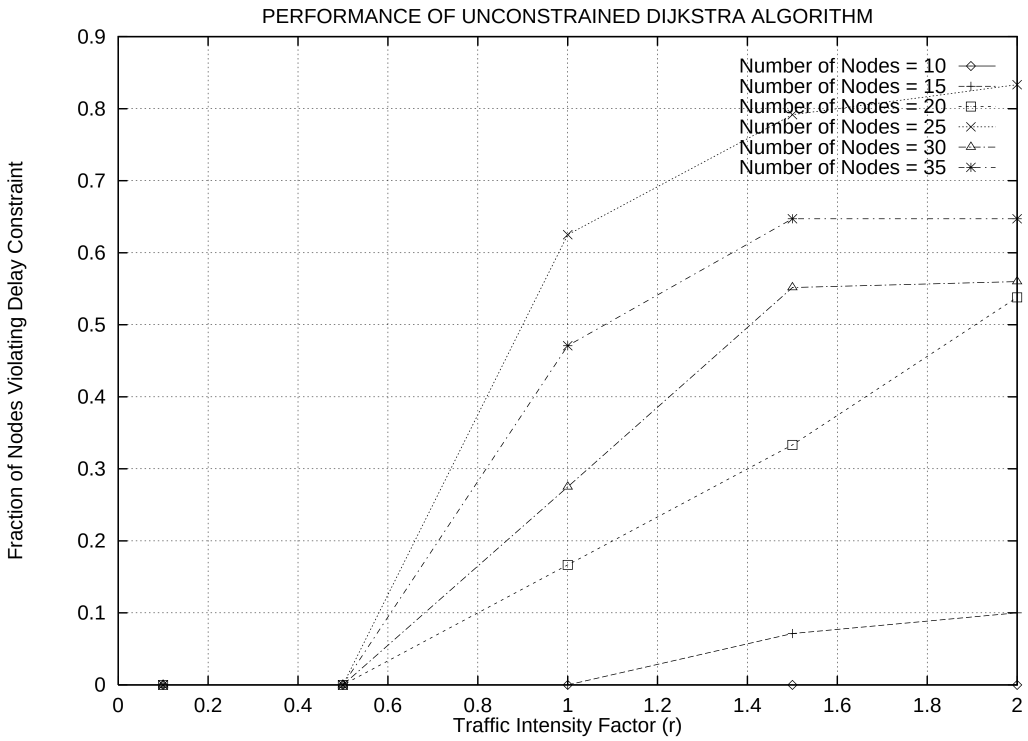

Figure 4.1: *Performance of Unconstrained Dijkstra Algorithm in Meeting the Delay Constraint*

Chapter 5

Delay Sensitive Multicast Routing[1].

5.1 Introduction

Dynamic Multicast Routing is a problem of growing importance with the advent of multimedia applications. It is better to use incremental, dynamic schemes for constructing multicast trees rather than monolithic ones, which produce solutions whose success is only ephemeral (Doar and Leslie [43]). The applications are quite varied; some require a large amount of bandwidth, some require strong end to end delay guarantees, and some require both. Application sensitive multicast routing is critical to the success of these applications. At the same time, it is important to minimize the cost of the multicast tree. In this chapter we assume that each link has a cost and a delay, and data flows from a source to a set of receivers on the multicast tree. Our goal is to produce trees which have a small cost and also a small end to end delay from the source to each receiver. More formally. let $c(e)$ be the cost of link e and $d(e)$ its delay. Let T be the multicast tree, s the source, and $P_T(v)$ the path from s to v along the tree T. Then the total cost of the tree is $\Sigma_{e \in T} c(e)$ and the end to end delay of receiver r is $\Sigma_{e \in P_T(r)} d(e)$.

For multicast applications where end to end delay is of overriding importance but the bandwidth requirement is small (such as stock tickers), shortest path routing strategies perform well. A shortest path routing strategy connects the source of the multicast to each

[1]This chapter represents joint work with Kamesh Munagala [62]

receiver using the shortest unicast route from the source to the receiver (or the other way round) in the underlying IP/ATM network. The three commonly used multicast routing protocols, DVMRP [129], CBT [15], and PIM [41], all use shortest path routing strategies. In shortest path strategies, any sharing of routes by different receivers is incidental; the routing scheme itself makes no effort to share routes[2]. In this chapter we focus on applications which are both bandwidth intensive and delay sensitive, and we argue that shortest path strategies will not scale well for such applications. It is clear that for such applications we need algorithms which explicitly try to maximize route sharing between different multicast receivers.

For applications which require large amounts of bandwidth, but are not delay sensitive, the dynamic greedy algorithm[3] proposed by Imase and Waxman [74] works very well, *i.e.*, produces trees with good total cost. This algorithm adds a new multicast receiver using the cheapest path to any of the nodes on the existing multicast tree. Imase and Waxman proved that no other dynamic algorithm can perform much better(in terms of the cost of the tree) than the greedy algorithm if multicast receivers are allowed to join the multicast group at arbitrary times, but are not allowed to leave (see Theorem 5.1). In the worst case, the tree produced by shortest path strategies can be upto a factor k worse than the tree produced by the greedy algorithm (in terms of the total cost of the tree), where k is the number of multicast receivers. One might argue that such pathological worst case scenarios do not occur in practice. We show that even when all multicast receivers are chosen randomly from a very large grid, the shortest path algorithm produces trees which are expected to be a factor $\sqrt{k}$ worse (in terms of the total cost of the tree) than those produced by the greedy algorithm (Theorem 5.2). This demonstrates that shortest path routing strategies will not scale well for bandwidth intensive applications. We support our theoretical results by extensive simulations, along the lines of Doar and Leslie [43], and Waxman [132].

A node is said to be rerouted when its parent pointer in the multicast tree changes (i.e. when the node directly upstream of it changes). Neither the greedy strategy nor the shortest path routing strategy reroutes nodes already on the multicast tree. However,

[2]Intuitively, this is the reason why shortest path routing can result in costly multicast trees.

[3]This algorithm has also been referred to as the *naive* algorithm. The term *naive* is a misnomer; this algorithm is indeed quite simple, but it is optimal in the sense that no other algorithm can perform better. An elegant analysis of this algorithm was first presented by Imase and Waxman [74]; a different but equally elegant analysis was done by Alon and Azar [3]. These authors call it the greedy algorithm and we use their terminology.

rerouting, used in a controlled fashion, is a potentially helpful tool for constructing "good" multicast trees.

We have implemented an application-switched multicasting system which uses the greedy algorithm as the routing protocol. Our experience suggests that the greedy algorithm lends itself to efficient and simple implementations. We are currently in the process of deploying our system for broadcasting an entire crawled copy of the world wide web. We describe this system briefly in Section 5.2.3.

Why then is the greedy algorithm not used more widely in practical settings? There are three main reasons. The first has to do with the fact that this algorithm does not handle deletions gracefully [74] *i.e.*, the strong theoretical guarantees break down when nodes can both join and leave the multicast group. We observe (Section 5.2.4) that a small number of reroutes, done after each deletion, can restore these guarantees. The second reason is that this algorithm does not offer any end to end delay guarantees. Thus it is not suitable for applications such as video transmissions, which have a high bandwidth requirement as well as stringent end to end delay requirements. In contrast, the shortest path based strategies offer end to end delay guarantees which are comparable to those offered for unicast routing, since the path from the source to each receiver is the same as the unicast (or the reverse unicast) route. We address this issue in Section 5.3 by giving an algorithm that does a small number of reroutes and offers both a good overall cost for the tree (comparable to the greedy heuristic), and good end to end delay guarantees (comparable to the shortest path algorithm). Our simulation results (Section 5.4) show that our algorithm achieves end to end delays comparable to the shortest path strategy, and overall tree cost comparable to the greedy algorithm.

The third reason is more historical. The underlying IP network makes it easier to implement shortest path strategies. Further, the MBone [102, 104], the most popular multicasting system in use, does not support source routing or weak source routing and inherently requires shortest path routing. So implementing any new routing protocol will require significant amount of effort. Given the fact that the shortest path heuristic does not scale well with bandwidth for a model as simple as random points on a grid, we believe that this effort would be well spent.

In this chapter we focus on applications with a single sender. Since trying to optimize for arbitrary cost and delay metrics is hard even in the static case(where all multicast receivers are known in advance), we further assume that the cost and delay metrics are proportional to each other (see Section 5.3.1).

Our main result

Our main result is an algorithm which gives good end to end delay guarantees as well as good guarantees on the overall cost of the tree constructed. We call our algorithm the DSG(*Delay Sensitive Greedy*) Algorithm. We need to do a small amount of rerouting – each multicast receiver gets rerouted at most once. We assume that the delay metric and the cost metric are proportional to each other; we discuss this assumption in greater detail in Section 5.3.1. The total cost of the tree constructed by our algorithm is at most a constant factor away from the cost of the tree constructed by the greedy algorithm; since the greedy algorithm is the best possible dynamic algorithm (upto constant factors) for minimizing the total cost of the tree, the DSG algorithm is also the best possible upto constant factors. For each multicast receiver, the end to end delay between the source and that receiver is at most a small constant factor away from the shortest possible end to end delay between the two. Thus, the DSG algorithm is provably within constant factors of the best possible cost guarantee as well as the best possible end to end delay guarantees. We would like to reiterate that the end to end delay guarantees we give are per receiver. For each receiver, the end to end delay experienced is within a constant factor of the best possible end to end delay for that receiver. This is stronger than having each path from the source to some receiver satisfy a global end to end delay threshold. Our algorithm provides a continuous tradeoff; if we want more strict end to end delay guarantees, the guarantee for the total cost of the tree degrades.

If there are no deletions, the running time of our algorithm is $O(1)$ per multicast receiver, over and above the running time of the greedy algorithm.

We simulate our DSG algorithm for Waxman networks [132] and grid networks. Our simulation results (Section 5.4) show that our algorithm performs almost as well as the greedy algorithm in terms of the overall cost of the tree, and obtains end to end delays which are quite comparable to those obtained by the shortest path algorithm. The actual results observed are much better than the worst case guarantees we offer. Most receivers never get rerouted.

The DSG algorithm is modeled after an elegant algorithm due to Khuller et al. [85], which in turn extends the work of Awerbuch et al. [12]. We adapt the algorithm of Khuller et al. to an online setting. We believe that our algorithm is simple enough to lend itself, like the greedy algorithm, to efficient applications. We briefly explain how to handle deletions using our algorithm.

We also note that the DSG algorithm can optimize for any additive metric, and not just the delay metric. For instance, we could consider the packet drop probabilities on a link, which is additive under the standard *Link Independence* assumption. We could then construct trees with small cost in which each individual multicast receiver sees a small packet loss probability.

Previous work

For unicast routing the problem of finding delay constrained shortest paths is well understood. Efficient solutions to this problem are known even when the two metrics are completely unrelated (see Chapter 4 for details). Multicast routing is a hard problem even in an abstract theoretical setting, and we should not expect the kind of simple and efficient solutions that we obtain for unicast routing.

In the offline case, the KMB heuristic [94] is known to produce trees with near optimal total cost. Alternatives to such heuristics were first studied by [25]. Doar and Leslie, as well as Wei and Estrin presented a detailed discussion of the shortest path heuristic versus greedy/KMB heuristics [43, 133]. They concluded, not surprisingly, that the shortest path heuristic results in good delay, whereas the greedy algorithm/KMB result in worse delays but better overall cost of the multicast tree. Over the last few years there have been several attempts to address the problem of finding cheap multicast trees with some delay guarantees [91, 109, 72]. These heuristics assumes that the cost and the delay are two distinct metrics. But these algorithms do not have good worst case (or even probabilistic) performance/running time guarantees. The algorithm of Kompella et al. [91] is pseudo-polynomial. There is no guarantee that it will terminate in polynomial time; further, the cost of the tree produced may be as bad as k times the optimal (recall that k is the number of multicast receivers). The algorithm mentioned in [72] offers strong guarantees on performance, *i.e.*, it finds an optimal cost tree with the path from the source to each receiver satisfying a given threshold. But this algorithm is very complicated to implement, and does not offer even a probabilistic guarantee of running in polynomial time. On the other hand, the algorithm in [109] offers performance guarantees, and a good probabilistic running time, but it considers only the case where all the multicast nodes are known in advance. It seems difficult to adapt the algorithm to the dynamic setting that we are considering in this chapter.

Notation and Formal Description of the Problem

We are given a weighted undirected graph $G = (V, E)$ with a cost function $c : E \to \Re$, and a source node s. Receivers $R = \{v_1, v_2, \ldots, v_k\}$ arrive in an online fashion. Our goal is to maintain a Steiner tree T on $R \cup \{s\}$. Let $c(T) = \sum_{e \in T} c(e)$ be the total cost of T. Let $d_G(s, v)$ be the shortest path distance from s to v with respect to the cost function c, and let $d_T(s, v)$ be the path length from s to v in T. Also, for a receiver v, let $\text{Stretch}_T(v) = d_T(s, v)/d_G(s, v)$. Now, define $\text{Stretch}_T = \max_{v \in R} \text{Stretch}_T(v)$ and let $\text{CostRatio}_T = c(T)/c(T^*)$, where T^* is the minimum weight Steiner tree on the vertices in $R \cup \{s\}$. An algorithm for this problem is (p, q)-competitive if $\text{Stretch}_T \leq p$ and $\text{CostRatio}_T \leq q$ for the tree T produced by the algorithm. In terms of this notation, our main result is an $O(1, \log |R|)$-competitive algorithm.

5.2 Background: The Greedy Algorithm

The greedy algorithm does the following. Suppose there is a multicast already in progress, over tree T and with a single sender s. Suppose further that a new node v wants to join the multicast group as a receiver. The greedy algorithm finds the node in T that is the closest in distance (where distance is measured using an appropriate link metric) to v, and attaches v to T via this node. Examples of appropriate link metrics are cost, delay, and hop-count.

Imase and Waxman [74] proved the following theorem.

Theorem 5.1 [74] *Assume that the total number of receivers in the multicast tree is k. The greedy online algorithm produces a tree with cost at most $\lceil \log k \rceil$ times the cost of the optimal tree. Further, no algorithm can guarantee a factor better than $\frac{\lceil \log k \rceil}{2}$.*

In this section we do a theoretical analysis of the algorithm for a model of random networks, describe its performance in simulated networks, and describe our preliminary implementation experience. We describe an example application where this algorithm is clearly the right one to use. We then point out that this algorithm is not appropriate for delay sensitive applications.

5.2.1 Analysis

Assume that the multicast group has a single sender and $k-1$ receivers. Let T_G denote the tree constructed by the greedy algorithm, T^* denote the shortest (optimal) tree, and T_S denote the shortest path tree (ie. the tree constructed using an approach like DVMRP). For any tree T, let $C(T)$ denote its total cost. Imase and Waxman proved that $C(T_G) \leq (\log k)C(T^*)$. In contrast, it is possible for $C(T_S)$ to be as large as $(k-1)C(T^*)$ in the worst case. This should make a clear case for the superiority of the greedy algorithm for minimizing use of network resources. However, one might object that the pathological networks which result in the large difference in the performance of the greedy algorithm and DVMRP like approach do not occur in practice. To address this concern, we analyze the greedy and the shortest path algorithms in the following simple model. We assume that we have a completely connected network, where the nodes in the network are drawn randomly from within a unit square and the cost of an edge is the Euclidean (l_2) distance between its endpoints. This is equivalent to drawing points randomly from a very large grid.

The Θ-notation denotes equality (up to constant factors) as k grows large.

Theorem 5.2 *The expected cost of the tree T^* is $\Theta(\sqrt{k})$. The expected cost of the tree T_G is also $\Theta(\sqrt{k})$. The expected cost of the tree T_S is $\Theta(k)$.*

The distance between two randomly chosen points on a grid is $\Theta(1)$; using linearity of expectations we immediately obtain $\mathbf{E}[c(T_S)] = \Theta(k)$. $\mathbf{E}[c(T^*)] = \Theta(\sqrt{k})$ was shown by [130]; $\mathbf{E}[c(T_G)] = \Theta(\sqrt{k})$ follows from [130] and [128].

Thus even when the multicast nodes are chosen probabilistically from a simple network (ie. the grid), the performance difference between the greedy algorithm and a DVMRP-like approach is of the order of $\sqrt{k}$. Further, the greedy algorithm is asymptotically optimal in this model (ie. optimal up to constant factors). This analysis makes a strong case why a DVMRP like approach will not scale; the input in the above analysis is not adversarial and the metric space is realistic (Euclidean grid). We would like to point out that CBT and PIM also use a shortest path tree in some form or another; therefore the routing inefficiencies in these schemes too will increase with the size of the multicast group.

We compare (Figure 5.1) the cost of the tree produced by the greedy algorithm to the cost of the shortest path tree as a function of k in one specific random experiment, where the nodes are random points in a unit square.

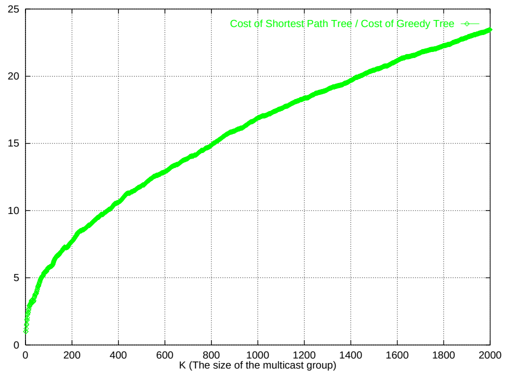

Figure 5.1: The ratio of the cost of the shortest path tree to that of the greedy tree when nodes are drawn at random from within a unit square.

5.2.2 Simulations

Doar and Leslie [43] simulated the performance of the greedy algorithm and demonstrated that it is not much worse than the well known KMB heuristic [94] which produces near optimal solutions on real life networks (and a 2-approximation in the worst case). Doar and Leslie use graphs which are similar to grids; in view of Theorem 5.2 their results are not surprising. We do similar simulations as in [43] but for larger networks and larger multicast groups. We demonstrate the difference between the shortest path heuristic and the greedy algorithm in figures 5.6 and 5.10.

5.2.3 A Case Study

We have developed a small application-switched multicasting system which uses the greedy algorithm for routing. We are currently in the process of deploying this algorithm for specific applications (one of which is described later in this section). We use hop count as the metric to be optimized. This is equivalent to finding the tree which minimizes the total amount of bandwidth used up in the network. The routing protocol is quite simple. A new receiver sends a "JOIN" request to the source of the multicast. The source propagates this request down the tree using a separate control channel. Each multicast enabled node in the tree then sends a single ICMP[4] packet to the new receiver after setting the TTL[5] field in the ICMP header to a fixed value. The receiver reads the TTL off the ICMP packet, and

[4]Internet Control Message Protocol [103]
[5]Time To Live [103]

subtracting the current TTL value from the initial TTL value gives the number of hops from the node in the tree to the receiver. The receiver then sends a "CONNECT" message to the node in the tree which is the least number of hops away.

Our implementation experience suggests that implementing the greedy heuristic is not hard. The amount of communication needed for each addition to the tree is small – just one ICMP packet per multicast participant. The delay for adding a new node to the multicast tree is high. For some applications this delay is not crucial; for applications where this delay is crucial, we could use a two step JOIN process where the new receiver starts receiving packets immediately from the sender and the routing protocol runs in the background. This is the approach followed by current CBT implementations.

There are applications where it is critical to reduce the amount of network usage. We are currently in the process of deploying our multicasting system for one such application – repeatedly broadcasting an entire crawled copy of the world wide web. The amount of data to be transferred is large enough that it is important to ensure that the routing protocol does not waste network bandwidth.

The two problems with the greedy algorithm are that it is not good for delay sensitive applications, and that it does not handle deletion from the multicast group as gracefully as shortest path based protocols. We now show how the greedy algorithm can be extended to handle deletions. We defer the discussion of delay sensitive applications to Section 5.3.

5.2.4 Handling deletions using the greedy algorithm

We show how using the greedy algorithm for additions to the tree, coupled with a minimal amount of rearrangement after deletions, can result in good guarantees for the total cost of the multicast tree. We assume, for simplicity, that each branching point in the multicast tree is a participating node. Each time a node gets deleted, we look at all the nodes just downstream from the deleted node. These nodes are then re-attached, one by one and in the order in which they joined the group, to the closest destination on the remaining multicast tree. The entire subtree at these downstream nodes just migrates along with the downstream nodes themselves. For example, in Figure 5.2, if node A gets deleted, we can just re-attach nodes B and C to the closest nodes in the rest of the tree, and the subtrees at B and C will remain unchanged. Thus the parent pointers at D and E need not change.

Theorem 5.1 continues to hold for the above algorithm. The crucial observation is that after deleting v and then reattaching the nodes immediately downstream from v, the tree

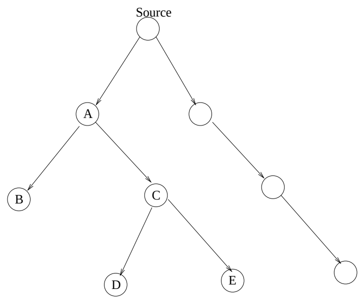

Figure 5.2: Deleting node A affects only nodes B and C – the parent pointers at nodes D and E need not change.

obtained is the same as the tree that would have been obtained if v had never joined the multicast tree in the first place.

It has been suggested by some researchers (for example Doar and Leslie [43]) that when a node gets deleted, we could just leave it in the multicast tree anyway. This can result in the cost of the multicast tree becoming much worse than the best possible tree for the group. Imase and Waxman [74] show that it is not possible for any dynamic algorithm to handle deletions without rearranging the multicast tree.

5.3 Adapting the greedy algorithm to delay sensitive applications

5.3.1 The delay metric

Ideally, we would like to have two different link metrics, a delay metric and a cost metric. However the problem of finding the cheapest multicast tree where the paths from the source to each receiver satisfy a certain delay bound is very hard, even in an offline setting where all the multicast participants are known in advance. More specifically, any polynomial time algorithm for this problem must have an $\Omega(\log n)$ approximation ratio unless NP $\subseteq$ DTIME$(n^{\log\log n})$ [16]. Partial progress has been made towards solving this problem; a polylogarithmic approximation that runs in quasipolynomial time is presented in Chapter 7 [93, 29]. A heuristic for the case of two different metrics was proposed by [91]; they give empirical evidence but provide no guarantees on the optimality of the solution produced. Further, their heuristic works only if all the nodes are given in advance. To get around this problem, we propose using a single metric for both delay and weight. We do not make specific recommendations on the choice of this metric; that is a separate problem beyond the scope of this chapter. We do believe that just using the delay of a link as both the delay and the cost metric would be a good candidate to consider.

5.3.2 The DSG (*Delay Sensitive Greedy*) Algorithm

We are given an undirected (symmetric) network G and we assume that each edge e in the network has a delay $d(e)$ defined. As explained in Section 5.3.1, the cost of an edge is also proportional to $d(e)$. Since the network is symmetric, the delay of an edge is the same in both directions. Finding good multicast trees in an asymmetric network is a hard problem [29], and is beyond the scope of this chapter.

Suppose a multicast session S is already in progress. Let T be the tree which is used to serve S. We treat s, the source of the multicast, as the root of the tree T. For any node v in the tree, let $d_T(s,v)$ denote the delay of the path from s to v along the tree T, and $d_G(s,v)$ denote the delay along the fastest possible path from s to v in G. Let us call nodes in T which actually wish to receive the data as *receivers*. All other nodes in T merely act as intermediate nodes on paths between receivers. Let β be the maximum permissible value of the ratio $d_T(s,v)/d_G(s,v)$ over all receivers v. Let α be any number between 1 and β (ie. $1 < \alpha < \beta$). The algorithm accepts α and β as parameters.

For any node v in tree T, let $\text{parent}(v)$ denote the node in T to which v is directly connected. In other words, v receives its multicast data from $\text{parent}(v)$.

We describe below our DSG algorithm to add a new vertex v to an existing multicast tree T. We first provide some informal intuition, and then give a formal description.

The basic idea behind the DSG algorithm is quite simple. When a request to join the multicast tree T arrives, the requesting node v is first connected to the node in T which is the closest to v (This is just the greedy algorithm). As soon as it happens that $d_T(s,v) > \beta d_G(s,v)$ for the newly added receiver v, we reroute v so that it obeys the constraint $d_T(s,v) \leq \alpha d_G(s,v)$. Since $\alpha < \beta$, the above constraint is more strict than the one we are required to satisfy. Along with v some other nodes on the path, along T, from v to s may also get rerouted; these rerouted nodes also satisfy the more stringent constraint. No node gets rerouted more than once; most nodes never get rerouted.

We now present the DSG Algorithm in detail. The algorithm works in four steps:

(1) JOIN: We first check if the new receiver v is already on the existing tree T[6]. If v is on the tree then proceed directly to Step (2). If not, v is joined to the node u in T that is closest to it, *i.e.*, to the multicast node $u \in T$ with minimum $d_G(u, v)$. We add the shortest path from v to u to T, and for all nodes t on this path, set parent(t) to be the neighbor of t on the path in the direction of u.

(2) CHECK: Check whether $d_T(s, v) \leq \beta d_G(s, v)$. If so, exit. If not, goto Step 3.

(3) CUT: Starting from v, traverse parent pointers in T and locate the first node v' such that $d_T(s, v') \leq \alpha d_G(s, v')$. "Cut" the path just downstream of v'. Let P be the path from v to v', excluding v'.

(4) RELAX: This is the main step where the DSG algorithm deviates from the greedy strategy. Let w be the child of v' on path P, *i.e.*, w is the end of the path P opposite node v. Traverse path P starting at w and ending at v. For every node t on P encountered during the traversal, perform the following steps:

1. If $d_T(s, t) > \alpha d_G(s, t)$, add the shortest path from s to t to the tree T, and delete the edge connecting t to t' from T. For all nodes x on the shortest path from s to t, set $d_T(s, x) = d_G(s, x)$, and parent(x) to be the neighbor of x on the shortest path, towards s. This step is called the *Rerouting Step*.
2. Let t' be the node succeeding t on P. If $d_T(s, t') > d_T(s, t) + d_G(t, t')$, set $d_T(s, t') = d_T(s, t) + d_G(t, t')$. All this means is that since the distance $d_T(s, t)$ may have changed due to some rerouting either at t or upstream of t, we have to update $d_T(s, t')$ to reflect this fact. It may have happened that t' may already have got rerouted due to its being on some shortest path that was added upstream, in which case we need not update $d_T(s, t')$.

5.3.3 Example

The DSG algorithm is best illustrated by means of this simple example(see Fig. 5.3). Let G be the 4 node graph shown in Fig. 5.3(a). We choose $\alpha = 1.6$ and $\beta = 2$, and let the node A be the source of the multicast.

We consider the sequence of addition of node B, followed by C, and finally D.

[6]As mentioned before, a node could be on the tree without being a receiver

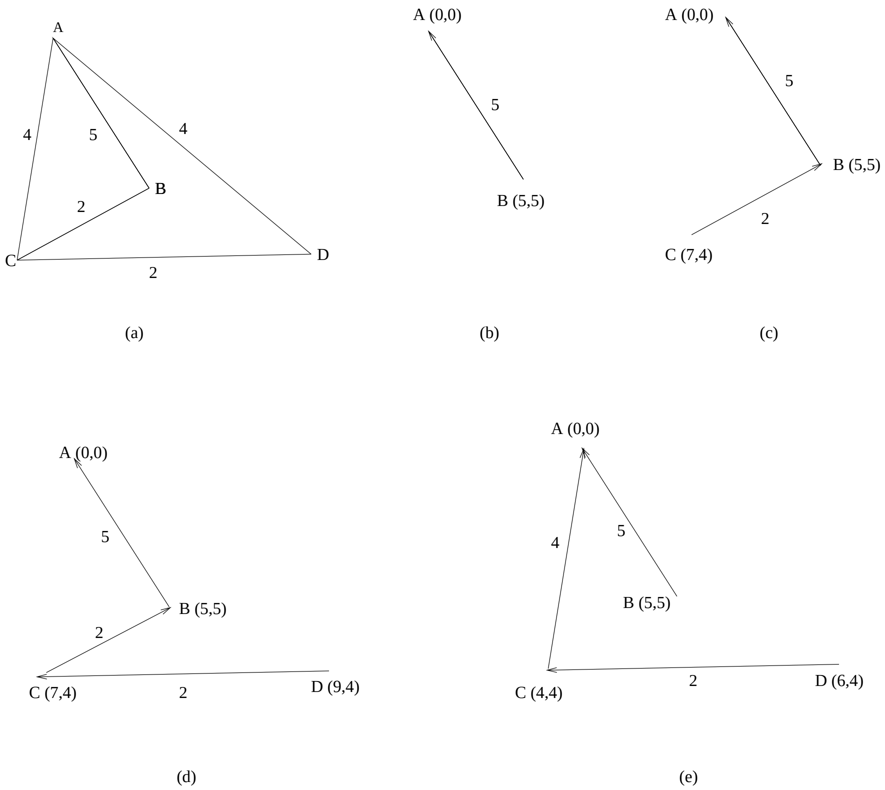

Figure 5.3: Execution of the Algorithm on a 4 Node Graph.

When node B is added(Fig. 5.3(b)), it is connected to A, and so, parent(B) = A, and $d_T(A, B) = 5$.

When node C is added(Fig. 5.3(c)), it is connected in the greedy fashion to its closest neighbor in T, in this case, B. Therefore parent(C) = B and $d_T(A, C) = 7$. Since $d_T(A, C)$ is less than β times $d_G(A, C)$, no rerouting takes place.

When node D is added(Fig. 5.3(d)), it is first connected greedily to its closest node, C, setting parent(D) = C and $d_T(A, D) = 9$. However, $d_T(A, D)$ violates the β condition, so we need to reroute it.

We start from D and follow parent pointers finding the first node satisfying the α condition. Note that C violates the α requirement, while B does not. With reference to the notation used in the description of the algorithm, in steps (3) and (4), $v' = B$, $w = C$, $v = D$, and our path P consists of the nodes C and D. We reroute $v' = C$ in Fig. 5.3(e) using the shortest path from C to A. This makes parent(C) = A, and $d_T(A, C) = 4$. This causes $d_T(A, D)$ to be set to 6 in sub-step (2) of step (4) of the algorithm. Since D now satisfies the α condition, it is not rerouted. This completes the multicast routing of all the nodes in the graph, and the final tree is shown in Fig. 5.3(e).

5.3.4 Theoretical Analysis

We now prove the following results, which will show that the tree produced by the DSG algorithm is within a constant factor of the optimum weight, while all path lengths get dilated by at most a constant factor. We define a reroute as the change in the parent pointer of an existing node in the tree.

Theorem 5.3 *No node in the tree T generated by DSG gets rerouted more than once.*

Proof. Note that when a node v gets rerouted, the following two conditions always hold:

1. Just before rerouting, $d_T(s, v) > d_G(s, v)$. This is because the reroute is always along the shortest path to the source.

2. After rerouting, $d_T(s, v) = d_G(s, v)$, and this value remains the same as long as v is in T.

This implies that once a node gets rerouted, condition 1 is no longer satisfied for it, and so, it cannot get rerouted again. □

In practice, most nodes never get rerouted (see Fig. 5.7, 5.11).

Theorem 5.4 *The cost of the tree produced by DSG is at most* $\frac{\alpha(1-\alpha+2\beta)}{(\alpha-1)(\beta-\alpha)}$ *times the cost of the tree generated by the greedy algorithm.*

Proof. Our analysis is similar to the technique used by Khuller et al. [85] for showing the existence of shallow spanning trees in any graph.

First observe that since we are assuming costs to be proportional to delays, and we are only concerned with ratios of costs, for the purpose of analysis, we can assume that cost of any edge is *equal* to the delay of the edge. This does not change the result, and simplifies the analysis.

If the RELAX step does not reroute, the DSG algorithm adds exactly the path that would be added by the greedy algorithm. If this step actually results in a relaxation, we will show that the new paths added in that step are at most a constant factor multiple of the old path. This will prove the result.

Consider the path P that was relaxed in one particular RELAX step of the algorithm. Let us focus our attention on this step. For the purpose of analysis, let us include v' in the path. Let $v_1, v_2, \ldots, v_k$ be the sequence of nodes along P for which we were forced to add shortest paths to s. Note that $v_1 = w$. For any two nodes x and y on P, let $L_P(x,y)$ be the distance between nodes x and y along path P. Since the greedy algorithm adds only shortest path segments and since v' and v are neighbours in the tree, $d_G(v', v_1) = L_P(v', v_1)$. The following inequalities are easily deducible from the algorithm:

1. $d_T(s, v') + d_G(v', v_1) > \alpha d_G(s, v_1)$.
2. $d_G(s, v_{i-1}) + L_P(v_{i-1}, v_i) > \alpha d_G(s, v_i)$ for $i = 2, 3, \ldots, k$.

The value of $d_T(s, v')$ in the above inequalities is that at the beginning of the RELAX step. Adding the inequalities enumerated above, we get

$$d_T(s, v') + L_P(v', v_1) + \Sigma_{i=2}^{k} L_P(v_{i-1}, v_i) > (\alpha - 1)\Sigma_{i=1}^{k} d_G(s, v_i) \tag{5.1}$$

Let L_{new} denote the total length of the new paths added during the relaxation phase. Note that all the new paths are shortest paths, and their total length is clearly at most $\Sigma_{i=1}^{k} d_G(s, v_i)$. Also, $L_P(v', v) \geq L_P(v', v_1) + \Sigma_{i=2}^{k} L_P(v_{i-1}, v_i)$. Inequality 5.1 can now be simplified to:

$$L_P(v', v) + d_T(s, v') > (\alpha - 1) L_{new} \tag{5.2}$$

Note that since the greedy algorithm adds unrelaxed paths, the path it has added for inserting the vertices w through v on the path P is precisely the length of the path P including the edge (w, v'). Therefore, this phase of the greedy algorithm adds weight $L_P(v', v)$. The DSG algorithm will add paths of length at most $L_{new} + L_P(v', v)$ in the same phase. Since no vertex on P will ever get relaxed again, if we bound L_{new} in terms of $L_P(v', v)$, the same bound holds of the weight of the tree the DSG algorithm constructs in terms of the weight of the greedy online tree.

Observe that the following inequalities hold for the vertices v and v':

1. $d_T(s, v') < \alpha d_G(s, v')$,

2. $\beta d_G(s, v) < d_T(s, v') + L_P(v', v)$, and

3. $d_G(s, v') < d_G(s, v) + L_P(v, v')$, by triangle inequality.

From these, we deduce:

$$d_T(s, v') < \frac{\alpha(1+\beta)}{\beta - \alpha} \cdot L_P(v', v). \tag{5.3}$$

From equations 5.2 and 5.3, it follows that

$$L_{new} < \frac{\beta(1+\alpha)}{(\alpha - 1)(\beta - \alpha)} \cdot L_P(v', v). \tag{5.4}$$

Therefore,

$$L_{new} + L_P(v', v) < \frac{\alpha(1 - \alpha + 2\beta)}{(\alpha - 1)(\beta - \alpha)} \cdot L_P(v', v). \tag{5.5}$$

This completes the proof of the theorem. □

Since the weight of the tree produced by the greedy algorithm is optimal, we know that our tree is at most a small constant factor away from the optimal *in the worst case.* In typical scenarios, this factor is much smaller than that predicted in Theorem 5.4.

Theorem 5.5 *Let T be the tree generated by DSG. For all receivers v on T, $d_T(s, v) < \beta d_G(s, v)$.*

Proof. The proof follows trivially from the observation that we reroute v if it does not satisfy the condition stated in the theorem. □

Theorems 5.4 and 5.5 allow us to trade off the cost of the multicast tree with the end to end delay guarantees in a controlled fashion, by changing parameters α and β. For example, if we choose $\beta = 3$, the constant in Theorem 5.4 is minimized when $\alpha = 1.83$; the minimum value is around 10. If we choose $\beta = 2$ the minimum value of the constant in theorem 5.5 is around 21. From the simulations we can see that for realistic networks, the guarantees for the cost are much better than these worst case predicted values. The tradeoff between end to end delay and overall cost is clearly discernible from our simulation results.

Theorem 5.6 *Let the greedy algorithm take total time T to add all nodes of the graph to the multicast tree. Then, the DSG algorithm takes total time $O(n+T)$ to perform the same task. Here, n is the number of nodes in the graph.*

This means that on the average, the DSG algorithm takes an additional $O(1)$ time to add a node to the multicast tree compared to the greedy algorithm. The proof of theorem 5.6 is simple; since each node gets rerouted at most once, each pointer in the tree changes at most once.

5.3.5 Deletions

To preserve the guarantees offered in Theorems 5.4 and 5.5, we need to reroute each node in the descendent tree of a deleted node. Thus, when node A gets deleted in figure 5.2, we will have to reroute each of nodes B, C, D, and E; each of these nodes will have to leave the tree and rejoin. These rejoins can happen gradually and in arbitrary order.

5.4 Experimental Analysis

We have simulated the DSG algorithm on two classes of graphs, the Waxman networks and the rectangular grid graphs. For both models, we consider just the addition of nodes in our experiments.

5.4.1 Waxman Networks

In this model [132] we choose n nodes uniformly at random from a square Cartesian grid on the plane. For every pair of nodes (u, v), we add an edge in G between them using the probability function:

$$P_e(u,v) = \frac{k\bar{e}}{|G|}\gamma \cdot \exp\left(\frac{-d(u,v)}{\delta L}\right)$$

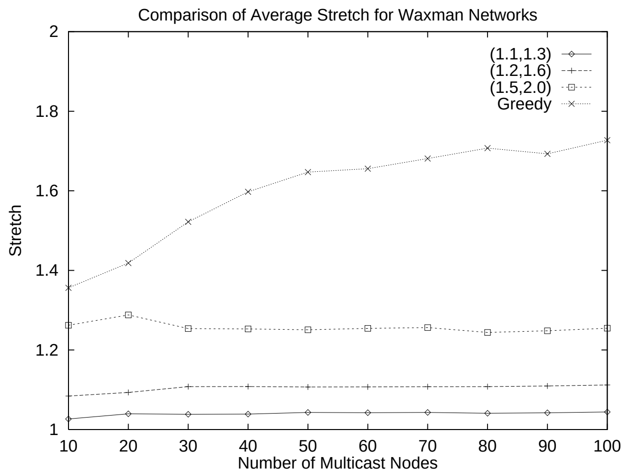

Figure 5.4: Comparison of Average Stretches for Waxman Networks with 1000 nodes.

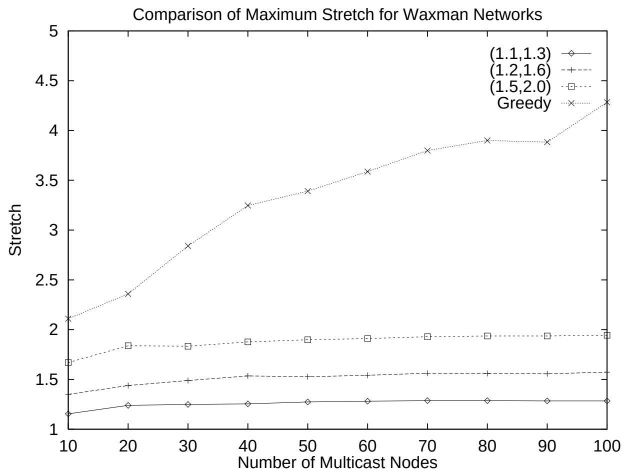

Figure 5.5: Comparison of Maximum Stretches for Waxman Networks with 1000 nodes.

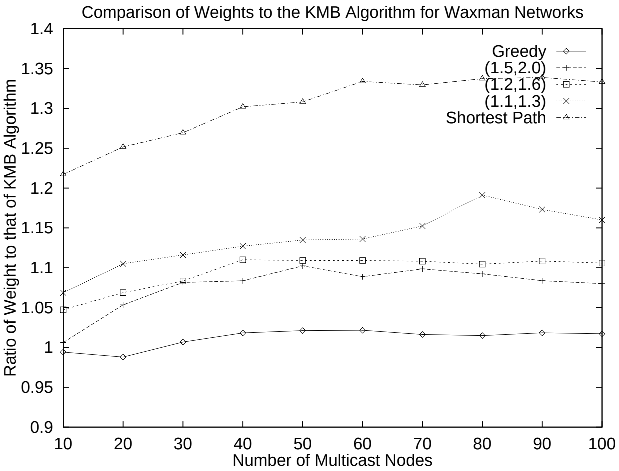

Figure 5.6: Comparison of Weights for Waxman Networks with 1000 nodes.

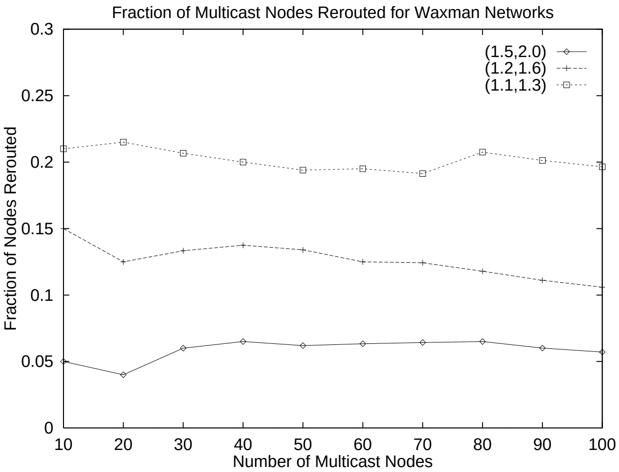

Figure 5.7: Fraction of the Multicast Nodes which are rerouted by the DSG algorithm for Waxman Networks with 1000 nodes.

where $d(u, v)$ is the Euclidean distance between u and v, L is the diagonal of the grid, γ and δ are parameters in the range $(0, 1]$, $\bar{e}$ is the mean degree of each node, $|G| = n$ is the number of nodes in the graph, and k is a constant chosen by experiment. The cost of an edge is simply its Euclidean length.

In our case, the values of the parameters are: $\gamma = 0.2$, $\delta = 0.25$, $\bar{e} = 4$, $n = 1000$ and $k = 25$. In other words, we will be considering 1000 node graphs with mean degree 4. The factors δ and γ are chosen to make the graph resemble geographical maps of major nodes on the Internet [43].

The experimental results are summarized in figures 5.4,5.5, 5.6 and 5.7. The curves corresponding to the DSG algorithm are labeled by the pair (α, β) that we use as parameters in the execution(see section 5.3.2). We run the DSG algorithm on ten random graphs each for three distinct pairs of (α, β) values(shown in the figures), and take the average for each pair. The source is chosen from the pool of nodes at random. We add randomly chosen nodes to the multicast tree in increments of 10, till we have added 100 nodes, which is 10% the size of the network. Readings are taken after every addition of 10 nodes.

In figures 5.4 and 5.5, we compare the average and maximum stretches that the DSG algorithm produces versus that produced by the greedy algorithm. By the *stretch* of any node v on the multicast tree T, we mean the ratio of its path to the source in T to the ratio of the shortest path to the source in the network G, *i.e.*,

$$\text{Stretch}_T(v) = \frac{d_T(s, v)}{d_G(s, v)}$$

In Fig. 5.6, we compare the weights of the trees generated by the DSG algorithm to those generated by the greedy algorithm and the shortest path algorithm. The y-axis gives the ratio of the weight to the weight of the tree generated by the best known offline algorithm, the KMB Algorithm [94].

Finally, in Fig. 5.7, we plot the fraction of the nodes which are rerouted by the DSG algorithm.

The experimental results clearly support the claims made in previous sections.

The weight of the shortest path trees is at least 20% away from the optimum, while the weight of the trees produced by the DSG algorithm and the greedy algorithm are very close to optimal.

The DSG algorithm has bounded maximum stretch, but the average stretch is very close

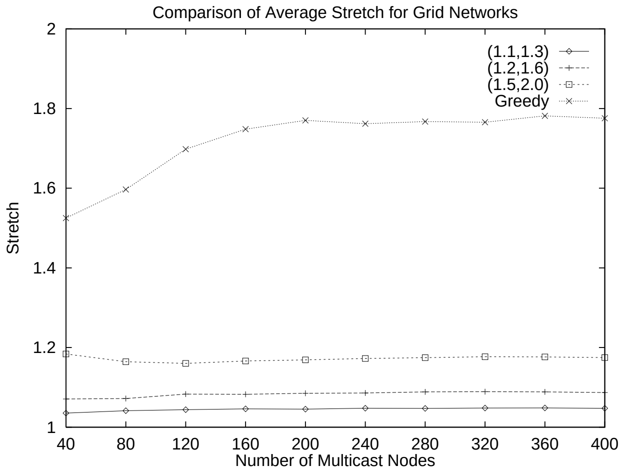

Figure 5.8: Comparison of Average Stretches for 200 × 200 Grid Networks.

to one. This compares well with the stretch of the shortest path tree. The greedy algorithm, however, does very poorly in this regard.

Observe also that there is a tradeoff between the weight of the tree our algorithm produces, and its stretch. By tweaking the values of α and β, we can reduce the stretch at the expense of increasing weight, or vice versa.

The DSG algorithm guarantees that no node is rerouted more than once. But, the results clearly show that the fraction of nodes actually rerouted is much smaller(below 20%). This fraction can be reduced by increasing the values of α and β.

5.4.2 Grid Graphs

Here, we consider a rectangular grid of size $n \times n$. All points on the grid are nodes in the graph, and the only edges are horizontal or vertical edges between adjacent grid points. All edges have the same length.

We run the DSG algorithm on 200 × 200 grids. The source is chosen at random. We add nodes chosen at random in increments of 40, till we reach 400 nodes, which is 1% the total number of nodes. The paths we add between any two nodes are made to closely resemble straight line-segments, *i.e.*, for connecting any node to any other node, we use a uniform step-like path. For this purpose, we use the standard *Bresenham's Algorithm* from computer graphics.

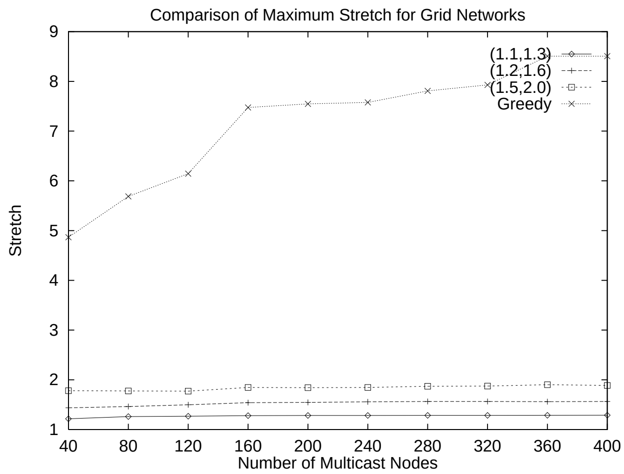

Figure 5.9: Comparison of Maximum Stretches for 200 × 200 Grid Networks.

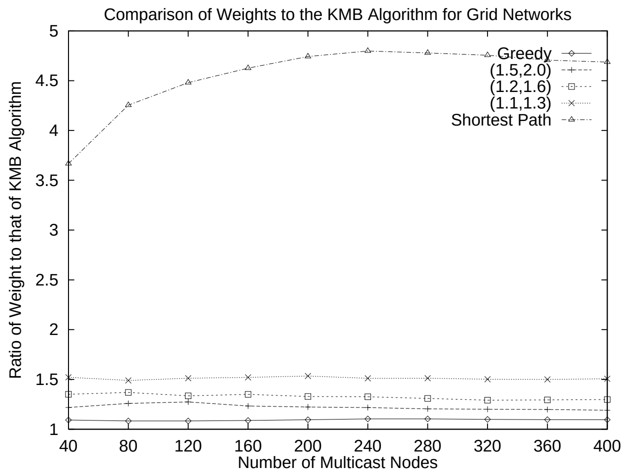

Figure 5.10: Comparison of Weights of the Trees for 200 × 200 Grid Networks.

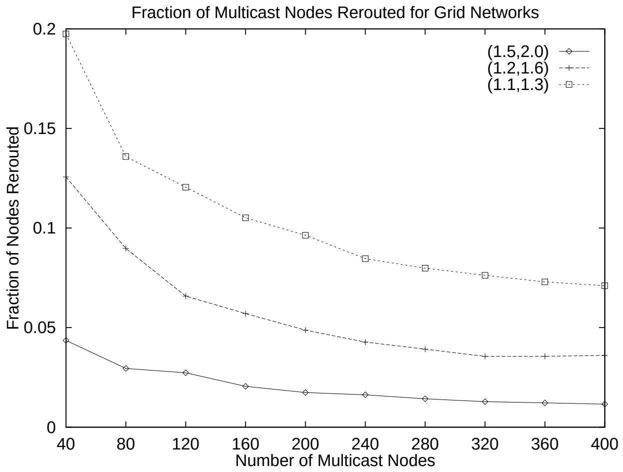

Figure 5.11: Fraction of the Multicast Nodes which are rerouted by the DSG algorithm for 200×200 Grid Networks.

The motivation behind considering such networks is that they closely resemble the simple completely connected network model described in the lower bound construction in section 5.2.1. The paths we use for joining nodes to each other are nearly straight lines. The length of the lines is measured in the l_1 norm(since the distance between two nodes is the *Manhattan Distance*), instead of the usual Euclidean norm. But, this does not change the analysis given in section 5.2.1.

Our results are summarized in figures 5.8, 5.9, 5.10, and 5.11. It is clear that the greedy algorithm produces trees with much worse stretches, and the shortest path algorithm produces trees with much worse weights than the corresponding trees for Waxman Networks. This conforms with the analysis presented in section 5.2.1, where it was shown that the weight of the shortest path tree could be $\Theta(\sqrt{k})$ times the greedy solution, where k is the number of nodes in the multicast.

These results reinforce the fact that even for simple networks, the greedy and shortest path algorithms could perform very badly in terms of the delays and total cost respectively.

5.5 Conclusions

Shortest path routing strategies such as DVMRP, PIM, and CBT produce good multicast trees for delay critical applications with small bandwidth requirements but do not scale well for bandwidth intensive applications. If the bandwidth requirement is large, it becomes more important to optimize the total cost of the multicast tree. The greedy routing algorithm does a good job of optimizing this quantity, but does not produce trees with good end to end delay between the multicast source and receivers. We give an algorithm that has a small running time and communication overhead, and which guarantees a small total cost of the multicast tree as well as small end to end delays from the source to each receiver.

Despite producing trees with small cost and delays, the DSG algorithm as it stands has a few drawbacks. First, we need to perform some amount of rerouting whenever a node gets added or deleted from the system. We show both theoretically and experimentally that the total amount of rerouting is small, but it would be good to see if we can obtain the same performance guarantees as the DSG algorithm without rerouting. Secondly, every router need to maintain state for every (source, multicast group) pair, which implies that the state of the routers will not scale well with the number of groups and sources. Finally, the DSG algorithm works only if the delay metric is proportional to the cost metric. All these issues need to be addressed before the algorithm can be deployed on a large scale.

5.6 Acknowledgements

The graphs in this chapter were obtained by Kamesh Munagala.

Chapter 6

Multicast Routing and Admission Control[1]

6.1 Introduction

Future high-speed communication networks such as ATM will use bandwidth-reservation in order to achieve Quality of Service (QoS) guarantees. Given a request for a Virtual Circuit (VC), the router has to either accept or reject this request and, if it decides to accept it, allocate the requested bandwidth along a path connecting the endpoints of the VC.

In case of multicast requests, the bandwidth has to be allocated along a tree spanning the nodes participating in the multicast group. In the general case, a multicast request specifies the user (endpoint) and the multicast group that this user wants to participate in. The router should either reject the request or accept it and allocate bandwidth along a path connecting the new endpoint with the already existing tree for this group.

In this chapter we present the first polylog-competitive algorithm for the general multicast problem. Our algorithm is randomized since it is impossible for any deterministic algorithm to achieve polylog competitive ratio. The ratio of the number of requests accepted by the optimum offline algorithm to the expected number of requests accepted by our algorithm is $O((\log n + \log\log \mathcal{M})(\log n + \log \mathcal{M})\log n)$, where $\mathcal{M}$ is the number of multicast groups and n is the number of nodes in the graph. If each vertex is allowed to serve at most one multicast group, the competitive ratio simplifies to $O(\log^3 n)$.

[1] This chapter represents joint work with M.R. Henzinger and S. Plotkin, and was published as [59].

Unicast Routing Routing and admission control problems in the online setting were extensively studied. Two related performance measures were considered. In the *congestion* model, the algorithm is required to accept all of the requests, and the goal is to reduce the maximum edge congestion (utilization). Here the congestion is allowed to exceed 100%. In the *throughput* model the algorithm is allowed to reject some of the requests and is not allowed to exceed 100% congestion on any link. The goal is to maximize the total bandwidth-duration product for all the accepted (routed) requests.

The first competitive algorithm in the congestion model for general topology networks was presented in [6]. The competitive ratio of this algorithm is $O(\log n)$, where n is the number of nodes in the network. The first competitive algorithm in the throughput model was given in [52] for the case of a single-link network and extended in [53] for a line network. Competitive algorithm for general topology networks in the throughput model was presented in [11]. This algorithm achieves $O(\log nT)$ competitive ratio, where T is the maximum duration (holding time) of a virtual circuit. The competitive ratio improves to $O(\log n)$ for the Permanent Virtual Circuits case, i.e. circuits with infinite holding times. The algorithm assumes that each circuit specifies its holding time upon arrival. It is impossible to achieve polylogarithmic competitive ratio if the holding times become known only upon termination of the circuit [14, 101].

Routing in a probabilistic model where there are assumptions on the distribution of call arrivals times and source-destination pairs was considered in [84]. The results in this chapter are incomparable to the results mentioned above since they are based on the *edge-independence* assumption stating that the random variables describing instantaneous load on edges are independent. This assumption is not satisfied in general topology networks. The randomized model without the independence assumption was considered in [78]. In the case where the durations are exponentially distributed and the arrivals are Poisson with unknown rates, their algorithm achieves a $(1+\epsilon)$ competitive ratio, where ϵ depends on the ratio of the minimum capacity to maximum bandwidth of a single VC. Both [11] and [78] assume at least logarithmic ratio between maximum VC bandwidth and minimum link capacity. Similar results without this assumption were developed for special network topologies (see e.g. [90]).

All of the above algorithms are related to the (offline) combinatorial approximation algorithms for multicommodity flow [88, 96, 116, 79]. As in these algorithms, the basic idea is to assign each edge a cost that is proportional to an exponent of the congestion on this edge, and try to route along short paths with respect to this cost.

Multicast Routing The techniques in the above mentioned papers can be used to solve several restricted multicast problems in the throughput model. In particular, [11] shows that if the participants in a single multicast group arrive together("batch arrivals"), and the accept/reject decision is for the whole multicast group, it is possible to achieve $O(\log n)$ competitive ratio. Roughly, the idea is to route the multicast request along a minimum-cost Steiner tree instead of along a shortest path. The case where we keep the restriction of batch arrivals, but allow rejection of some of the group members and acceptance of others can be solved by replacing an approximation algorithm for Steiner tree with an approximation algorithm for the k-MST problem. The first polylogarithmic and constant factor approximations for this problem were presented in [9, 26], and a 3-approximation algorithm was given in [55]. This algorithm is relatively slow and is using polynomial number of calls to an approximation algorithm for the Prize Collecting Traveling Salesman problem [64]. The technique developed in this chapter can be directly used to reduce calculations to a single call to the Prize Collecting Traveling Salesman algorithm.

Recently, Awerbuch and Singh [13] have shown how to combine the "winner-picking" technique [10] with the techniques in [11] to achieve a polylog competitive ratio for the case where members of each multicast group arrive *sequentially*, i.e. the size and membership of the group is unknown upon its creation. Their algorithm can deal only with the *non-interleaved* case, i.e. when all the members of a particular multicast group arrive before a new group can be created.

The algorithm in [13] is not applicable in the general case, where the arrivals of requests belonging to different multicast groups are *interleaved*. The main problem is that this algorithm depends on the fact that, at every instance, the algorithm is dealing with the construction of only a single multicast tree and all accept/reject decisions with respect to all existing multicast groups are already known. As in [11], the algorithm in [13] uses edge costs that are exponential in the current link load. One of our contributions is a new definition of edge-costs that are independent of the specific accept/reject decisions made with respect to each multicast group. This decoupling between multicast groups is what allows us to generalize the algorithm in [13] and to combine it with techniques in [11] to achieve a polylog competitive ratio for the general multicast problem.

Lower Bounds A natural question to ask is if it is possible to make the competitive ratio independent of $\mathcal{M}$, the number of multicast group. We address this issue by showing a lower bound of $\Omega(\log \mathcal{M} \log n)$ when $\mathcal{M}$ is much larger than the link capacities. This is the first bound for this problem that is stronger than $\Omega(\log n)$ and that works even if we require that the bandwidth requested by each multicast is significantly smaller than bandwidth of a single link. A similar lower bound for the case where a multicast is allowed to request a constant fraction of a link bandwidth was shown in [10]. For the case where a multicast can request bandwidth equal to bandwidth of a single link, a polynomial lower bound was shown in [19].

It is interesting to note that the algorithm presented in this chapter works even against a *semi-oblivious* adversary, i.e. the adversary is allowed to look at the tree used by the online algorithm to service a multicast group only after all the requests for that group have been processed[2]. We show that no algorithm can do well against an adaptive online adversary.

As is customary for online algorithms, previous papers on multicast ignored the issue of computational complexity. In particular, the algorithm in [13] assumes an NP-hard computation at each routing decision. We show that it is possible to use the special properties of the Prize-Collecting Steiner tree algorithm in [64] to implement each step of our algorithm in polynomial time; the same technique also allows us to give a polynomial time, $O(\log n)$-competitive algorithm for selective batch arrival multicast requests (See Section 6.9.3 for a description of the problem).

In Section 6.2 we introduce the model and the terminology. Section 6.3 describes the algorithm, and Section 6.5 presents the proof of the competitive ratio. The algorithm as described in Section 6.3 may violate capacity constraints on edges. In Section 6 we show that the above algorithm can be slightly modified to guarantee that capacity constraints are never violated. In section 7 we remark that several of the assumptions made in the earlier sections can be relaxed. Lower bounds are presented in Section 6.8. Section 6.9 explains how to implement each decision step of our online algorithm in polynomial time and also addresses the selective batch arrival problem.

[2]A semi-oblivious adversary is at least as powerful as an oblivious adversary.

6.2 Model and Definitions

A request to join a multicast group specifies the group, the node that wants to join, and the amount of the requested bandwidth. The multicast routing and admission algorithm can either reject this "join" request or accept it and allocate the requested bandwidth along some path from the new node to the current tree associated with the requested multicast group. The algorithm is not allowed to allocate above link capacity.

We model the network as a capacitated graph with n nodes and m edges. For simplicity, we will assume that all edges have capacity u and all requests are for unit bandwidth. We also assume that the number of multicast groups $\mathcal{M}$ is known in advance. The issue of removing some of the assumptions is deferred to Section 6.7.

We also assume that $u \geq \log \mu$, where μ is a parameter that is polynomial in $n, \mathcal{M}$ defined later.[3] We assume that multicast groups, once established, never leave.

6.3 The Algorithm

The online algorithm can be viewed as consisting of $L = \log n + \log \mathcal{M}$ "virtual" algorithms for each one of the $\mathcal{M}$ multicast groups. We call these algorithms virtual because the routing and accept/reject decisions of these algorithms are not implemented. Instead, they only modify internal data structures and, in particular the cost associated with each edge. The description of the cost computation is deferred to Section 6.4. For now, it is sufficient to assume that each edge has an associated cost that is deterministic, depends only on the input sequence of requests, and is monotonically non-decreasing in time.

The jth virtual algorithm associated with the ith multicast – $\mathrm{VA}_{i,j}$ – is shown in Figure 6.1. The goal of $\mathrm{VA}_{i,j}$ is to build a tree $T_{i,j}$. This tree spans some of the nodes that requested to be added to the ith multicast group and that are already spanned by trees $T_{i,k}$, for $k < j$. In other words, a request is first generated at $\mathrm{VA}_{i,1}$; if it immediately gets added to $T_{i,1}$, it is passed on to $\mathrm{VA}_{i,2}$, etc.

Each request to join the ith multicast group is considered as a potential unit of profit, and the virtual algorithms use ("consume") this profit to "pay" for their trees. $\mathrm{VA}_{i,j}$ can expand its tree $T_{i,j}$ by adding a subtree only if it can pay for this subtree. We will refer to these subtrees as "fragments". As payment, $\mathrm{VA}_{i,j}$ can use only the profit that is on

[3] This requirement corresponds to a similar requirement in [11].

VA$_{i,j}$

A `Initialization (`s_i` is the root of the `i`-th multicast):`

1 $T_{i,j} \leftarrow \{s_i\}$.

2 **for** all u

$\textsc{Profit}_{i,j}(u) \leftarrow 0$.

$\textsc{Used-Profit}_{i,j}(u) \leftarrow 0$ *(*$\textsc{Used-Profit}_{i,j}$ *is used only for the analysis, specifically in Lemma 6.3)*

B `Invoked due to receiving profit `π` at node `v` from VA`$_{i,j-1}$` (VA`$_{i,0}$` is invoked with unit profit due to a join request at node `v`):`

1 $\textsc{Profit}_{i,j}(v) \leftarrow \textsc{Profit}_{i,j}(v) + \pi$.

2 `Contract `$T_{i,j}$` to a single vertex `s.

3 `Find smallest `$\pi' \leq \textsc{Profit}_{i,j}(v)$` such that `$\exists$` tree `$S$` with the following properties:`
`(i)`$v \in S, s \in S$ `(ii) Cost(`S`)` $\leq (\sum_{u \in S, u \neq v} \textsc{Profit}_{i,j}(u) + \pi')/d$.

4 **if** such π' found

4.1 `Let `S` be the tree which satisfies the conditions in Step 3.`

4.2 **for** all $u \in S, u \neq v$

$\textsc{Used-Profit}_{i,j}(u) \leftarrow \textsc{Profit}_{i,j}(u)$;

$\textsc{Profit}_{i,j}(u) \leftarrow 0$.

4.3 $\textsc{Used-Profit}_{i,j}(v) \leftarrow \pi'$;

$\pi \leftarrow \textsc{Profit}_{i,j}(v) - \pi'$;

$\textsc{Profit}_{i,j}(v) \leftarrow 0$.

4.4 `Uncontract `$T_{i,j}$;

$T_{i,j} \leftarrow T_{i,j} \cup S$.

4.5 `Update the cost of each edge `$e \in S$.

4.6 `Pass profit `π` at node `v` to VA`$_{i,j+1}$.

Figure 6.1: The j-th *Virtual* phase of the i-th *Real* algorithm. Recall that d is the density value defined at the beginning of Section 6.3.

Real(i)

1 `Choose` $\eta_i \in [1 \dots L]$ `such that` $\texttt{Prob}(\eta_i = j) = \beta \cdot 2^j$.

2 `Follow the computation of` $\texttt{VA}_{i,\eta_i}$. `When an edge gets added to` T_{i,η_i}, `add that edge to the` *real* `tree used to service the` i`-th multicast group.`

Figure 6.2: The *Real* algorithm for multicast group i.

the nodes of this subtree and that was not used by $\mathrm{VA}_{i,k}$ for $k < j$ (This is denoted by $\textsc{Profit}_{i,j}(v)$ in Figure 6.1). More precisely, $\mathrm{VA}_{i,j}$ monitors the profit passed from $\mathrm{VA}_{i,j-1}$. Each time it gets π units of profit at some node v, it adds π to $\textsc{Profit}_{i,j}(v)$. It then tries to find a fragment that includes v such that the ratio of the unused profit associated with nodes of this fragment plus π' is at least $d = d^T \cdot \frac{1}{6L \log n}$ times the cost of adding this fragment to $T_{i,j}$, where $d^T = 1/(mu)$ and $\pi' \leq \textsc{Profit}_{i,j}(v)$. The goal of the algorithm is to minimize π'.

This subtree is added to the $T_{i,j}$, d times the cost of this tree is "consumed", and the rest of the profit (in fact, at most one unit) on the newly added nodes is bequeathed to $V_{i,j+1}$.

Observe that, since costs are increasing, the total profit used to construct $T_{i,j}$ is bounded by its final cost divided by d. Since $\mathrm{VA}_{i,j}$ builds its tree in an online fashion, there might be a larger (in terms of the spanned nodes that requested participation in ith multicast) tree that can be constructed offline using the same profit. In Lemma 6.3 we show that this "loss" is not very significant. Also, note that the only way virtual algorithms dealing with different multicast groups interact with each other is through edge costs. Another important property is that for all j, the vertices which contribute towards the profit collected by $\mathrm{VA}_{i,j}$ are a subset of the vertices that contribute towards the profit collected by $\mathrm{VA}_{i,j-1}$.

The "real" algorithm is shown in Figure 6.2. For each multicast group, it randomly chooses one of the virtual algorithms and implements the construction of the tree built by this virtual algorithm. We set the probability of choosing $\mathrm{VA}_{i,j}$ to $p_j = \beta \cdot 2^j$, where β is chosen such that $\sum_{j=1}^{L} p_j = 1$.

Observe that if the i-th real algorithm has chosen $\mathrm{VA}_{i,j}$ for a specific multicast, it does not get the profit for the requests that were taken into account when $\mathrm{VA}_{i,j}$ was constructing its tree. Instead, it will get the profit for the later requests. In particular, it will get the profit that was inherited by $\mathrm{VA}_{i,j+1}$.

6.4 Edge Costs

Our algorithm can be viewed as a generalization of the algorithm of Awerbuch and Singh [13]. The main conceptual difference lies in definition of edge costs. In this section we define the cost metric and the way it is updated as a result of each new request. The cost metric is updated by the virtual algorithms and hence is deterministic.

The online algorithm constructs the cost metric as it goes along. When profit propagates from $\mathrm{VA}_{i,j-1}$ to $\mathrm{VA}_{i,j}$, we consider this an "event". An event might cause $\mathrm{VA}_{i,j}$ to consume some profit and update its tree $T_{i,j}$. Let $c_e(k)$ denote the cost of edge e after the k-th event. When the kth event occurs, the virtual algorithms use costs $c_e(k-1)$ for making their decision. These decisions are then used to compute $c_e(k)$ in a deterministic fashion.

Let $\vec{\eta} = (\eta_1, \ldots, \eta_{\mathcal{M}})$ represent the indices of the virtual algorithms chosen for the $\mathcal{M}$ multicasts. Also, let $p_{\vec{\eta}}$ represent the probability of making this sequence of choices. Define the load on an edge as $1/u$ times the number of trees it was used in by the real algorithm, and let $\lambda_e^{(\vec{\eta})}(k)$ represent the load on edge e after the first k events have occurred, where $\vec{\eta}$ represents the choices made by the *Real* algorithms. Since the random choices of the *Real* algorithms for different multicasts are independent, $p_{\vec{\eta}} = \prod_{i=1}^{\mathcal{M}} p_{\eta_i}$.

Let $c_e(0) = u$ for each edge e. Suppose costs $c(0)$ to $c(k-1)$ were already computed. Then $c_e(k)$ is computed as follows.

$$c_e(k) = u \sum_{\vec{\eta}} p_{\vec{\eta}} \mu^{\lambda_e^{(\vec{\eta})}(k)}$$

The value of μ is set to $4m^6 \log^2 \mathcal{M}$. The reason for this value will become clear in Section 6.6. Observe that, given $\vec{\eta}$, the expression $\lambda_e^{(\vec{\eta})}(k)$ is deterministic, and hence the costs $c_e(k)$ are deterministic as well.

Define $X_e^{(i,j)}(k)$ as indicator variables, with $X_e^{i,j}(k)$ being 1 if edge e is used by $\mathrm{VA}_{i,j}$ during the first k events and 0 otherwise. Notice that $X_e^{(i,j)}(k)$ are deterministic quantities. Now, $\lambda_e^{(\vec{\eta})}(k) = (1/u) \cdot \sum_{i=1}^{\mathcal{M}} X_e^{(i,\eta_i)}(k)$. We can use this to rewrite the cost $c_e(k)$:

$$c_e(k) = u \sum_{\vec{\eta}} \prod_{i=1}^{\mathcal{M}} \left(p_{\eta_i} \cdot \mu^{X_e^{(i,\eta_i)}(k)/u} \right)$$

Interchanging the order of the summation and the product, we get

$$c_e(k) = u \prod_{i=1}^{\mathcal{M}} \sum_{j=1}^{L} p_j \mu^{X_e^{(i,j)}(k)/u} \tag{6.1}$$

The above representation gives an easy way to compute $c_e(k)$ efficiently. Since only one of the sums changes during any event, the online algorithm can recompute that sum and obtain the new costs.

The following claim follows from the way we construct the cost metric.

Claim 6.1 *The cost $c_e(k)$ is the expectation of the quantity $u\mu^{\lambda_e(k)}$ where $\lambda_e(k)$ is a random variable representing the load on edge e after k events.*

6.5 Proof of competitiveness

In order to prove the competitive ratio, we will divide the multicast groups into "profitable" and "unprofitable", based on the cost of the *optimum* trees for these groups with respect to the cost metric constructed by our algorithm. Here, by optimum trees we mean the trees constructed by the optimum offline algorithm.

Consider the ith multicast group, and let the number of requests satisfied by the optimal offline algorithm be $r^*(i)$. Similarly, let $r(i)$ be the profit obtained by the online algorithm. Let $w^*(i)$ be the cost (in the final cost metric) of the tree T_i^* used by the optimum algorithm to service multicast group i. We call a multicast group *profitable* if the optimal's tree for this multicast group has a high profit to cost ratio in the *final* cost metric:

Definition 6.2 *The i-th multicast group is* profitable *if $\frac{r^*(i)}{w^*(i)} \geq d^T$, where $d^T = \frac{1}{mu}$.*

We use the quantities R^* and R to represent $\sum_{i=1}^{\mathcal{M}} r^*(i)$ and $\sum_{i=1}^{\mathcal{M}} r(i)$, respectively. Let P and U represent the set of profitable and unprofitable multicast groups, respectively. Also, we define $R_P^* = \sum_{i \in P} r^*(i)$ and $R_U^* = \sum_{i \in U} r^*(i) = R^* - R_P^*$.

We first show (Lemma 6.4) that the online algorithm obtains almost as much profit from profitable groups as the optimal solution. Then we show that the total profit obtained by the online algorithm can only be poly-logarithmically smaller than optimal's profit from unprofitable groups. To prove the latter claim, we take an indirect route. We use capacity constraints to argue that the quantity mR_U^* is bounded by the sum of the final costs of all edges (Lemma 6.5). Finally, we bound the final costs in terms of the expected profit obtained by the online algorithm (Lemma 6.9).

Consider the quantities $\textsc{Profit}_{i,j}(v)$ and $\textsc{Used-Profit}_{i,j}(v)$ at the end i.e. after all requests have been received. Let $\mathrm{P}_{i,j}(v) = \textsc{Profit}_{i,j}(v) + \textsc{Used-Profit}_{i,j}(v)$. Informally, the quantity $\mathrm{P}_{i,j}(v)$ denotes the profit consumed by $\mathrm{VA}_{i,j}$ at node v. For any set X of vertices, $\mathrm{P}_{i,j}(X) = \sum_{v\in X} \mathrm{P}_{i,j}(v)$; the definitions of $\textsc{Profit}_{i,j}$ and $\textsc{Used-Profit}_{i,j}$ are similarly extended.

Lemma 6.3 $\mathrm{P}_{i,j}(T_i^*) \leq 3w^*(i)d\log n$.

Proof. We first bound the quantity $\textsc{Profit}_{i,j}(T_i^*)$. This contribution comes from nodes in T_i^* which do not belong to $T_{i,j}$. The profit consumed on these nodes by $\mathrm{VA}_{i,j}$ must be at most $w^*(i)d$, else these nodes would have formed a fragment on their own and been added to $T_{i,j}$.

Now we bound $\textsc{Used-Profit}_{i,j}(T_i^*)$. This contribution comes from nodes that belong to $T_{i,j}$. Recall that $VA_{i,j}$ acquires $T_{i,j}$ in tree fragments. Consider an Eulerian tour D of T^*. Let a *segment* of tour D be a maximal contiguous piece of D such that all edges of the segment belong to the same fragment of $T_{i,j}$. Initially, all segments are marked active. If two consecutive active segments on this tour belong to the same fragment, they are merged together along with the portion of the tour between them to form a single segment. Let $t(s)$ denote the event at which the edges of segment s were added to $T_{i,j}$.

Furthermore, we define a $pred$ and $succ$ relation on active segments such that $pred(s, D)$ is the predecessor of s in tour D and $succ(s, D)$ is the successor of s in D.

Let $D_0 = D$. For $h \geq 1$, let $\mathcal{H}_h = \{s \text{ is an active segment of } D_{h-1}, t(s) < t(pred(s, D_{h-1})), t(s) < t(succ(s, D_{h-1}))\}$. Let $\mathcal{L}_h$ denote the remaining segments of D_{h-1}, and let D_h denote the tour D_{h-1} with each segment in $\mathcal{L}_h$ marked inactive. The segments in $\mathcal{H}_h$ remain active in D_h. As mentioned above, consecutive active segments are merged if they belong to the same fragment.

Note that for all h:

$$|\mathcal{L}_h| > |\mathcal{H}_h|.$$

This implies that there at most $\log n$ non empty sets $\mathcal{L}_h$. Let $s \in \mathcal{L}_h$ for some h. Also, let s' be the successor or predecessor segment of s in D_{h-1} with $t(s') < t(s)$ and let p consist of the part of D between s and s'.

Assume $\textsc{Used-Profit}_{i,j}(s) > d(w^*(s) + w^*(p))$. Let $v \in s$ be the node with the last request in multicast i among all nodes in s. When the request at v arrived, the sum $\textsc{Profit}_{i,j}(s) = \sum_{u \in s} \textsc{Profit}_{i,j}(u)$ is more than $d(w^*(s) + w^*(p))$, because $\textsc{Profit}_{i,j}(s)$ is the source of $\textsc{Used-Profit}_{i,j}(s)$. Thus, at that time we could have used at most $d(w^*(s) + w^*(p))$ to add $s + p$ as a fragment. Since the algorithm always tries to create a fragment using the minimum amount of profit, we have:

$$\textsc{Used-Profit}_{i,j}(s) \leq d(w^*(s) + w^*(p)).$$

Considering that D visits every node twice it follows that

$$\sum_{s \in \mathcal{L}_h} \textsc{Used-Profit}_{i,j}(s) \leq 2w^*(i)d.$$

Summing over all values of h it follows that the profit consumed by $VA_{i,j}$ from all nodes which belong to $T^* \cap T_{i,j}$ is at most $2w^*(i)d \log n$. This completes the proof of this lemma. □

The above lemma was first proved by [13] with worse constant factors. Our proof is simpler, and allows us to show (Section 6.9) that the algorithm can be implemented in polynomial time.

For profitable multicasts, the profit obtained by our online algorithm is high:

Lemma 6.4 $R \geq R_P^*/2$.

Proof. Since there are L levels, Lemma 6.3 guarantees that the total wasted profit for multicast group i is at most $3Lw^*(i)d \log n$. Plugging in $d = d^T \cdot \frac{1}{6L \log n}$ and using the fact that i profitable implies that $r^*(i) \geq d^T w^*(i)$, we obtain a bound of $r^*(i)/2$ on the wasted profit. Therefore, $r(i) \geq r^*(i)/2$ for all profitable groups i. Summing over all the profitable groups, we get the desired result. □

Having bounded the profit from the profitable groups, we now concentrate on the unprofitable groups. Recall that c_e is the cost of edge e at the end i.e. after all the events have taken place, and that the costs are non-decreasing in time.

Lemma 6.5 $mR_U^* \leq \sum_e c_e$

Proof. Let k_e^* be the number of multicast groups which use edge e in the optimal offline solution. Consider the tree T_i^* used by the optimal to route the i-th multicast group. If this group is unprofitable then by definition $r^*(i) \leq \frac{1}{mu}\sum_{e\in T_i^*} c_e$. We sum this over all the unprofitable multicast groups, and then reverse the order of summation.

$$\begin{aligned} R_U^* &\leq \frac{1}{mu}\sum_{i\in U}\sum_{e\in T_i^*} c_e \\ &\leq \frac{1}{mu}\sum_e k_e^* c_e \\ &\leq \frac{1}{m}\sum_e c_e \end{aligned}$$

The last inequality follows from the fact that the optimal offline is not allowed to exceed capacities, implying $k_e^* \leq u$. □

Let $w_j(i)$ represent the cost incurred by $\mathrm{VA}_{i,j}$ in constructing the tree $T_{i,j}$. In other words, each tree fragment of $T_{i,j}$ contributes to $w_j(i)$ its cost associated with the event of adding this fragment. We use $w(i)$ to denote $w_{\eta(i)}(i)$, where η represents the choice of the real algorithm. Let $r_j(i)$ represent the profit consumed in constructing this $T_{i,j}$. The following lemma implies that if the expected profit is small, then the expected cost of the constructed trees is small as well.

Lemma 6.6 $\mathbf{E}(r(i)) \geq (d/2)\mathbf{E}(w(i)) - \frac{1}{\mathcal{M}}$, *where* $w(i)$ *is the cost paid by the* Real *algorithm for multicast group* i.

Proof. If the real algorithm chooses to follow $\mathrm{VA}_{i,j}$, i.e. $\eta(i) = j$, then it will get at least the profit used by $\mathrm{VA}_{i,j+1}$. Therefore:

$$\mathbf{E}(r(i)) \geq \sum_{j=1}^{L-1} p_j r_{j+1}(i).$$

By definition, $p_j = p_{j+1}/2$, and hence

$$\mathbf{E}(r(i)) \geq \sum_{j=2}^{L} p_j r_j(i)/2.$$

By construction:

$$\mathbf{E}(w(i)) = \sum_{j=1}^{L} p_j w_j(i) \leq (1/d)\sum_{j=1}^{L} p_j r_j(i).$$

Thus, we have

$$d \cdot \mathbf{E}(w(i)) \leq 2\mathbf{E}(r(i)) + p_1 r_1(i).$$

Now notice that $r_1(i)$ can be at most n, since each request brings in one unit of profit, and there can be at most n requests for a single multicast group. Also, $p_1 \sum_{j=1}^{L} 2^{j-1} = 1$, which implies that $p_1/2 < 2^{-L}$. Substituting $L = \log n + \log \mathcal{M}$, we obtain $\mathbf{E}(r(i)) \geq (d/2)\mathbf{E}(w(i)) - \frac{1}{\mathcal{M}}$.

We remark that in this proof the fact that the VAs are deterministic is quite crucial; otherwise, the profits $r_j(i)$ would be conditioned on the random choices made by the real algorithms and the above argument would break down completely. $\square$

Now we prove that if the expected cost of the constructed trees is small, then the total cost of all the edges is small as well. But first, we need to prove the following technical lemma. Roughly speaking, this lemma implies that if an event caused an edge to be used by one of the trees, the increase in the cost of this edge is proportional to its current cost.

Consider an event k that caused $\mathrm{VA}_{i,j}$ to augment its tree, and let E_k represent the set of edges of the newly added subtree.

Lemma 6.7 *For all $e \in E_k$, $c_e(k) - c_e(k-1) \leq \frac{\log \mu}{u} p_j c_e(k-1)$. For the edges $e \notin E_k$, $c_e(k) = c_e(k-1)$.*

Proof. The second part of the lemma is obvious. We concentrate on edges $e \in E_k$. By definition of the indicator variables, $X_e^{(i,j)}(k-1) = 0$ and $X_e^{(i,j)}(k) = 1$. Using Equation 6.1, we have:

$$\begin{aligned} c_e(k) - c_e(k-1) &= p_j(\mu^{X_e^{(i,j)}(k)/u} - 1)u \prod_{i' \neq i} \sum_{j'} p_{j'} \mu^{X_e^{(i',j')}(k)/u} \\ &= p_j(\mu^{1/u} - 1) \frac{c_e(k-1)}{\sum_{j'} p_{j'} \mu^{X_e^{(i,j')}(k-1)/u}} \\ &\leq p_j(\mu^{1/u} - 1) c_e(k-1). \end{aligned}$$

The last inequality above follows from the fact that $\sum_{j'} p_{j'} \mu^{X_e^{(i,j')}(k-1)/u} \geq \sum_{j'} p_{j'} = 1$. For all x between 0 and 1, $2^x - 1 \leq x$. Therefore, $\mu^{1/u} - 1 = 2^{(\log \mu)/u} - 1 \leq (\log \mu)/u$, which completes the proof of the lemma. $\square$

Let $W = \sum_i w_i$ represent the total cost of the trees constructed by the online algorithm. The following lemma relates the cost incurred by the algorithms and the final cost of the edges.

Lemma 6.8 $\frac{\log \mu}{u}\mathbf{E}(W) \geq \sum_e (c_e - u)$.

Proof. Let $\Delta_e(k) = c_e(k) - c_e(k-1)$ represent the increase in cost on edge e during the kth event. Clearly, $c_e = c_e(0) + \sum_k \Delta_e(k)$ where the summation is over all events and $c_e(0) = u$ for all edges e. Now, let VA_{i_k, j_k} be the virtual algorithm that updates its tree during event k. Lemma 6.7 implies that

$$\sum_e (c_e - u) \leq (\log \mu / u) \sum_i \sum_j p_j \sum_{k: i_k = i, j_k = j} \sum_{e \in E_k} c_e(k-1).$$

Using definition of $w_j(i)$, we can rewrite this expression as follows:

$$\begin{aligned} \sum_e (c_e - u) &\leq (\log \mu / u) \sum_i \sum_j p_j w_j(i) \\ &= (\log \mu / u) \sum_i \mathbf{E}(w(i)). \end{aligned}$$

Using linearity of expectations, $\sum_i \mathbf{E}(w(i)) = \mathbf{E}(\sum_i w_i))$, which completes the proof. □

We are now ready to show that if the obtained profit is small, then the total cost of all the edges is small as well.

Lemma 6.9 $(5\frac{d^T}{d} \log \mu) m \mathbf{E}(R) \geq \sum_e c_e$

Proof. Summing up Lemma 6.6 over all multicast groups, we have:

$$\frac{2}{d}\left(\mathbf{E}(R) + \sum_{i=1}^{\mathcal{M}} \frac{1}{\mathcal{M}}\right) \geq \mathbf{E}(W).$$

As we will show below, $\mathbf{E}(R) \geq 1$. Therefore, the above inequality can be rewritten as $\frac{4}{d}\mathbf{E}(R) \geq \mathbf{E}(W)$. Using Lemma 6.8, and the fact that $m d_T u = 1$, we obtain

$$\begin{aligned} \sum_e c_e &\leq \frac{\log \mu}{u} \cdot \frac{4}{d}\mathbf{E}(R) + mu \\ &= m \log \mu \left(\frac{4 d_T}{d}\mathbf{E}(R) + \frac{u}{\log \mu}\right). \end{aligned}$$

To complete the proof, it remains to show that the first $u/\log \mu$ requests are always accepted, i.e. $\mathbf{E}(R) \geq u/\log \mu$. Suppose the first $k < u/\log \mu$ requests have been accepted. As a result, the load on each edge is no more than k, and the cost of servicing the next request can be at most $mu\mu^{k/u} < mu\mu^{\frac{1}{\log\mu}} = 2mu$. By construction, the profit needed to pay for this cost is at most

$$2mud = 2mu\frac{1}{mu}\frac{1}{6L\log n} = \frac{1}{3L\log n}$$

Thus, the unit of profit brought by this request is enough to pay for extending the trees of all VA algorithms dealing with the corresponding multicast group. Thus, this request is going to be accepted by the real algorithm as well. In other words, if there are less than $u/\log\mu$ requests generated by the adversary then the *Real* algorithm accepts them all and has a competitive ratio of 1. Else, R (and therefore $\mathbf{E}(R)$) is greater than $u/\log\mu$, which completes the proof of the claim. □

Combining Lemma 6.5 and Lemma 6.9 with Lemma 6.4, we obtain the following result:

Theorem 6.10 $R^*/\mathbf{E}(R) = O(\log n \log\mu(\log n + \log\mathcal{M}))$

Section 6.9 explains how to implement each decision step of our online algorithm in polynomial time.

6.6 Capacity Constraints

In the previous section we showed that the algorithm accepts a significant fraction of the requests accepted by the optimum offline algorithm. It remains to show that our online algorithm does not overflow the available capacities. To that end, we set $\mu = 4m^6\log^2\mathcal{M}$. Note that, by Theorem 6.10, this implies that we get an $O(\log n(\log n + \log\log\mathcal{M})(\log n + \log\mathcal{M}))$-competitive algorithm. For the special case where each node is allowed to serve at most one multicast group, we clearly have an $O(\log^3 n)$-competitive algorithm.

We now show that the above value of μ is sufficient to ensure that the capacity constraints are never violated with high probability.

Lemma 6.11 *For any edge e, the cost c_e does not exceed $u\mu^{1/2}$.*

Proof. Suppose $c_e(k) > u\mu^{1/2-1/u}$ for some k. Since $\mu = 4m^6\log^2\mathcal{M}$ and $u \geq \log\mu$, we get $c_e > um^3\log\mathcal{M}$. Since maximum profit of a single tree fragment is n, this cost is above maximum profit divided by d. Thus, this edge will never be used again by any VA. The claim follows from the fact that during any one event, $c_e(k)$ can increase only by a factor of $\mu^{1/u}$. □

Lemma 6.12 *With probability at least $1 - 1/m^2$, no edge violates its capacity constraint.*

Proof. Claim 6.1 states that c_e is equal to the expected value of the quantity $u\mu^{\lambda_e}$, where λ_e is the final load on an edge. The event $\lambda_e \geq 1$ implies that $u\mu^{\lambda_e} \geq \mu^{1/2}\mathbf{E}(u\mu^{\lambda_e})$. Using Markov inequality, the probability of this event happening is at most $\mu^{-1/2} < 1/m^3$. Therefore, with probability at least $1 - 1/m^2$, all edges satisfy the capacity constraints. □

If the algorithm tries to exceed capacity of an edge, we terminate it. Lemma 6.12 guarantees that this does not affect the competitive ratio given in Theorem 6.10.

6.7 Relaxing some of the assumptions

Some of the assumptions made in Section 6.2 can be relaxed using techniques from [11]. In particular, we can handle the case where the bandwidth requirements of different multicast groups, the capacities of different edges, and the profits of different multicast groups can vary arbitrarily. The modifications to our algorithm, analysis, and competitive ratio go exactly as in [11].

6.8 Lower bounds

The competitive ratio of our algorithm holds against a *semi-oblivious* adversary – the adversary is allowed to look at the multicast tree generated by the online algorithm but only after all the requests for that multicast group have been processed. A *semi-oblivious* adversary is at least as powerful as an oblivious one, so our competitive ratio holds against an oblivious adversary as well. For the purpose of proving lower bounds it is desirable to use the less powerful adversary; accordingly we first show a lower bound of $\Omega(\log \mathcal{M} \log n)$ against an oblivious adversary. Then we show that a polynomial lower bound on the competitive ratio of any online algorithm against an adaptive online adversary.

6.8.1 Against an oblivious adversary

A lower bound of $\log n$ for the problem studied in this chapter immediately follows from [11]. The challenge in the online multicast problem is to decide which requests to service ("winner picking") and how to route a request ("online routing"). We now show how to combine a lower bound for winner picking [10] with the lower bound for online routing [11] to achieve a lower bound of $\log \mathcal{M} \log n$ for the online multicast problem. This is the first lower bound stronger than $\log n$ for the online multicast problem if the bandwidth requested by each multicast can be significantly smaller than bandwidth of a single link.

Theorem 6.13 *No algorithm for selective online multicast can have a competitive ratio better than $\Omega(\log(\mathcal{M}/u)\log n)$ even against an oblivious adversary, and even when the requests are non-interleaved.*

Proof. The basic idea behind the winner picking lower bound for online multicast is the following: Assume $\mathcal{M}$ multicasts are created, but both the online and the offline algorithm are just allowed to pick one. A multicast consists of at least one and up to $\log \mathcal{M}$ classes, each class consisting of c requests for some parameter c. Half of the multicasts, chosen randomly from all multicasts, consist of exactly c request. One fourth of the multicasts, chosen randomly from the remaining half of the multicasts, consist of exactly $2c$ request, etc. Thus, the expected profit of online is $2c$, while the expected profit of offline is $c \log \mathcal{M}$.

The lower bound for online routing works in phases: There are $\log n + 1$ phases, with the "profit", i.e. number of requests, doubling in each phase. It can be shown that there must be a phase such that the expected profit that online has received so far is at most $2/\log n$ of the profit that is available in the current phase. In this phase, offline services all the request, i.e., takes all the profit, and the sequence of requests terminates.

We show next how to combine these two bounds. To simplify the presentation we assume that all demands and all edge capacities are 1, but it is permissible to satisfy a fractional demand and obtain a fractional profit (the profit for a multicast group is the product of the satisfied demand and the number of satisfied requests). We explain later how this result carries over to our model.

We restrict ourselves to values of $\mathcal{M}$ such that $\sqrt{n} > \log \mathcal{M}$. Consider the graph G on $n+2$ vertices (see Figure 6.3) which is defined as follows. The vertex set is $\{r, x, v_1, \ldots, v_n\}$. There is an edge from r to x, and there is an edge from x to each of $v_1 \ldots v_n$. For convenience, define $M = \mathcal{M}/\log n$ and $N = n/\log M$. Notice that the restriction we have placed on $\mathcal{M}$ implies that $N > \sqrt{n}$.

The adversary operates in at most $\log N$ phases: we describe the i-th phase, $1 \leq i \leq \log N$. In phase i the adversary divides the vertices $v_1 \ldots v_n$ into classes of size 2^{i-1}. Notice that there must be at least $\log M$ classes. The adversary then generates M multicasts, each with r as the root. The requests for these multicasts will be non-interleaved. For each multicast, the adversary generates a request at each of the nodes in the first class. Then the

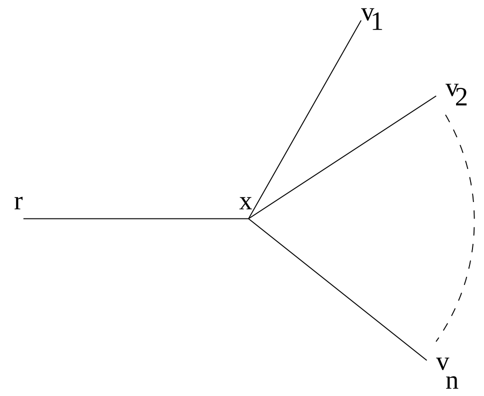

Figure 6.3: The lower bound graph for Theorems 6.13 and 6.16

adversary flips a coin. If the coin toss is a Head (ie. with probability half) the adversary moves on to the next multicast. Else, it generates a request at each node in the next class, flips another coin, and repeats the same process again. If requests have been generated at $\log M$ classes for the same multicast, the adversary moves on to the next multicast. At the end of all M multicasts for this phase, the adversary moves on to the next phase. Notice that setting the class size to 2^{i-1} is equivalent to doubling available profit by 2 for each phase.

Let $c(i)$ be the capacity on the edge (r, x) used by the online algorithm during phase i. Also, let $p(i)$ be the profit obtained by online during the i-th phase. Let $p^*(i)$ and $c^*(i)$ be the corresponding quantities for the solution generated by the oblivious adversary. Notice that $\sum_i c(i)$ can be at most 1. Define $S(k) = \frac{1}{2^k} \sum_{1 \le i \le k} \mathbf{E}(p(i))$. The total expected profit obtained by the algorithm in the first k phases is $2^k S(k)$.

The following two claims now hold:

Claim 6.14 $\mathbf{E}(p(i)) < 2^i \mathbf{E}(c(i))$.

Proof. Suppose the online algorithm decides to satisfy a fractional demand of x for a specific multicast in the i-th phase. The cost incurred is x. Suppose that this commitment is made by the algorithm after the j-th request for this multicast group comes in. Then the expected profit from this multicast group is $2^{i-1} \cdot x \sum_{j \le j' \le M} 2^{j'-j} < 2^i x$. Now we sum this up over all the multicast groups in phase i to get the desired result. □

Claim 6.15 *During any phase i, the adversary can ensure that* $\mathbf{E}(p^*(i)) \ge \frac{\log M}{4} 2^i c^*(i)$.

Proof. During phase i, the adversary can pick the multicast with the maximum number of classes of requests. Let $P^{<}(i)$ denote the probability of this number being less than i. Now, $P^{<}(i) = (1/2+1/4+\ldots 2^{1-i})^M = (1-2^{1-i})^M$, for $i < \log M$. Clearly, $P^{<}(\frac{2}{3}\log M) < 1/3$.[4] This tells us that the expected number of classes is greater than $\frac{\log M}{2}$. To complete the proof of this claim we observe that each class in the i-th phase has 2^{i-1} requests. □

We now prove that there exists a phase k such that the total expected profit obtained by the online algorithm during the first k phases is no more than $2^{k+1}/\log N$. Suppose this is not true. Then, $S(i) > 2/\log N$ for all i. In particular, $\sum_{1\le i\le \log N} S(i) > 2$. But $\sum_i S(i) = \sum_i \mathbf{E}(p(i)) \sum_{i\le j\le \log N} \frac{1}{2^j} \le 2\sum_i \mathbf{E}(p(i))/2^i$. Using Claim 6.14, we have $\sum_i \mathbf{E}(c(i)) > 1$. But this is a contradiction, as the online is not allowed to overflow capacities. This proves the existence of a phase k with $S(k) \le 2/\log N$.

The oblivious adversary cannot see the coin tosses of the online algorithm but it can compute in advance the quantities $S(i)$. Having found the value k guaranteed by the above argument, the adversary stops after phase k and does not generate any more multicast requests. The adversary also generates a 'good' solution as follows: It does not satisfy any demands in the first $k-1$ phases, and in the last phase, it uses up the entire edge (r, x). Now from Claim 6.15, $\mathbf{E}(p^*(k)) \ge 2^{k-2}\log M$. The total expected profit obtained by the online algorithm is $2^k S(k) \le 2^{k+1}/\log N$. This gives a lower bound of $\Omega(\log M \log N)$ on the competitive ratio of any online algorithm. Since $N \ge \sqrt{n}$ and $M = \mathcal{M}/\log n$, this is also a $\Omega(\log \mathcal{M} \log n)$ lower bound.

In the above analysis, we assumed that $\sqrt{n} > \log \mathcal{M}$. This is not a very restrictive assumption, because for $\log \mathcal{M} > \sqrt{n}$, our proof shows that the competitive ratio is already as bad as $\Omega(\sqrt{n})$.

Now we adapt this lower bound proof to our model. Assume that the capacity is u. Let $\mathcal{M}$ be the number of multicasts, and let $\mathcal{M}' = \mathcal{M}/u$. The adversary proceeds as before, except that each phase gets repeated u times. Also, the online algorithm is restricted to satisfy the entire demand of 1 unit or none at all. The same calculation as done above gives a lower bound of $\Omega(\log \mathcal{M}' \log n) = \Omega(\log(\mathcal{M}/u)\log n)$. □

[4] This is a very loose statement. $P^{<}(\frac{2}{3}\log M)$ is much,much smaller, but we do not need a stronger bound.

6.8.2 Against an adaptive-online adversary

The next obvious question to ask is whether any algorithm can work well against a more powerful adversary. We answer this question in the negative in this section. Recall that an adaptive-online adversary is one which can adapt the input sequence depending on the response of the online algorithm; however the adversary must also generate a solution as it goes along.

Theorem 6.16 *No randomized algorithm for selective online multicast can have better than* $\Omega(\min(n, \frac{\mathcal{M}}{u}))$ *competitive ratio against an adaptive-online adversary. The lower bound holds even when the requests are non-interleaved.*

Proof. Consider the same graph as in the previous section (see Figure 6.3). Each edge has a capacity u, all requests have demand 1. The value of $\mathcal{M}$ is fixed at $2nu$. The adversary works in $2u$ phases.

During the i-th phase, the adversary chooses a number m_i uniformly at random from the set $\{1, \ldots, n\}$. The adversary then does the following for at most $m_i - 1$ sub-phases. It generates a new SET-UP request at r. It then generates, in sequence, requests at nodes $v_1, \ldots, v_n$ for the newly set up multicast group. As soon as the online algorithm accepts any of these requests, the adversary aborts this phase completely and moves to the next phase. If the online algorithm does not accept any of these n requests, the adversary moves to the next sub-phase. The adversary does not accept any request during the first $m_i - 1$ sub-phases. If all $m_i - 1$ sub-phases end without the online algorithm having accepted even one request, then the adversary generates yet another SET-UP request at node r. It then generates requests in sequence at nodes $v_1, \ldots, v_n$ and accepts them all without bothering about the online algorithm (if the edge $< r, x >$ is already full in the adversary's network, the adversary rejects these requests). During any phase, the online algorithm gets a profit of n if it 'guesses' correctly the value m_i, and at most 1 otherwise. Since m_i is chosen uniformly at random from $\{1, \ldots, n\}$, the expected profit for the online algorithm during any one phase is at most 2.

At the end of these $2u$ phases, the adversary is guaranteed to have accepted un requests. The expected number of requests accepted by the online algorithm can be no more than $4u$. This proves the desired result. Notice that the requests generated by the adversary are non-interleaved. If each node is allowed to serve no more than one multicast group, the above argument can be modified to obtain a $\Omega(\sqrt{n})$ lower bound for $u = O(\sqrt{n})$. $\square$

6.9 Making our algorithm run in polynomial time

There are two issues we need to address to make our algorithm run in polynomial time. First, in step 3 of $\mathrm{VA}_{i,j}$ (Figure 6.1), we need to compute the minimum profit needed out of the new request to create an appropriate tree fragment. Second, we need to provide a polynomial time approximation algorithm for finding the fragment in step 3 of $\mathrm{VA}_{i,j}$.

The first problem can be solved by doing a binary search. Let π be the available profit on the node where the new request has been generated. We run a maximal dense tree algorithm with all π units of profit available. If a non-empty tree is returned, we try with half the profit, and so on. The key observation here is that we can afford to incur an additive error of $1/(8nL)$ in finding the minimum profit π'. Therefore, the binary search needs to be done only up to a depth $\log 8\pi nL$. The competitive ratio remains unaffected by this modification.

The second problem is resolved using the prize collecting Steiner tree algorithm of Goemans and Williamson [64] which has some very nice properties [63, 55].The prize collecting Steiner tree problem is the following: given a cost metric on the edges, and a penalty function defined on the vertices, find a tree such that the sum of the cost of the edges in the tree and the penalty of the vertices not in the tree is minimum. We already have a cost metric defined on edges. For a vertex v with available profit π_v, the penalty of missing v is set to π_v/d. Now the approximation algorithm [64] for the prize collecting Steiner tree problem is invoked, and the tree fragment returned by this algorithm is used. If d is set to half its current value, we obtain the same bound on the competitive ratio (except the loss of a factor of two).

6.9.1 The Prize-Collecting Steiner Tree Problem

The problem is the following: Given a graph $G(V, E)$ with a cost function c defined on the edges, a set of destinations D, and a prize function π defined on D, find a tree T that minimizes

$$\sum_{e \in T} c(e) + \sum_{v \in D, v \notin T} \pi(v).$$

We use C_T to denote the first component of the cost and P_T for the second component. This problem is NP-hard, being a strict generalization of the Steiner tree problem. A

2-approximation for this problem was given by Goemans and Williamson [64]. Their approximation generates trees T which have the following nice property:

$$C_T + 2P_T \leq 2(C_{T^*} + P_{T^*}) \tag{6.2}$$

where T^* is the optimal tree. This property was first used by Goemans and Kleinberg [63] and a proof of this property is presented in [55]. This property guarantees, as we will see, that with appropriately defined prizes, the prize collecting Steiner tree approximation given by [64] is also a good enough approximation to the maximal dense subtree problem.

6.9.2 Using the prize collecting Steiner tree problem to make our algorithm polynomial time

Suppose the prize collecting algorithm is run with $\pi(u) = \textsc{Profit}_{i,j}(u)/d$, returning a tree T with cost w and profit r, where the quantities $\textsc{Profit}_{i,j}(u)$ are as defined in step 3 of Figure 6.1. Property 6.2 gives the following two results.

Claim 6.17 $w \leq 2r/d$

Claim 6.18 *Let T' be any tree, with cost w' and profit r'. Then $(r' - r)/d \leq w'$.*

In particular, Claim 6.18 implies that if T is empty then for any tree T', $w' \geq r'/d$. Now we make the following changes to the algorithms $\mathrm{VA}_{i,j}$ defined in Figure 6.1. First, we define $d = d^T \cdot \frac{1}{32L \log n}$, which is half of the previous value. Also, instead of finding the maximal dense subtree, $\mathrm{VA}_{i,j}$ repeatedly invokes the prize collecting algorithm, contracts the tree found, and runs the prize collecting algorithm again, till the prize collecting algorithm returns an empty tree. Claims 6.17 and 6.18 are now sufficient to guarantee the competitive ratio in Theorem 6.10.

6.9.3 The selective batch arrival problem

Awerbuch, Azar, and Plotkin [11] show how to solve the semi-selective batch arrival problem: all JOIN requests arrive in batches and they must be either all accepted or all rejected. They use a cost metric very similar to the one used by us in this paper. Each time a new request arrives, they find the minimum Steiner tree (within a factor of 2) that can satisfy these requests. If this tree is cheap enough, they accept the multicast and update the costs.

We make the following modification: assign to each vertex that wants to be a part of the multicast a prize equal to the profit from that vertex, and then run a single instance of the prize collecting algorithm. The tree so obtained is used to service the current multicast. Claims 6.17 and 6.18 are sufficient to give the same (up to constant factors) bound as [11].

Chapter 7

An Approximation Algorithm for the Directed Steiner Tree Problem[1]

7.1 Introduction

The Steiner tree problem is defined as follows: given a graph $G = (V, E)$ with a cost function c on the edges, and a subset of vertices $X \subseteq V$ (called *terminals*), the goal is to find a minimum cost tree that includes all the vertices in X. The cost of the tree is defined as the sum of the costs of the edges in the tree. Note that the tree may include vertices not in X as well (these are known as Steiner vertices). The Steiner tree problem is NP-Complete even when the graph is induced by points in the plane [54], and is MAX SNP-hard [24] for general graphs. The undirected version has been very well studied and algorithms achieving constant factors have been given by several authors [94, 127, 137, 23, 82].

The directed version of the Steiner tree problem is a natural extension of the undirected version and is defined as follows. Given a directed weighted graph $G = (V, A)$, a specified root $r \in V$, and a set of terminals $X \subseteq V(|X| = k)$, the objective is to find the minimum cost arborescence rooted at r and spanning all the vertices in X (in other words r should have a path to every vertex in X). The Steiner tree problem and its variants have several applications in network design and network routing. For example, multicasting involves the distribution of the same data from a central server to several nodes in the network and

[1] This chapter represents joint work with M. Charikar, C. Chekuri, T. Cheung, Z. Dai, S. Guha, and M Li, and was published as [29].

the problem is to choose a set of edges (or communication links) of minimum cost for the server to route the data. The connection between this problem and the Steiner tree problem is clear. For a general survey on the use of Steiner tree problems in networks, see [135]. Most of the theoretical work till now has focussed on the undirected versions of the Steiner problems. However, there are many networks in practice where the communications links are asymmetric and cannot be modeled by undirected edges. The problem of multicast routing in asymmetric networks has received considerable attention recently [120, 123].

From a theoretical point of view, the directed version of the Steiner tree problem is very useful for the following reason. A wide variety of problems involving connectivity and covering, in both directed and *undirected* graphs, can be reduced to this problem in approximation preserving ways. These include the group Steiner tree problem [121], node weighted Steiner tree problem [66, 89], several interesting problems in connected domination, namely edge weighted connected dominating sets, group (or set) TSP, node weighted Steiner connected domination, and others (see [65] for some of the reductions). These problems arise in several different applications and no non-trivial approximation algorithms were known for any of these. All of the above problems are known to be hard to approximate within a logarithmic factor from reductions via set cover. Our result on directed Steiner tree gives a unified approximation algorithm for these problems. In addition, from a hardness of approximation point of view our result gives some evidence that these problems are in the class of problems approximable to within polylogarithmic factors in polynomial time. In fact for the group Steiner tree problem on undirected graphs, Garg et al. [56] have very recently obtained a poly-logarithmic approximation.

A fairly easy reduction from the Set cover problem shows that it is hard to approximate directed Steiner tree to a factor better than $\ln k$ where k is the number of terminals unless $NP \subseteq DTIME[n^{O(\log\log n)}]$ [46]. It is also easy to obtain an approximation factor of k, by connecting every terminal to the root via a shortest path. The only known polynomial time approximation algorithm even for a special case is due to Zelikovsky [138], where he gives an approximation algorithm which achieves a ratio of $(2+\ln k)^{i-1}k^{1/i}$ for any $i>0$ for directed *acyclic* graphs. Zelikovsky further conjectures that no subpolynomial approximation guarantees are possible unless $P = NP$. In this chapter, we present a factor $i(i-1)k^{1/i}$ approximation algorithm which runs in time $O(n^i k^{2i})$ for any $i>0$ for the directed Steiner tree problem. Not only is our approach simpler compared to that of Zelikovsky [138], but also generalizes to arbitrary directed graphs. Further, setting $i = \log k$, we obtain an

$O(\log^2 k)$ approximation in $O(n^{3\log k})$ time. Our result gives some evidence that the problem can be approximated to within poly-logarithmic factors in polynomial time contrary to Zelikovsky's conjecture. Many of the problems we mentioned earlier which reduce to the Steiner tree problem are also hard to approximate to within a logarithmic factor. Thus our results have implications for the complexity of those problems as well. For the directed generalized Steiner tree problem we present an algorithm which achieves an approximation ratio of $O(k^{\frac{2}{3}}\log^{\frac{1}{3}} k)$ where k is the number of pairs that need to be connected.

Our Contribution: A nearly identical algorithm, to the one presented here, was used earlier by Kortsarz and Peleg [93] to obtain approximation ratios for a related problem on undirected graphs which they call "shallow light trees". Such trees have applications in multicast routing with bounded path lengths.

Our work provided the first correct proof of the results in [93]. The algorithm in [93] can be used to prove a weaker version of the claims. The change in the algorithm is essential to achieve the stronger results as first mentioned in [93].

We extend the application to Directed Steiner trees, providing th first approximation for the most general case. Our results are the first correct, polylogarithmic approximations achieved in quasi-polynomial time. We also provide the first approximations for the directed generalized Steiner network problem.

The rest of the chapter is organized as follows. In Section 7.2 we set up the notation and prove some basic lemmas. Section 7.3 describes the algorithm for the directed Steiner tree problem. We briefly sketch the reductions of some of the other problems in Section 7.4.

7.2 Preliminaries

In this section we will give some basic definitions and lemmas concerning partial solutions and l-level trees. We define a slightly more general version of the directed Steiner tree problem below. We assume without loss of generality that between every pair of vertices u and v, there exists an edge $\langle u, v\rangle$ of cost equal to the shortest path distance from u to v in G.

Definition 7.1 *Given root $r \in V(G)$, an integer k and a set $X \subseteq V$ of terminals with $|X| \geq k$, the problem* D-STEINER(k, r, X) *is to construct a tree rooted at r, spanning any k terminals in X and of minimum possible cost.*

We give an $i(i-1)k^{1/i}$ approximation to the D-STEINER(k, r, X) problem for any $i > 1$. This then implies the same approximation ratio for the directed Steiner tree problem. Our approach gives similar approximation ratios for connectivity problems where it is sufficient to connect any k of a given set of terminals. For example, we can solve the generalization of the k-MST problem [26, 55] to directed graphs, where the objective is to find an arborescence on k vertices of G of minimum cost.

Let $c(e)$ denote the *cost* of edge e and let $c(T)$ denote the *cost* of a tree T, or the sum of the costs of the edges in T. Let $k(T)$ denote the number of terminals in T; in other words $k(T) = T \cap X$.

Definition 7.2 *Define $d(T)$, the* density *of tree T to be the ratio of the cost of the tree to the number of terminals in T; in other words $d(T) = c(T)/k(T)$.*

We will assume without loss of generality that all terminals are at the leaves of any Steiner tree. We can arrange this by connecting a dummy terminal to the real terminal with a zero cost edge. Our algorithm is based on repeatedly finding trees with good density each spanning only a subset of the terminals. The final solution is the union of all these trees. The following definition of a partial approximation procedure and the next lemma relate the quality of the final solution to the density of the trees found at each step.

Definition 7.3 *Define an* $f(k)$-partial approximation procedure *for* D-STEINER(k, r, X) *to be a procedure which constructs a tree T' rooted at r, spanning $1 \leq k' \leq k$ terminals in X such that $d(T') \leq f(k) \cdot \frac{C_{OPT}}{k}$, where C_{OPT} is the cost of an optimal solution to* D-STEINER(k, r, X).

If $\mathcal{A}(k, r, X)$ is a partial approximation procedure for D-STEINER(k, r, X), we can apply $\mathcal{A}$ repeatedly to obtain an approximation algorithm $\mathcal{B}(k, r, X)$ for D-STEINER(k, r, X) as follows. $\mathcal{B}(k, r, X)$ first calls $\mathcal{A}(k, r, X)$. Suppose this returns a tree T_1 spanning k_1 terminals. If $k_1 = k$, T_1 is returned. Otherwise $\mathcal{B}(k, r, X)$ returns the union of T_1 and the tree returned by a recursive call to $\mathcal{B}(k - k_1, r, X - X_1)$ where X_1 is the set of terminals spanned by T_1.

Lemma 7.4 *Given $\mathcal{A}(k, r, X)$, an $f(k)$-partial approximation for* D-STEINER(k, r, X) *where $f(x)/x$ is a decreasing function of x, the algorithm $\mathcal{B}(k, r, X)$ gives a $g(k)$-approximation algorithm for* D-STEINER(k, r, X) *where $g(k) = \int_0^k \frac{f(x)}{x} dx$.*

Proof. We will prove the claim by induction on k. The base case $k = 1$ follows as $f(1) \leq \int_0^1 \frac{f(x)}{x} dx$ (by the decreasing property of $\frac{f(x)}{x}$). Suppose it is true for all values less than k. We will prove the claim for k. Let T_{OPT} be an optimal solution to D-STEINER(k, r, X). Suppose the call to $\mathcal{A}(k, r, X)$ returns tree T_1 rooted at r spanning k_1 terminals, i.e. $k(T_1) = k_1$.

$$\begin{aligned} d(T_1) &= \frac{c(T_1)}{k_1} \leq f(k) \frac{c(T_{OPT})}{k} && (7.1) \\ c(T_1) &\leq k_1 \cdot \frac{f(k)}{k} \cdot c(T_{OPT}) && (7.2) \\ &\leq \left(\int_{k-k_1}^{k} \frac{f(x)}{x} dx \right) c(T_{OPT}) && (7.3) \end{aligned}$$

where the last inequality follows from the decreasing property of $\frac{f(x)}{x}$. If $k_1 = k$, the algorithm returns T_1. For this case, $c(T_1) \leq g(k) \cdot c(T_{OPT})$ proving that the algorithm gives a $g(k)$-approximation.

Suppose $k_1 < k$. Let X_1 be the set of terminals spanned by T_1. Let T_2 be the tree returned by the recursive call to $\mathcal{B}(k - k_1, r, X - X_1)$. Since T_{OPT} spans k terminals in X, it spans at least $k - k_1$ terminals in $X - X_1$. Hence the minimum cost tree on $k - k_1$ terminals in $X - X_1$ has cost at most $c(T_{OPT})$. By the inductive hypothesis, $c(T_2) \leq g(k - k_1) \cdot c(T_{OPT})$, i.e.

$$c(T_2) \leq \left(\int_0^{k-k_1} \frac{f(x)}{x} dx \right) c(T_{OPT}) \tag{7.4}$$

Adding (7.3) and (7.4), we get

$$c(T_1) + c(T_2) \leq g(k) c(T_{OPT})$$

This proves that for this case too, the algorithm gives a $g(k)$-approximation. □

Definition 7.5 *An l-level tree is a tree where no leaf is more than l edges away from the root.*

Lemma 7.6 (Zelikovsky [138]) *For all $l \geq 1$ there exists an l-level tree that provides a $k^{\frac{1}{l}}$ approximation to* D-STEINER(k, r, X).

7.3 Directed Steiner Tree Problem

Before we describe our algorithm formally, we will provide some intuition and an outline of our techniques. The density of a tree can be interpreted as the average cost of connecting a terminal to the root. Lemma 7.4 shows that a partial approximation procedure, which finds a tree with density close to that of the optimal tree, leads to a good approximation algorithm. Our algorithm to find trees of good density is motivated by the following.

A trivial algorithm for the problem is to compute shortest paths from each of the terminals to the root and combine them. It is instructive to consider an example for which this algorithm gives a ratio of k. Consider a graph where there is path of cost C_{OPT} from the root to a vertex v, and zero cost edges from v to each of the terminals. In addition, there are paths of cost $C_{OPT} - \epsilon$ from the root to each of the terminals. The naive algorithm picks each of the shortest paths and does not use the path through v, thus incurring a cost $k(C_{OPT} - \epsilon)$.

Motivated by the above extreme example, we can think of finding subtrees of good density which have the following structure. The tree consists of a path from the root r to an intermediate node v, and v is connected to some set of terminals using a shortest path to each of them. For obvious reasons, we call such a structure a *bunch*. The advantage of choosing such a simple structure is that we can compute, in polynomial time, a bunch with the best density. It is not difficult to see that a 2-level tree can be decomposed into disjoint bunches. Therefore, we can use Lemma 7.6 to claim that the best bunch has density no more than $\sqrt{k}$ times the density of the optimal tree. This immediately gives us a simple $\sqrt{k}$ approximation.

An obvious way to improve the result is to find structures which are more general than bunches and approximate the optimal tree more closely. We obtain a much improved ratio using this approach but the simplest way to describe it is via recursion, which we proceed to do next. We will define a sequence of algorithms $A_i(k, r, X)$ such that A_i gives a $i(i-1)k^{1/i}$-approximation for D-STEINER(k, r, X) and runs in time $n^{O(i)}$. We can set $i = \log k$ to obtain an $O(\log^2 k)$ algorithm in quasi-polynomial time.

Let $T_i(k, r, X)$ refer to the tree returned by $A_i(k, r, X)$. Let $T^{(i)}_{OPT}(k, r, X)$ denote the optimum i-level tree that solves D-STEINER(k, r, X) with cost $C^{(i)}_{OPT}(k, r, X)$, and let $d^{(i)}_{OPT}(k, r, X) = C^{(i)}_{OPT}(k, r, X)/k$ be its density. Let $\langle u, v\rangle$ refer to the edge from u to v in the graph G.

Algorithm $A_i(k, r, X)$
OUTPUT: A tree T rooted at r that satisfies at least k terminals in X.

0 if there do not exist k terminals in X reachable from r, **then return** ϕ.

1 $T \leftarrow \phi$;

2 while k>0

  3 $T_{BEST} \leftarrow \phi$;
  4 for each vertex $v \in V$, and each $k', 1 \leq k' \leq k$
    5 $T' \leftarrow A_{i-1}(k', v, X) \cup \{\langle r, v\rangle\}$;
    6 if $d(T_{BEST}) > d(T')$ **then** $T_{BEST} \leftarrow T'$
  7 $T \leftarrow T \cup T_{BEST}$; $k \leftarrow k - |X \cap V(T_{BEST})|$; $X \leftarrow X - V(T_{BEST})$

8 return T

Figure 7.1: Algorithm $A_i(k, r, X)$

$A_1(k, r, X)$ is simple to describe – it finds the k terminals which are closest to the root and connects them to the root using shortest paths. $A_i(k, r, X)$ repeatedly finds a vertex v and a number $k', 1 \leq k' \leq k$ such that the density of the tree $T_{i-1}(k', v, X) \cup \{\langle r, v\rangle\}$ is the least among all trees of this form. The algorithm $A_i(k, r, X)$ is described formally in Figure 7.1.

We now prove the following lemma for arbitrary i. Recall that we are working with the transitive closure of the original graph.

Lemma 7.7 *The trees T_{BEST} chosen by the algorithm $A_i, i \geq 2$ in step 7 have the following property: $d(T_{BEST}) \leq (i-1) \cdot d^{(i)}_{OPT}(k, r, X)$, where k and X refer to the current values being used by the algorithm A_i.*

Proof. The proof is by induction. Since $A_2(k, r, X)$ actually finds the best density tree, the lemma holds for $i = 2$. We now assume that the lemma holds for all j less than i.

Consider an outgoing edge $\langle r, v\rangle$ in the tree $T^{(i)}_{OPT}(k, r, X)$. Let α_v denote the cost of the edge $\langle r, v\rangle$, and let T_v represent the subtree of $T^{(i)}_{OPT}(k, r, X)$ rooted at v. Notice that T_v must be at most an $i-1$ level tree. We can always add zero cost edges and assume that T_v is exactly an i level tree. Notice that this is why the algorithm does not exclude the vertex while finding its best ratio subtree of lower level. Let β_v be the cost of T_v, and

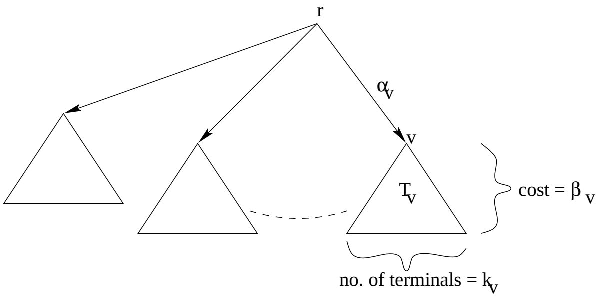

Figure 7.2: The optimum i level tree

$k_v < k$ be the number of terminals satisfied by T_v (see Figure 7.2). Let v be the child of r which minimizes the quantity $d_v = (\alpha_v + \beta_v)/k_v$. It is easy to see that d_v can be no more than $d^{(i)}_{OPT}(k, r, X)$.

Now consider the behavior of the algorithm $A_{i-1}(k_v, v, X)$. Let s_j denote the total number of terminals picked up by $A_{i-1}(k_v, v, X)$ at the end of the jth iteration of the while loop. Since $A_{i-1}(k_v, v, X)$ eventually picks at least k_v terminals, there is an l such that $s_l < k_v/(i-1)$ and $s_{l+1} \geq k_v/(i-1)$. Let $k_1 = s_l$ and $k_2 = s_{l+1}$ and let T_1 and T_2 denote the corresponding trees constructed by $A_{i-1}(k_v, v, X)$ at end of iterations l and $(l+1)$ respectively. At the end of iteration l we observe that $|T_v \cap X| \geq k_v - k_1 \geq k_v - k_v/(i-1) = k_v(i-2)/(i-1)$. Thus since so many terminals are left and there exists a tree of cost at most β_v to connect them, the density of the optimal $i-1$ level tree rooted at v before the start of the $(l+1)$st iteration can be no more than $(\beta_v/k_v) \cdot (i-1)/(i-2)$.

From our inductive hypothesis it follows that the density of the tree found in the $(l+1)$st iteration can be no worse than $(\beta_v/k_v) \cdot (i-1)$. Also notice that the density of the best $(i-1)$ level tree rooted at v can be bounded by the same quantity for the first l iterations as well. Therefore it follows that the density of the tree T_2 at the end of iteration $(l+1)$ is at most $(\beta_v/k_v) \cdot (i-1)$ and further the number of terminals in T_2 is k_2 which is at least $k_v/(i-1)$.

Notice that the algorithm $A_{i-1}(k_2, v, X)$ would produce exactly the tree T_2 described above. When $A_i(k, r, X)$ invokes $A_{i-1}(k_2, v, X)$, it would get back a tree satisfying $k_2 \geq k_v/(i-1)$ terminals. Taking into account the connection cost from r to v, the density of this tree would be at most $\alpha_v/k_2 + (\beta_v/k_v) \cdot (i-1) \leq (i-1)d_v$. Recall that $d_v \leq d^{(i)}_{OPT}(k, r, X)$.

This completes the proof of the inductive hypothesis. □

Theorem 7.8 *If* $i > 1$, $A_i(k, r, X)$ *gives an* $i(i-1)k^{1/i}$ *approximation to* D-Steiner(k, r, X)

in time $O(n^i k^{2i})$.

Proof. We divide the execution of $A_i(k,r,X)$ into stages. Each stage corresponds to one iteration of the outer loop (line 2, Figure 7.1). Let k_j be the number of terminals that still need to be satisfied at the beginning of stage j, and let X_j be the set of unsatisfied terminals. Using Lemma 7.7, we can claim that the density d_j of A_i during stage j is no worse than $(i-1)\cdot C^{(i)}_{OPT}(k_j,r,X_j)/k_j$. We can now invoke Lemma 7.6 to claim that $C^{(i)}_{OPT}(k_j,r,X_j) \le k_j^{1/i} C_{OPT}$. Therefore, $d_j \le (i-1)\cdot C_{OPT}\cdot k_j^{1/i}$. Notice that each stage behaves like an $(i-1)\cdot k_j^{1/i}$-partial approximation to D-STEINER(k_j,r,X_j). Using Lemma 7.4 we obtain the following bound on the cost of the solution produced by $A_i(k,r,X)$.

$$\begin{aligned} cost(T_i(k,r,X)) &\le (i-1)\cdot C_{OPT}\int_0^k \frac{y^{1/i}dy}{y} \\ &= i(i-1)k^{1/i}\cdot C_{OPT} \end{aligned}$$

We observe that A_i invokes A_{i-1} at most nk^2 times and the running time bound follows.

□

Theorem 7.9 $A_{\lceil \log k\rceil}(k,r,X)$ *provides an* $O(\log^2 k)$ *approximation to the directed Steiner Tree problem in time* $O(n^{3\log k})$.

Proof. Follows from Theorem 7.8. □

Zelikovsky conjectures in [138] that directed Steiner tree problem cannot have a subpolynomial approximation guarantee unless $P = NP$. Contrary to his conjecture, Theorem 7.9 gives strong evidence that polylogarithmic approximation in polynomial time is possible.

For many applications of multicasting there is a restriction on the number of hops (the number of edges used in the path from the root to each of the terminals). This problem where the number of hops are bounded by d, has been studied by Kortsarz and Peleg in [93] in the undirected setting where they refer such trees “shallow-light trees”. They prove a lemma very similar to Lemma 7.7. However, their proof is incorrect. They claim an $O(i^2 \log k\cdot k^{1/i})$ approximation for shallow-light trees in time $n^{O(1/i)}$, but in fact their results only guarantee an $O(2^i k^{1/i})$ approximation in time $n^{O(1/i)}$. Based on this for fixed d their algorithm will only yield a $O(2^d \log n)$ approximation and an $O(2^i k^{1/i})$ approximation in the general case. This does not lead to a polylogarithmic approximation in quasi-polynomial time. Our proof of lemma 7.7 can be used to get an algorithm that achieves the bounds claimed in [93]. The algorithm uses repeatedly contracts the tree $A_d(k, r, X)$ in the first case and $A_i(k, z, X)$ in the second. Based on this we can prove the following theorem for shallow-light trees:

Theorem 7.10 *For any fixed $d > 0$, there is an algorithm which runs in time $n^{O(d)}$ and gives a tree of cost at most $O(d \log k)$ times the cost of the best Steiner tree with path lengths bounded by d. Furthermore for arbitrary d, there is an algorithm which achieves an approximation ratio of $i(i-1)k^{1/i}$ and runs in time $n^{O(i)}$ for all $i > 1$. Setting $i = \log k$ we obtain an $O(\log^2 k)$ approximation in quasi-polynomial time.*

7.4 Related Problems

In this section we show how the results in Section 7.3 can be used to obtain $O(k^\epsilon)$-approximations for several related problems which are at least as hard as set cover.

The Group Steiner Tree Problem: The Group Steiner Tree Problem is the following: Given an undirected graph $G(V, E)$, a root r, and a collection of groups $X = \{g_1, \ldots, g_k\}, g_i \subseteq V$, find the minimum cost tree T that connects at least one vertex in each of the groups g_i to the root. This problem arises in wire routing in VLSI design [121]. There is also a directed version of this problem. The undirected (as well as directed) group Steiner tree problem reduces to the directed Steiner tree problem as follows: For each group g_i, introduce a new dummy vertex x_i and connect x_i using zero cost directed edges to each of the vertices in g_i. These dummy vertices are the terminals in a directed Steiner tree instance with the same root. Some other related problems like set TSP on undirected graphs can also be reduced similarly. For the group Steiner tree problem on undirected graphs, Garg et al. [56] have recently obtained a poly-logarithmic approximation using probabilistic approximation of metric spaces, a new and powerful tool introduced by Bartal [17].

Node Weighted Variants: There is a simple approximation preserving reduction between the node weighted and the edge weighted cases when the graph is directed. Therefore all our results carry over to the node weighted cases as well.

The Strongly Connected Steiner Subgraph problem: Given a digraph $G(V, E)$ and $V_s \subseteq V$ where $| V_s |= k$, find a minimum cost subgraph S such that the vertices in V_s are strongly connected in S. It can be approximated to a factor of $2i(i-1)k^{1/i}$ by a simple reduction to directed Steiner tree.

7.5 Conclusions

Our work leaves several interesting open problems. The most important of them is to obtain a polynomial time algorithm with a poly-logarithmic approximation ratio for the directed Steiner tree problem. Our results give strong evidence of the existence of such an algorithm.

Chapter 8

Scheduling Data Transfers and the Set Scheduling Problem[1]

8.1 Introduction

Consider the problem of sending large files (eg. bitmap images) through a general topology network. The requests arrive online and the goal is to eventually satisfy all the requests. Since the bandwidth of the links in the network is limited, it makes sense to try to schedule the transmissions in a way that utilizes the available resources optimally.

In this chapter we consider the *online ftp problem*, which is a formal abstraction of the above file transfer problem. We assume that each ftp request specifies source/destination nodes and the size of the file. The goal of the online algorithm is to choose a path that will be used for transmitting each file, and to decide on the transmission rate. The main difference between this model and the (well-studied) models for online routing and admission control [52, 6, 53, 11] is that here we do not assume that the sources have prespecified transmission rate requirements, i.e. we can deal with non-streaming types of information.

There are two related measures of performance that can be used to compare different algorithms for the online ftp problem. The first measure is the *total flow time*, i.e. the sum over all jobs of the time that elapses between the instant the ftp request is submitted and the time it is satisfied (including the transmission time). The other measure is the

[1]This chapter represents joint work with M. Henzinger, S. Plotkin, and E. Tardos, and was published as [60].

max-stretch, which is the maximum over all ratios of the flow time of each request and the smallest time needed to satisfy this request. The second quantity is determined by the link bandwidth and the size of the file. Both measures are useful since they are directly related to the performance of the network perceived by the end-user.

Let n be the number of requests and P the maximum ratio between the sizes of the files. Assume that the smallest request can be processed in one time unit. Let F^*_{MAX} denote the optimum max-flow i.e. the smallest value for the maximum time a request spends in the system. The main results of the chapter are algorithms that achieve the *optimum max-stretch* and the *optimum total flow time* using resource augmentation[2]. For the max-stretch algorithm we need to increase capacities by a factor of $O(\log P)$, whereas the total flow time algorithm needs a factor of $O(\log F^*_{MAX})$ increased capacity[3]. The latter algorithm does not only achieve the optimum total flow time, but *simultaneously* optimizes many other objective functions, like the maximum flow time, the total square-of-flow-time, etc.

To justify the need for giving larger capacities to the online algorithm (i.e. resource augmentation), we show polynomial lower bounds on both max-stretch and total flow time for the case where both online and offline algorithms are using the same capacities. Moreover, we show that in order to achieve constant competitive ratio against an adaptive adversary we have to give the online algorithm $\Omega(\log P / \log\log P)$ factor more capacity for the max-stretch metric, and $\Omega(\sqrt{\log \gamma / \log\log \gamma})$ more capacity for the total flow metric, where $\gamma = \min\{n, P\}$.

In the context of machine scheduling, total flow time is known to be a hard metric to approximate [83] and it is only recently that progress has been made towards obtaining algorithms that give total flow time guarantees. In particular, logarithmic-factor resource augmentation was used in [113] to obtain optimum flow time for machine scheduling. Unlike the problem we consider, resource augmentation is not necessary to obtain good approximation ratios for minimizing flow time machine scheduling problem; a logarithmic approximation was obtained in [98]. Max-stretch was recently proposed as a good metric to measure user satisfaction [22]. Our lower bound on the amount of resource augmentation needed for max-stretch holds in the machine scheduling model as well, and therefore our upper bounds for max-stretch are also of interest in the machine scheduling model.

[2]Throughout this chapter, when we refer to an optimum solution, we mean the optimum without any resource augmentation.

[3]Note that $F^*_{MAX} \leq nP$ and therefore $\log F^*_{MAX} \leq \log n + \log P$.

When proving upper bounds, we restrict our algorithms to use a single rate when transmitting a specific file, and do not allow preemption. The competitive ratio is computed against an offline algorithm that does not have these restrictions. Our lower bounds for online flow-time minimization algorithm without resource augmentation (i.e. both the online and the offline algorithms work in the same network) hold even if we remove this restriction, i.e. allow the algorithm to use time-varying rate when transmitting a file. This contrasts with minimizing flow time for machine scheduling, where a $\log P$-competitive preemptive algorithm is known [98]. Also, the lower bound for total flow time is achieved using same-size files. This is in contrast to machine scheduling where the unit jobs case is trivial.

We view the online ftp problem as a special case of the *set scheduling problem*. In this problem we have a set of resources and each job requires a specific subset of these resources (or one of a set of subsets). Set scheduling is a natural generalization of the machine scheduling problem that was extensively studied under several different metrics (See [71] for a survey of offline approximation algorithms, and [98, 113, 22, 68, 69, 37, 114] for a sampling of recent results in online algorithms.). The set scheduling model is similar to the parallel jobs model studied by [48, 125]. We show how to apply several techniques developed in the context of machine scheduling to the set scheduling problem (and hence the online ftp problem) for simpler metrics such as makespan and total completion time. In particular, we use the technique that allows us to convert an offline optimization algorithm that maximizes the number of scheduled jobs into an online algorithm that minimizes total completion time [68, 69, 114]. We also develop new techniques that help us attack more difficult metrics such as total flow time and max-stretch.

The techniques developed in this chapter can be better understood when compared to the technique of Hall et al. [68, 69]. There the approach is to use offline ρ-approximation algorithms for offline packing problems to construct $O(\rho)$-competitive online algorithms for average completion time. Our techniques allow the transformation of offline packing algorithms that achieve the optimum packing using $O(\rho)$ resource augmentation into online algorithms that achieve the optimum flow time using $O(\rho{\cdot}\log F^*_{MAX})$ resource augmentation. If the online algorithm is not required to work in polynomial time, then an optimum offline solution ($\rho = 1$) can be used. Unlike the work of Hall et al. [68, 69] our techniques apply only when jobs are *malleable* [68, 28, 48, 125] i.e. extra capacity/resources can be used to reduce the processing time of jobs. Two such problems are the parallel jobs problem [28, 48, 125] and the vector scheduling problem [58, 36, 21]. Using our techniques we can obtain non-polynomial-time online algorithms for minimizing the total flow time for these problems.

In Section 8.2 we explain the models that we use. Section 8.3 contains the main technical contributions of the chapter – the lower and upper bounds on the performance of online algorithms using the total flow time and max-stretch metrics. In Section 8.4 we describe online algorithms for the ftp problem using the makespan and total completion time metrics. Not all online algorithms in Sections 8.3 and 8.4 run in polynomial time; polynomial time online algorithms and offline approximation algorithms are discussed in Section 8.5. Section 8.5 also sketches an offline, polynomial time algorithm for minimizing the makespan for the set scheduling problem (and hence the online ftp problem) if the rate at which a request is serviced is allowed to vary arbitrarily.

8.2 Models and Definitions

In the *set scheduling problem* there are n jobs and m resources. Job j has an arrival time (release date) a_j, a processing time p_j, and a resource requirement S_j where S_j is a subset of S, the set of resources. We define $P = \max_j p_j / \min_j p_j$. The quantity P plays a crucial role in the analysis of our algorithms. As in traditional scheduling, both the preemptive and non-preemptive variants are of interest. The Set Scheduling Problem can be formulated as either an offline or an online problem. As in job shop scheduling and multiprocessor scheduling, the performance of an algorithm for this problem can be studied under several different metrics – most notably makespan, total completion time, total flow time, and max-stretch. In this chapter we will concentrate mainly on online algorithms.

The *online ftp problem* is defined as follows. We are given a network $G = (V, E)$ where all edges have identical bandwidths. Assume that the transmission delay along any link is zero, and that there are no buffers in the network. Once a source starts transmitting data to another node, the other node starts receiving it immediately. Of course the rate at which the sender transmits the data is bounded by the minimum available bandwidth along the route over which the transmission is taking place. Let m be the number of links in the network, and n the number of ftp requests. Request j has an arrival time a_j, specifies file size p_j, and a route R_j over which the data needs to be transmitted. We also address the case where instead of the route, the request specifies only the source and the sink nodes. The former model is closer to the IP world, where the routes are determined by an external algorithm, while the second model is closer to the ATM world, where one can use source routing.

Let C_j be the completion time of job j in a schedule. The quantity $F_j = C_j - a_j$ is called the flow time of job j. The makespan of a schedule is $\max_j C_j$; total completion time is $\sum_j C_j$; total flow time is $\sum_j F_j$ and max-stretch is $\max_j F_j/\tau_j$ where τ_j is the time it would take to satisfy job j if it had the whole network to itself. We also permit jobs to have weights w_j. In the presence of weights the total completion time and total flow time metrics are defined as $\sum_j w_j C_j$ and $\sum_j w_j F_j$ respectively. Traditionally, the total flow time and max-stretch metrics are considered to be the hardest. These are also perhaps the most interesting metrics as they most directly measure end user experience.

The following theorem captures the hardness of the set scheduling problem – the reduction is straightforward and is sketched in Section 8.6 for completeness.

Theorem 8.1 *The Vertex Color problem reduces (via polynomial time reductions) to Minimizing Makespan for Set Scheduling in an approximation preserving fashion.*

For the vertex color problem lower bounds are known for both the approximation ratio ($\Omega(n^{1-\epsilon})$ unless P=NP [47]) and competitive ratio ($\Omega(n^{1/3})$ [19]). The above reduction also holds for the online ftp problem if the routes as well as the transmission rates are given as input. Thus to make progress with the set scheduling/online ftp problems, we need to relax the model. The first relaxation we propose is to allow rate control for jobs. Thus each job would be assigned a start time s_j ($s_j \geq a_j$) and a rate r_j by the scheduler. The job would execute from time s_j to $s_j + p_j/r_j$ and would consume an r_j fraction of each resource in its resource set S_j during this interval. More than one jobs may use a resource at the same time. However, the total usage of a resource at any time must be at most 1. The term *malleable jobs* is commonly used to describe this property [68, 28, 48, 125]. This relaxation is particularly appropriate to the ftp problem: it is possible to control the rate of a TCP connection; more than one connections can use the same link; further, a connection uses up the same bandwidth on each link along its route[4].

[4]Instead of allowing a fixed rate r_j for each job, we could also allow the rate to vary. It turns out that our online algorithms, even though they use just one rate r_j for job j, are competitive against optimal solutions which are allowed to vary the rate. For offline algorithms it may help to vary the rate; we will delve into this a little in Section 8.5

8.3 Flow time and max-stretch using resource augmentation

8.3.1 Upper bounds with resource augmentation but no preemption

Assume that all links have the same capacity in the original network; rescale capacities so that this capacity becomes 1. Further rescale time such that the smallest request takes four units of time to finish if it has the entire network to itself. Then the time required to service the largest request (if the request has the entire network to itself) is at most $4P$.

Let n be the number of requests, and m the number of links. Let $K = 3 + \log n + \log P$. We assume that the online algorithm can use a capacity of $5K$ on each link.

The online algorithm pretends that there are K copies of the network, $G_0 \ldots G_{K-1}$, each with edge capacities 5. We call this algorithm **MRHP** (Most Recent Highest Priority) since at any given time, connections which have been waiting in the system the shortest are the most likely to get scheduled. The online algorithm does its processing only at integral time instants. Figure 8.1 describes the behavior of MRHP at time t such that $t = 2^k.t'$, where t' is odd.

`for` $i = 0$ `to` $\min\{k, K-1\}$

1. `Let` S_i `be the set of requests arriving in the interval` $[t - 2^i, t)$
2. `Find the largest weight subset of` S_i `that can be completed in the network` G_i `between times` t `and` $t + 2^i$ *[Note: This step may not run in polynomial time in general]*
3. `Schedule this subset in` G_i `such that each request has starting time` t`, finishing time` $t + 2^i$`, and a uniform rate during this interval.`

Figure 8.1: Algorithm MRHP at time $t = 2^k.t'$, where t' is odd.

The same job may get scheduled by multiple copies of the network. The flow time of such a job is taken to be the smallest flow time from all its copies. All the jobs ultimately get scheduled by the online algorithm, as the G_{K-1} alone has sufficient capacity to schedule all the jobs.

Let w_j be the weight of the jth job, and F_j the total time this job spends in the system. Let Q_k be the total weight of the requests which get scheduled in at least one of the networks

$G_0 \dots G_k$. Let Q_k^* be the total weight of all requests that have a flow time of at most 2^{k+2} in the optimum solution. Let $q_k = Q_k - Q_{k-1}$, and $q_k^* = Q_k^* - Q_{k-1}^*$ (for convenience define Q_{-1} and Q_{-1}^* to be 0.). Each job j which contributes to q_k must have a flow time $F_j \leq 2^{k+1}$ in the MRHP schedule, and each job j which contributes to q_k^* must have a flow time $F_j^* \geq 2^{k+1}$ in the optimum schedule.

Let $\mathcal{F}^*$ and $\mathcal{F}$ denote the total weighted flow times of the optimum and online algorithms, respectively. Clearly, $\mathcal{F}^* \geq \sum_{0 \leq k \leq K} 2^{k+1} q_k^*$. Further, $\mathcal{F} \leq \sum_{0 \leq k \leq K-1} 2^{k+1} q_k$.

Lemma 8.2 $Q_k \geq Q_k^*$

Proof. Let S_k^* be the set of requests which contribute to Q_k^*. By definition, each of these requests has a flow time of at most 2^{k+2}. Divide time into intervals of the form $[i \cdot 2^k, (i+1) \cdot 2^k)$ for $i \geq 0$. Let $S_k^{(i)^*}$ denote the set of requests from S_k^* which arrive during the i-th interval, and let $Q_k^{(i)^*}$ denote their combined weight. All these jobs are scheduled by the optimum algorithm to finish before time $(i+1) \cdot 2^k + 2^{k+2}$. Hence all these jobs must arrive and finish in the interval $[i \cdot 2^k, (i+1) \cdot 2^k + 2^{k+2})$, which has length $5 \cdot 2^k$. Since G_k has 5 times the original capacity on each edge, and since it has all the jobs in $S_k^{(i)^*}$ available for scheduling during the interval $[(i+1) \cdot 2^k, (i+2) \cdot 2^k)$, it will schedule jobs with a weight of at least $Q_k^{(i)^*}$ during this interval. Summing up over all i, $Q_k \geq Q_k^*$. □

Let g be any function from $\Re^+$ to $\Re^+$. Let $\mathcal{F}_g^*$ denote the optimum value of $\sum_j w_j g(F_j)$ that can be obtained in an unaugmented network, and $\mathcal{F}_g$ denote the corresponding value obtained by MRHP.

Theorem 8.3 *$\mathcal{F}_g \leq \mathcal{F}_g^*$, for all non-decreasing functions g.*

Proof. Let $W = \sum_j w_j$. We define $P(k) = Q_k/W$ and $P^*(k) = Q_k^*/W$. P and P^* are probability measures, and Lemma 8.2 implies that P^* stochastically dominates P. Theorem 8.3 now follows from the properties of stochastic dominance. □

Theorem 8.3 is particularly interesting because it shows that MRHP simultaneously optimizes a very wide class of metrics. In particular, the following results can be obtained as corollaries.

Corollary 8.4 *$\mathcal{F} \leq \mathcal{F}^*$.*

Proof. Let g be the identity function in the statement of Theorem 8.3. □

Corollary 8.5 *Let F_{MAX} denote the maximum flow time (max-flow) in the schedule obtained by MRHP and F^*_{MAX} denote the max-flow in the optimum schedule. Then $F_{MAX} \leq F^*_{MAX}$*

Proof. For $p > 0$, define $\mathcal{F}_p$ to be $\sum_j w_j(F_j)^p$. $\mathcal{F}^*_p$ is defined analogously. Theorem 8.3 implies that $\mathcal{F}_p \leq \mathcal{F}^*_p$ for all $p > 0$. F_{MAX} and F^*_{MAX} are the limiting values of $(\mathcal{F}_p)^{1/p}$ and $(\mathcal{F}^*_p)^{1/p}$ respectively as $p \to \infty$. Therefore $F_{MAX} \leq F^*_{MAX}$. □

The average stretch of a job can be mimicked using a total weighted flow time objective function with appropriate weights. MRHP does not really need to know K in advance – it can maintain an estimate of K and increment this estimate by one if and when the current value of K does not suffice to schedule all the requests. Let F^*_{MAX} be the optimum max-flow for the given sequence of jobs, given that the shortest job takes one unit time to finish if it has the entire network to itself. Notice that $F^*_{MAX} \leq nP$. The following theorem gives a sharper bound on the amount of resource augmentation needed by MRHP.

Theorem 8.6 *MRHP needs $O(\log F^*_{MAX})$ resource augmentation. Further, F^*_{MAX} need not be known in advance.*

The above theorem represents a significant improvement, since n can be arbitrarily large even in a well behaved system with small max-flow. Section 8.5 shows how to implement the algorithm in expected polynomial time with $O(\log n + \log P + \log m)$ resource augmentation.

We now return to the max-flow metric introduced in Corollary 8.5. The max-flow metric (F_{MAX}) is interesting primarily because it relates to the max-stretch metric. We give a simple online algorithm **MMF** (Minimum Max-Flow) that uses only a constant factor resource augmentation. More specifically, MMF uses at most five times the capacity of the original network. MMF assumes that the optimum max-flow is at least T and at most $2T$ (Initially, T is assumed to be the time required to complete the very first job in the original network.). At times t which are multiples of $T/2$, MMF looks at all requests which arrived during the last $T/2$ time units. It then assigns to each of these jobs a rate which is just sufficient for this job to finish in the next $T/2$ time units. If the load on any edge exceeds five times the capacity of that edge in the original network, MMF doubles T, aborts the current phase, and waits till the current time becomes a multiple of the new value of $T/2$. The following theorem subsumes Corollary 8.5.

Theorem 8.7 *The maximum flow time of a job in the schedule produced by MMF is no larger than the optimum max-flow. MMF runs in time polynomial in n, m, and $\log P$.*

Proof. Consider all jobs that arrive in the interval $\left(\frac{T}{2}(i-1), \frac{T}{2}i\right]$. Let the optimum max-flow be F^*_{MAX}. All these jobs must finish in the optimal schedule by the time $(T/2)i + F^*_{MAX}$. If $F^*_{MAX} \leq 2T$ then the total requirement of all the jobs that need edge e can be at most $2T + T/2 = 5T/2$ (recall that the capacity of each edge is one). Hence by *stretching* all these jobs over an interval of length $T/2$, no edge can exceed its capacity by more than a factor of 5. If an edge exceeds capacity by more than a factor of 5, we can conclude that $F^*_{MAX} > 2T$, and hence double our estimate of T. □

We are now ready to present **MMS** (Minimum Max-stretch) which uses $O(\log P)$ resource augmentation and guarantees a max-stretch that is no worse than the optimum max-stretch. We first observe that MMF can be modified to guarantee a max-flow that is at most half the optimum value if the amount of capacity on each edge is ten times that in the original network. Let p_1 be the amount of data transfer required by the first job. MMS bunches incoming requests into (at most $\log P$) classes, with class i containing all requests which have a data requirement in the range $[p_1 \cdot 2^i, p_1 \cdot 2^{i+1})$ (i may be negative as well). There can be at most $2 + \log P$ classes. For requests within class i MMS invokes a separate copy of modified MMF. Thus the resource augmentation needed by MMS is $O(\log P)$. Note that MMS does not need to know P in advance. The fact that the max-flow obtained within each class is at most half the optimum max-flow for that class is sufficient to guarantee that the max-stretch obtained by MMS is no more than the optimum max-stretch. The following theorem summarizes the claims made in the above discussion.

Theorem 8.8 *MMS uses $O(\log P)$ resource augmentation and obtains a max-stretch that is no more than the optimum max-stretch. Further, MMS does not need to know P. MMS runs in time polynomial in n, m, and $\log P$.*

Note that neither MRHP, nor MMF, nor MMS need to get the transmission routes R_j as input.

Theorem 8.9 *MRHP, MMF, and MMS can obtain optimum values for their respective metrics even if the routes R_j are not given as input.*

If routes are not provided as input, MRHP, MMF, and MMS as described above would not run in polynomial time. See theorem 8.17 for the amount of resource augmentation needed by polynomial time algorithms.

8.3.2 Lower bounds with preemption but without resource augmentation

We show that without extra capacity, the competitive ratio of any randomized online algorithm which tries to minimize the total flow time (max-stretch, resp.) for the data transfer problem against an oblivious adversary can not be bounded by any function of the network size. The lower bound for the competitive ratio in terms of the number of jobs, n, is $\Omega(\sqrt{n})$ for both metrics. The quantity P is 1 for the flow-time lower bound, and $\sqrt{n}$ for the max-stretch lower bound. The lower bounds hold even if the online algorithm is allowed to do preemption and use fractional capacities on links but the adversary is not.

Consider the length-3 path $A - B - C - D$. Assume that all 3 links have the same bandwidth, u. Each connection will request the same amount of data, r. We rescale time so that $u = r$ i.e. each request can be serviced in exactly one time unit.

Total flow time: The adversary first tosses an unbiased coin. If the outcome is "Heads" it chooses the link $A - B$ as a special link, else it chooses $C - D$. During the first time step, the adversary generates k requests from A to C and k from B to D. The adversary does not do anything for the next $k - 1$ time units. Then for the next k^2 time units the adversary generates one request per time unit over the special link.

Lemma 8.10 *The expected flow time of any online algorithm on this sequence must be $\Omega(k^3)$, even if preemption is allowed and the online algorithm is allowed to use fractional capacities. Further, the optimum flow time for this sequence is $O(k^2)$ even without using fractional capacities and preemption.*

Proof. During the first k time units, the algorithm can send only k units of data over the edge $B - C$. Hence, k units of data remains unsent at time k. Since the special edge is picked randomly by the adversary and not known to the algorithm, the expected amount of unsent data which needs to traverse the special edge is at least $k/2$. During the next k^2 time units, even if the special edge is kept continuously busy, the expected amount of unsent data waiting to cross the special edge is at least $k/2$. Therefore the expected flow time of the algorithm is at least $k^2 \cdot (k/2) = \Omega(k^3)$. The adversary on the other hand will schedule all the requests that need the special edge during the first k time units to obtain a total flow time of $O(k^2)$. □

Since the number of jobs is $n = 2k + k^2$, the competitive ratio of any online algorithm must be $\Omega(\sqrt{n})$ which does not depend on the network size.

Max-stretch: Again, the adversary first tosses an unbiased coin. If the outcome is "Heads" it chooses the link $A-B$ as a special link, else it chooses $C-D$. During the first time step, the adversary generates 1 request from A to C and 1 from B to D, each of size k; for the next $k-1$ time units the adversary does nothing. Over the next k^2 time units the adversary generates one request of size 1 every time unit over the special link. The proof of the following lemma is similar to the previous one, and is omitted.

Lemma 8.11 *The expected max-stretch of any online algorithm on this sequence must be $\Omega(k)$, even if preemption is allowed and the online algorithm is allowed to use fractional capacities. Further, the optimum max-stretch for this sequence is 2 even without using fractional capacities and preemption.*

The ratio $P = p_{max}/p_{min}$ for this sequence is k. Since the number of jobs is $n = 2+k^2$, the competitive ratio of any online algorithm must be $\Omega(\min\{P, \sqrt{n}\})$ which does not depend on the network size. A lower bound of $\Omega(P^{1/3})$ for the competitive ratio of an online algorithm for the minimum max-stretch problem in the context of machine scheduling was presented in [22].

8.3.3 Lower bounds on the amount of resource augmentation

In this section we give lower bounds on the amount of resource augmentation needed for any randomized online algorithm to achieve a constant competitive ratio. These lower bounds require an adaptive adversary, and assume that the online algorithm is not allowed to preempt requests or change the rate at which a request is being serviced. Notice that our upper bounds all work against adaptive adversaries, and do not preempt requests.

Theorem 8.12 *Against an adaptive adversary, any randomized online algorithm that achieves constant competitiveness for max-stretch must use $\Omega(\min\{n, \log P/\log\log P\})$ resource augmentation.*

Proof. The adversary uses a one link network with capacity 1. Let u be the resource augmentation that the online algorithm uses and let k be a parameter chosen suitably below. The sequence of requests created by the adversary consists of subsequences $A_0, A_1, \ldots, A_f$, for some $f \geq 0$. The beginning of a new subsequence A_i is called a *restart*. Initially $i = 0$. Each subsequence A_i consists of requests of size L_i, one every L_i time units where $L_i = (16uk)^{3u-i}$. Define an *i-phase* to be a time interval between the ith and the i+1st restart during which no new jobs of A_i arrive and no old jobs of A_i are completed by the online algorithm. Since the algorithm is not allowed to vary the rates, the adversary can determine at the beginning of an i-phase how long the i-phase would last if no new job arrived. The adversary also knows the bandwidth utilization of the online algorithm during the i-phase. If the adversary encounters an i-phase that would last at least $L_i/(8u)$ time units and where jobs of A_i use more than $1/3$ units of bandwidth, the adversary increments i and it restarts. If the adversary does not encounter such an i-phase, it stops when A_i consists of k jobs.

Note that whenever the adversary restarts, the bandwidth available to the online algorithm for jobs created after the restart is reduced by at least $1/3$. Thus the adversary restarts at most $3u$ times, i.e. $f \leq 3u$. It can be shown inductively that the optimum algorithm can schedule all jobs in $\cup_{l>i} A_l$ (ie all jobs of size less than L_i) in time at most L_i. Hence delaying the last job of each size by its size gives an algorithm with max-stretch at most 2.

We show next that the max-stretch of the online algorithm is at least k. Let $L_f = (16uk)^{3u-f}$ be the size of the shortest jobs generated by the adversary. When the adversary creates jobs of size L_f no f-phase exists of length at least $L_f/(8u)$ where jobs of A_f use more than $1/3$ units of bandwidth. Since k jobs of size L_f are created, there are at most $2k$ f-phases. The total amount of data of jobs in A_f transferred during f-phases where the jobs in A_f use more than $1/3$ units of bandwidth is at most $2k \cdot L_f/(8u) \cdot u = L_f k/4$. We consider next f-phases where the jobs in A_f use at most $1/3$ units of bandwidth. During the first $2kL_f$ time units of these f-phases at most $2kL_f/3$ data of jobs in A_f is transferred. Therefore the total amount of data of jobs in A_f transferred by the online algorithm during the first $2kL_f$ time units since the last restart is at most $11kL_f/12$. Hence, there are some jobs of A_f left unfinished at time $2kL_f$ and therefore, there must be some job with a stretch of k.

It follows that the competitive ratio is at least $k/2$. Note that the ratio P of the maximum job size to minimum job size is $(16uk)^f$ and that the number n of jobs is at most fk. Since $f \leq 3u$, $n \leq 3ku$ and $P \leq (16uk)^{3u}$. If the competitive ratio is a constant, both $n/3u$ and $P^{1/(3u)}/(16u)$ must be a constant. The first condition translates to $u = \Omega(n)$ and the second translates to $u = \Omega(\log P/\log\log P)$. Therefore

$$u = \Omega(\min\{n, \log P/\log\log P\}).$$

□

Theorem 8.13 *Let $\gamma = \min\{n, P\}$. Against an adaptive adversary, any randomized online algorithm that achieves constant competitiveness for Total Flow Time must use $\Omega(\sqrt{\log\gamma/\log\log\gamma})$ resource augmentation.*

Proof. We define the adversary recursively. Choose a fixed constant $\delta < 1/6$, and let $k > 2$ be a parameter whose value we will specify later. Let u be the amount of resource augmentation; we restrict our attention to those u that are multiples of δ. The network consists of just a single link and the adversary generates requests which need to transfer 1 unit of data over this link; we call these the *long* jobs. Define a basic interval to be of the form $[i/(2u), (i+1)/(2u))$, where i is a non-zero integer. If the online algorithm schedules a job over the duration $[t_1, t_2)$, then we round up t_1 and round down t_2 to be multiples of $1/(2u)$ *without* altering the rate which was assigned to this job by the online algorithm. This can only help the online algorithm. Define a basic interval to be "bad" if the online algorithm uses a capacity of more than δ on the link during this interval.

The adversary generates a long request at time 0. Then, at time $i > 0$, the adversary generates a long request only if the previous time unit $[i-1, i)$ did not contain a bad basic interval. The adversary stops as soon as it sees $k\delta/u$ basic bad intervals or has generated k long requests. Also, during each basic bad interval, the adversary recursively generates the lower bound sequence for $(u-\delta)$ resource augmentation with time scaled down by a factor of $1/(4ku)$.

Let $F^*(u)$ denote the optimal flow time for the above sequence, and $F(u)$ denote the flow time obtained by the online algorithm. Let $N(u)$ and $P(u)$ denote the number of jobs and the ratio of the largest to the smallest job generated by the adversary, respectively. As a base case, we use the lower bound example used in Lemma 8.10[5], where $u = 1$. For this, choose $N(1) = k$. Then $F(1) = \Theta(k^{1.5}), F^*(1) = \Theta(k)$, and $P(1) = 1$.

The optimum way to schedule all the jobs generated by the adversary is to schedule a long job during a time unit which does not contain a basic bad interval. There will never be more than one long job waiting in the system. Further, since there are never more than $k\delta/u$

[5]Lemma 8.10 uses a length three path as opposed to a single edge network used in the current theorem; it is easy to see that the proofs in the current theorem go through even for a length three path. Also, the base case can be modified slightly to work on a length 1 path without changing the lower bounds obtained in the current theorem.

basic bad intervals, the total duration of the schedule is at most $k + k\delta/u < 2k$. Therefore, the total contribution of the long jobs to the flow time is at most $2k$; this also ensures that all the jobs generated during the recursive procedure fit into the basic interval and do not overflow. During each basic bad interval time is scaled down by a factor of $1/(4ku)$, which implies the recurrence

$$F^*(u) \leq 2k + \frac{1}{4ku} F^*(u-\delta)\frac{k\delta}{u} < 2k + F^*(u-\delta).$$

Using the base case, we obtain

$$F^*(u) < 2ku/\delta + O(k) = O(ku). \tag{8.1}$$

Also, $P(u) \leq (4ku)P(u-\delta)$, which simplifies to $P(u) = (ku)^{O(u)}$, and $N(u) = k + (k\delta/u)N(u-\delta) = k^{O(u)}$.

We now provide a lower bound on $F(u)$. First, we study the case in which the adversary terminated before finding $k\delta/u$ basic bad intervals. Since the capacity available to the algorithm is u, any job must originally be scheduled for at least $1/u$ time. By rounding up the start and rounding down the end of an interval of length at least $1/u$ to multiples of $1/(2u)$, the length of the interval decreases by a factor of at most 2. Hence, the total data transferred originally by the algorithm is at most twice the data transfer after the rounding. If the number of bad basic intervals is less than $k\delta/u$ then the total data transfer during these bad intervals is at most $k\delta$. Further, the bandwidth used during the good basic intervals is at most $2k\delta$ during the first $2k$ time units. Hence, the total data transferred originally by the algorithm is at most $2(k\delta + 2k\delta) = 6k\delta$ during the first $2k$ time units. Therefore there is $k(1-6\delta)$ amount of data left unsent at the end of the $2k$ time units; the flow time because of this data must be at least $(k(1-6\delta))^2/(2u) = \Omega(k^2/u)$. If on the other hand, the adversary terminates because it found enough bad basic intervals, then the flow time must be at least $(1/4ku)(k\delta/u)F(u-\delta) = F(u-\delta)/(4u^2)$. Therefore,

$$F(u) = \min\left\{(k(1-6\delta))^2/(2u), F(u-\delta)/(4u^2)\right\}.$$

For large k, the second term is always smaller than the first (inductively) and we obtain

$$F(u) = k^{3/2}/u^{O(u)}.$$

Now, the competitive ratio is $C(u) = \sqrt{k}/u^{O(u)}$. For $C(u)$ to be constant, k must equal $u^{O(u)}$. We now lower bound u in terms of $n = N(u)$ and $P = P(u)$, given that $k = u^{O(u)}$. We can increase k as long as neither of the two conditions $N(u) = k^{O(u)}$ and $P(u) = (ku)^{O(u)}$ is satisfied; hence for a constant competitive ratio either $n = k^{O(u)} = u^{O(u^2)}$, or $P = (ku)^{O(u)} = u^{O(u^2)}$. In other words, if $\gamma = \min\{n, P\}$, then $\gamma = u^{O(u^2)}$, or $u = \Omega(\sqrt{\log \gamma / \log \log \gamma})$.

□

8.4 Online Algorithms for Makespan and Total Completion Time

Standard techniques can be used to obtain constant competitive online algorithms for makespan and average completion time for the online ftp problem without the use of resource augmentation.

Makespan: Define λ as the maximum over all edges, e, of the amount of data that needs to be transferred over e. We rescale time so that one unit of data can be transferred over a link in one unit of time. Let a_{MAX} be the time at which the last request arrives. Let L be the quantity $\max(a_{MAX}, \lambda)$. L is a lower bound on the makespan of any schedule. The online algorithm maintains a guess $\tilde{\lambda}$ for the value of L. We assume that the first request arrives at time 0. The initial value of $\tilde{\lambda}$ is set to p_1, the amount of data transfer needed by the first request. Each time a request arrives, the algorithm recomputes L. If $L > \tilde{\lambda}$, $\tilde{\lambda}$ is reset to $\max(L, 2\tilde{\lambda})$. The online algorithm schedules the newly arrived request to execute from time $\tilde{\lambda}$ to $2\tilde{\lambda}$, with a rate of $1/\tilde{\lambda}$. It is easy to see that the above algorithm does not violate capacity constraints. Let U represent the final value of $\tilde{\lambda}$; by construction U is at most $2L$. The makespan is at most $2U + U + U/2 + \ldots < 4U$. We can now claim the following result.

Theorem 8.14 *The above algorithm is 8-competitive.*

The above algorithm runs in polynomial time and hence, is also an offline approximation algorithm. However, an offline algorithm "knows" the exact value of L and hence can provide an approximation guarantee of 2. If routes are not given as part of the input, a slight variant of the above online algorithm can still obtain an 8-approximation, but it would not run in polynomial time.

Total Completion Time: The general scaling technique outlined by Hall et al. [69, 68] directly results in a 4-competitive online algorithm for the total completion time metric, regardless of whether routes are given as part of the input. Their technique requires an offline algorithm that can pack an optimum number of requests into a given interval. This problem is NP hard, and therefore, our online algorithm does not run in polynomial time. An $O(\log m)$-competitive polynomial time algorithm is outlined in Section 8.5.

8.5 Polynomial Time Approximation and Online Algorithms

In this section we give offline algorithms for total completion time, makespan, total flow time, average stretch, maximum flow time, and maximum stretch that run in polynomial time. The algorithms for total completion time and makespan approximate the optimum performance without resource augmentation. The algorithms for the remaining metrics achieve optimum performance using either a constant-factor or a polylogarithmic-factor resource augmentation. We conclude the section by giving polynomial-time algorithms with optimum makespan under two different relaxations of our model: (1) We relax the condition that the rate of a job has to be constant: we give a polynomial-time algorithm that varies the rates and achieves optimum makespan. (2) We assume that the start time s_j is part of the input and show that then the problem can be solved in polynomial time.

Theorem 8.15 *There exists an algorithm that achieves an $O(\log m)$-approximation of the total completion time for the online ftp problem in time polynomial in n and m, regardless of whether routes are given as part of the input.*

Proof. Consider the problem of maximizing the number of ftp requests (out of a given set of requests, all of which have the same arrival time) that can be scheduled over a given period of time. An $O(\log m)$ approximation to this problem can be obtained using multicommodity flow followed by randomized rounding [119]; plugging this into the general technique of Hall et al. [69, 68] results in an $O \log m)$-competitive polynomial time online algorithm for the total completion time of ftp requests. □

A polynomial time 2-approximation for the makespan of the ftp problem when routes are given as part of the input follows from the discussion in Section 8.4; a simple randomized rounding trick results in an $O(\log m)$-approximation if routes are not provided as input.

Theorem 8.16 *There exists an algorithm that achieves a 2-approximation for makespan in time polynomial in* n *and* m *if routes are given as part of the input, and* $O(\log m)$ *otherwise..*

We now describe how to implement algorithm MRHP in polynomial time. The only step of MRHP which might take super-polynomial time is step 2, finding the largest weight subset A_i of S_i that can be completed between times t and $t + 2^i$. To implement it in expected polynomial time we need to add $\log m + 2eK$ to the capacity of each edge, where $K = \log n + \log P + 3$.

We use first a linear programming relaxation of the problem, then round it probabilistically and finally show that with high probability no edge capacity constraint is violated. The linear program uses for each job j a variable x_j and maximizes $\sum_{J \in S_i} w_j x_j$ under the constraint that for each edge e, $\sum_{j \text{ uses } e} x_j p_j / 2^i \leq 1$ and that for each j, $x_j \geq 0$. Let x_j^* denote the value of x_j in the solution. We probabilistically round each job j for each network i such $P(j \in A_i) = x_j^*$. Let X_e be the random variable denoting the load of edge e in G. The expected value μ of X_e is $\sum_{0 \leq i < K} \sum_{j \text{ uses } e} x_j^* p_j / 2^i \leq K$. Using Chernoff bounds with $\delta = (\log m + 2eK)/\mu - 1$ shows that

$$
\begin{aligned}
Pr(X_e > \log m + 2eK) &\leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu \\
&\leq \left(\frac{e}{\log m + 2eK}\right)^{\log m + 2eK} \\
&< 1/(m(nP)^{2e})
\end{aligned}
$$

Thus, the probability that one of the edge capacities overflows is at most $1/(nP)^{2e}$ in which case we simply redo the rounding step.

Note that both MMF and MMS already run in time polynomial in n, m, and $\log P$ if routes are given as part of the input. The same ideas that we outlined above for total flow time also result in polynomial time algorithms for the max-flow and max-stretch problems when routes are not given as input.

Theorem 8.17 *There exist (online or offline) algorithms that run in time polynomial in* n, m, *and* $\log P$ *and*

- *achieve optimum total flow-time or average stretch with an expected* $O(\log n + \log P + \log m)$*-factor resource augmentation regardless of whether routes are given as part of the input;*

- *achieve optimum maximum flow time with a constant-factor resource augmentation if the routes are given as part of the input, and expected $O(\log n + \log m)$ resource augmentation otherwise;*

- *achieve optimum maximum stretch with an $O(\log P)$-factor resource augmentation if the routes are given as part of the input, and expected $O(\log n + \log m + \log P)$ resource augmentation otherwise.*

We finally relax some of our conditions. Consider first the case that the rate of jobs can vary. Assume that the optimum makespan is M. We present a linear program that given M checks whether there exists a feasible solution. By performing a binary search over M, with $0 < M \leq nP$ and assuming that time is rescaled so that the shortest job takes one time unit, we get a polynomial time algorithm that finds the optimum makespan.

We assume wlog that the first job arrives at time 0. Break the time from 0 to M into intervals whenever a new job arrives and number the time intervals from 1 to n. Let l_i be the length of interval i and let a_{MAX} be the arrival time of the last job. Note that the length of the last interval is $M - a_{MAX}$. There is a variable $x_{j,i}$ for each interval i and each job j. The linear program checks whether there is a non-negative assignment for the variables $x_{j,i}$ such that

1. For each job j, $\sum_i x_{j,i} l_i \geq p_j$,

2. For each edge e and interval i, $\sum_{j \text{ uses } e} x_{i,j} \leq 1$, and

3. For each job j and interval i such that j arrived after i, $x_{j,i} = 0$.

The linear program can be slightly modified to give a polynomial-time algorithm in the case that the start time s_j is given for each job j.

8.6 Reduction of Vertex Color to Set Scheduling

In this section we prove Theorem 8.1:*The Vertex Color problem reduces (via polynomial time reductions) to Minimizing Makespan for Set Scheduling in an approximation preserving fashion.*

Let $G(V, E)$ be an instance of the vertex color problem. We construct an instance of the set scheduling problem with $S = E$. For each vertex v in the original vertex color problem, we introduce a job j_v with arrival time 0, processing time 1, and resource requirement $I(v)$, where $I(v)$ is the set of all edges incident on v. Jobs j_u and j_v need a common resource iff there is an edge between vertices u and v. Thus, jobs j_u and j_v can be scheduled at the same time iff u and v can have the same color. This establishes a one to one correspondence between the makespan of the set scheduling problem and the number of colors needed for the vertex color problem.

Part III

Network Design Layer

Chapter 9

An Approximation Algorithm for the Generalized Directed Steiner Network Problem[1]

9.1 Introduction

The directed Steiner tree problem (Chapter 7) is defined as follows. Given a directed graph $G = (V, A)$, a specified root $r \in V$, and a set of terminals $X \subseteq V(|X| = k)$, the objective is to find the minimum cost arborescence rooted at r and spanning all the vertices in X (in other words r should have a path to every vertex in X). In this chapter we consider the directed generalized Steiner network problem where instead of a root and a set of terminals, we are given a set of k pairs of vertices, and the objective is to find a minimum cost subgraph which connects each pair. In the undirected case, it is easy to see that the optimal solution is a forest but that is not necessarily true in the directed case. The general version has several applications in network design and network reliability though it is only recently that progress has been made in terms of obtaining good approximation algorithms even in the undirected case. The first constant factor algorithms for the undirected case were given in [1, 134]. Agrawal et al. [1] also give $O(\log R)$ approximations for the more general case where each pair has a connectivity requirement r_{ij} (the number of edge disjoint

[1]This represents joint work with M. Charikar, C. Chekuri, T. Cheung, Z. Dai, S. Guha, and M Li, and was published as [29].

paths required for the pair (i, j)) and R is the maximum connectivity requirement; Jain [76] recently gave a 2-approximation for the same problem. In this paper we restrict ourselves to case where $r_{ij} = 1$ for all pairs.

Dodis et al. [44] showed that this problem is as hard as label cover [71] i.e. for all $\gamma > 0$, it is hard to approximate this problem within $2^{\log^{1-\gamma} n}$. Hence, a logarithmic approximation algorithm is unlikely, even in quasi polynomial time. We achieve an approximation ratio of $O(k^{2/3} \log^{1/3} k)$.

9.2 Directed Generalized Steiner Network Problem

Formally the problem is the following. Given a directed graph $G(V, E)$ and a set $X = \{< u_i, v_i >\}$ of k node pairs, find the minimum cost subgraph H of G such that for each node pair $< u_i, v_i > \in X$, there exists a directed path from u_i to v_i in H. The cost of the subgraph H is the sum of the cost of all the edges in H.

As in Chapter 7, we work with a slightly more general problem we call DG-STEINER (k, X) which is the problem of finding a minimum cost subgraph H of G that satisfies at least k node pairs from the set X. Clearly, directed generalized Steiner tree is a special case of DG-STEINER (k, X). In this section, we present an $O(k^{\frac{2}{3}} \log^{\frac{1}{3}} k)$ approximation to the DG-STEINER (k, X) problem.

Let $C_{OPT}(k, X)$ denote the cost of the optimal solution to DG-STEINER (k, X). Define $k(H)$ to be the minimum of k and the number of node pairs in X that get satisfied by H. Further, let $c(H)$ denote the cost of H, and $d(H) = c(H)/k(H)$ be the density of H. We will omit the parameters (k, X) where their values are clear from the context. An $f(k)$ partial approximation to DG-STEINER (k, X) can be defined in a manner similar to that for D-STEINER(k, X).

We assume without loss of generality that between every pair of vertices $< u, v >$, there exists an edge of cost equal to the shortest path distance from u to v in G. The main idea of the algorithm is to repeatedly find good bunches, where a bunch is defined as follows.

Definition 9.1 *Let $Y = \{< u_1, v_1 >, \ldots, < u_p, v_p >\}$ be a subset of X containing $p \leq k$ node pairs. Also let a and b be vertices of G. A* bunch *$B =< a, b, Y >$ is a graph with the vertex set $\{a, b, x_1, x_2, \ldots x_p, y_1, y_2, \ldots y_p\}$. B has an edge $< a, b >$ of cost identical to that of the edge $< a, b >$ in the graph G. B also has edges $< x_i, a >$ and $< b, y_i >$, with costs identical to those of $< u_i, a >$ and $< b, v_i >$ in G.*

Algorithm $\mathcal{C}(k, X)$

$d \leftarrow \infty$; $B \leftarrow \phi$

for all pairs $< a, b > \in V(G) \times V(G)$

  for each pair $< u, v > \in X$, $s[u, v] \leftarrow c(< u, a >) + c(< b, v >)$

  sort X in increasing order of s. Let $< u_j, v_j >$ refer to the jth pair in this sorted list.

  for p going from 1 to k

    $C \leftarrow c(< a, b >) + s[u_1, v_1] + s[u_2, v_2] + \ldots + s[u_p, v_p]$

    if $C/p \leq d$ **then** ($d \leftarrow C/p$; $B \leftarrow < a, b, \{< u_1, v_1 >, \ldots, < u_, v_p >\}$)

return B

Figure 9.1: Algorithm $\mathcal{C}(k, X)$

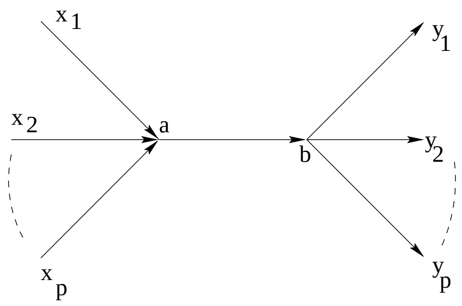

Figure 9.2: The bunch $< a, b, Y >$

Figure 9.2 illustrates the structure of a bunch. Notice that the x_i s and y_i s are unique even though the u_i s and v_i s may not be. In Figure 9.1 we give an algorithm $\mathcal{C}(k, X)$ that finds a bunch of minimum density. We then prove the existence of a bunch $B(k, X)$ that has density no more than $O(d(H_{OPT}) \cdot k^{\frac{2}{3}} \log^{\frac{1}{3}} k)$. This allows us to claim that $\mathcal{C}$ provides an $O(k^{\frac{2}{3}} \log^{\frac{1}{3}} k)$ partial-approximation to DG-Steiner (k, x). We then apply Lemma 7.4 to obtain an approximation to DG-Steiner (k, x).

Finding the best bunch is straightforward – we find the lowest cost bunch for all possible values of a, b, and p and then choose the minimum density bunch out of these (at most $n^2 k$) lowest cost bunches. The proof of the following lemma is quite simple and hence we omit it.

Lemma 9.2 *The algorithm* $\mathcal{C}(k, X)$ *finds a minimum density bunch.*

We now prove the existence of low-density bunches. For convenience, we define $h(k), k > 1$ to be the quantity $6k^{\frac{2}{3}} \log^{\frac{1}{3}} k$. Also, let $h(1) = 1$.

Theorem 9.3 *There exists a bunch $B(k, X)$ such that $d(B) \leq (C_{OPT}/k) \cdot h(k)$.*

Proof. Suppose there exists a node pair $< u, v > \in X$ such that the distance between u and v is no more than $(C_{OPT}/k) \cdot h(k)$. Then $< u, v, \{< u, v >\} >$ is the required bunch, and we are done. Notice that if $k = 1$ then such a node pair always exists. Therefore we will implicitly assume $k \geq 2$ for the rest of the proof.

If there exists no such node pair, then in the optimal solution $H^*(k, X)$ each node pair must be separated by a distance of at least $(C_{OPT}/k) \cdot h(k)$. Also, no node pair can be separated by more than C_{OPT}. For $i \geq 1$, let X_i denote the set of node pairs $< u, v >$ such that the distance from u to v in $H^*(k, X)$ lies in the range $\left[2^{i-1} \cdot h(k) \frac{C_{OPT}}{k}, 2^i \cdot h(k) \frac{C_{OPT}}{k}\right]$. Let I_{MAX} denote the maximum value of i for which X_i is non empty. Since $k \geq 2$, $I_{MAX} \leq \log k$. Therefore there exists an i' such that $|X_{i'}| \geq \frac{k}{\log k}$. Let $H^*_{i'}$ be a minimum cost subgraph of H^* satisfying all node pairs in $X_{i'}$. For each node pair $< u, v > \in X_{i'}$, let $P_{H^*_{i'}}(u, v)$ be the shortest path between u and v in $H^*_{i'}$. If there are multiple shortest paths between u and v, choose one of them arbitrarily.

The sum of the shortest paths between all the node pairs in $X_{i'}$ is greater than $2^{i'-1} h(k) \cdot \frac{C_{OPT}}{k} \cdot (\frac{k}{log k})$. But the cost of $H^*_{i'}$ is no more than C_{OPT}. Thus there has to be an edge which is shared by at least $\frac{h(k)}{\log k} \cdot 2^{i'-1}$ paths. Let this edge be $e =< u, v >$, and let X_e be the set of node pairs $< a, b >$ such that $P_{H^*_{i'}}(a, b)$ passes through e. Each of these paths can be split into three parts, $P(a, u)$, the edge $< u, v >$ and $P(v, b)$. Let H_1 be the union of paths of the form $P(a, u)$ (ie. the first components) and H_2 be the union of paths of the form $P(v, b)$ (ie. the third components). Clearly $c(H_1) \leq C_{OPT}$ and $c(H_2) \leq C_{OPT}$. Let T_1 be a shortest-incoming-path tree (rooted at u) in the graph H_1 and T_2 be a shortest-outgoing-path tree (rooted at v) in the graph H_2.

Let $dist_{T_1}(a, u)$ be the cost of the path from a to u in T_1. $dist_{T_2}(v, b)$ is defined similarly. Let $D = MAX_{<a,b> \in X_e}(dist_{T_1}(a, u) + dist_{T_2}(v, b))$. By definition of the sets X_i, D is no more than $(C_{OPT}/k)h(k) \cdot 2^{i'}$. Let c represent the quantity $(\frac{k}{\log k})^{\frac{1}{3}}$. We now divide the trees T_1 and T_2 into at most c segments, each of depth C_{OPT}/c. Node a belongs to segment $i \geq 1$ of T_1 if $dist_{T_1}(a, u) \in \left[i \cdot \frac{C_{OPT}}{c}, (i+1) \cdot \frac{C_{OPT}}{c}\right]$. Similarly, node b belongs to segment $j \geq 1$ of T_2 if $dist_{T_2}(v, b) \in \left[j \cdot \frac{C_{OPT}}{c}, (j+1) \cdot \frac{C_{OPT}}{c}\right]$. If a node is within a distance C_{OPT}/c of the root, it is still said to belong to segment 1.

Definition 9.4 *We define k_{ij} $(1 \leq i \leq c, 1 \leq j \leq c)$ to be the number of node pairs $< a, b > \in X_e$ such that a belongs to segment i of T_1 and b belongs to segment j of T_2. Also, let $n_i^{(1)}$ $(i \geq 2)$ be the number of nodes a' in a segment $s' \geq i-1$ of T_1 which satisfy the following properties:*

1. *There exists no vertex $a \neq a'$ in segment $i-1$ of T_1 such that a lies on the path from a' to u.*

2. *There exists a vertex a in segment $s \geq i$ of T_1 such that a' lies on the path from a to u.*

Define $n_1^{(1)}$ to be 1. The quantities $n_i^{(2)}$ are similarly defined on the tree T_2. Let n_{ij} be the product of $n_i^{(1)}$ and $n_j^{(2)}$.

Informally, $n_i^{(1)}$ represents the number of branches of the tree T_1 that cross the entire $(i-1)$-th segment of T_1.

Lemma 9.5 *There exist $1 \leq i \leq c$ and $1 \leq j \leq c$ such that $k_{ij}/n_{ij} \geq (\frac{h(k)}{\log k} \cdot 2^{i'-1})/c^2$.*

Proof. Each node pair that belongs to X_e contributes to k_{ij} for some pair (i, j). Therefore $\sum_{i,j} k_{ij} \geq (h(k)/\log k) \cdot 2^{i'-1}$. Now notice that $\sum_i n_i^{(1)}$ (and also $\sum_i n_i^{(2)}$) is bounded by c, since the number of branches that cross an entire segment of depth C_{OPT}/c can be no more than c (remember that the cost of T_1 can be no more than C_{OPT}). Now, $\sum_{i,j} n_{ij} \leq (\sum_i n_i^{(1)}) \cdot (\sum_j n_j^{(2)}) \leq c^2$. Therefore, we are guaranteed that there exists a pair $(i, j), 1 \leq i, j \leq c$ which satisfies the required condition. □

Using Lemma 9.5, we can claim the existence of nodes $a' \in T_1$ and $b' \in T_2$ and a set $X' \subseteq X_e$, $|X'| \geq (\frac{h(k)}{\log k} \cdot 2^{i'-1})/c^2$ such that the following properties hold for all $< a, b > \in X_e$:

1. a' lies on the path from a to u in T_1 and b' lies on the path from v to b in T_2.

2. $dist_{T_1}(a, a') \leq 2C_{OPT}/c$ and $dist_{T_2}(b', b) \leq 2C_{OPT}/c$.

Let the bunch B' be defined as $< a', b', X' >$. Let $k' = k(B')$ be the number of node pairs satisfied by this bunch $(k' \geq (\frac{h(k)}{\log k} \cdot 2^{i'-1})/c^2)$. Also, let $c(B')$ be the cost of this bunch. The shortest path from any first component of X' to a' is at most $2C_{OPT}/c$, by construction. The shortest path from a' to b' is at most D. The shortest path from b' to any second component of X' is also at most $2C_{OPT}/c$. Therefore, $c(B') \leq D + 4k' \cdot C_{OPT}/c$. Recall that $D \leq (C_{OPT}/k)h(k) \cdot 2^{i'}$, $c = (\frac{k}{\log k})^{\frac{1}{3}}$, and $h(k) = 6k^{\frac{2}{3}}(\log k)^{\frac{1}{3}}$. Substituting these values, we obtain $d(B') \leq \frac{C_{OPT}}{k} \cdot h(k)$. □

Theorem 9.6 *$\mathcal{C}(k, X)$ can be used to obtain a polynomial time algorithm which gives an approximation ratio of $O(k^{\frac{2}{3}} \log^{\frac{1}{3}} k)$ for the directed generalized Steiner network problem.*

Proof. From Theorem 9.3 and Lemma 9.2 it follows that $\mathcal{C}(k, X)$ produces an $h(k)$ partial approximation to DG-Steiner (k, x). We can now invoke Lemma 7.4 to obtain the desired result. □

Remark:

Since k could be $\Theta(n^2)$, the approximation ratio we guarantee is $\omega(n)$. We can use our result on directed Steiner trees to obtain a $O(\min\{k^{2/3} \log^{\frac{1}{3}} k, n^{1+\epsilon}\})$ algorithm for the generalized version. Obtaining an $O(n)$ approximation for all k seems non-trivial. The worst ratio we can prove for our algorithm is $\Omega(\sqrt{k})$ but the upper bound we can guarantee is only $O(k^{\frac{2}{3}} \log^{\frac{1}{3}} k)$. We would like to see the gap bridged. There is also the general case where each pair specifies a connectivity requirement r_{ij} which gives the number of edge disjoint paths required between i and j. No non-trivial upper bound is known for this problem.

Chapter 10

Approximating Arbitrary Metrics by a Small Number of Trees[1]

10.1 Introduction

The powerful result of Bartal [17] on probabilistic approximation of metric spaces has recently led to either the first known or improved approximation algorithms for several NP-Complete graph optimization problems. Some of these problems include the group Steiner tree problem [56], the k-median problem [80, 30, 32], the buy at bulk network design problem [8], and the minimum communication spanning tree problem [136]. All these problems have a similar structure: given a graph G with a cost metric c defined on the edges, find the minimum cost subset of edges satisfying certain constraints. Bartal's result can be used to approximate the metric space c by a tree such that, for any edge (u, v) in the graph, the *expected* distance between u and v in the tree is at most $O(\log^2 n)$ times its length in the graph. Thus many optimization problems where the objective is to minimize some function of the edge lengths (linear combinations are very common) can be solved now on the tree metric space losing only a poly-logarithmic factor. Due to the simple structure of trees, many NP-Complete problems can be solved exactly or approximated, in polynomial time on them. Buy at bulk network design [8] is an example of a problem where the objective function is non-linear, yet the paradigm can be applied to obtain poly-logarithmic

[1]This chapter represents joint work with M. Charikar, C. Chekuri, and S. Guha [30] and with M. Charikar, C. Chekuri, S. Guha, and S. Plotkin [31].

approximations. Recently, Bartal [18] has improved the expected stretch factor in his tree construction to $O(\log n \log \log n)$, thereby improving approximation algorithms that use his result. Since the first step in applying this paradigm consists of constructing a tree probabilistically, all the approximation algorithms which result are randomized. Our focus in this chapter is to *derandomize* these results.

We achieve this derandomization in two stages. First, we observe that since Bartal's result guarantees that *each* edge length stretches by no more than $O(\log n \log \log n)$ in the expected sense. By linearity of expectations, any linear combination of edge lengths stretches by no more than $O(\log n \log \log n)$. But in many applications of Bartal's result, all that is needed is a guarantee that *one* specific given linear combination of edge lengths does not stretch too much. This observation motivates the following general paradigm.

Preserving a fixed linear combination

Let $G = (V, E)$ be an *undirected* graph with a cost function c associated with the edges. Suppose we need to solve a problem which involves finding a minimum cost set of edges (the objective function is the sum of the costs of the chosen edges) satisfying certain connectivity constraints. Many applications in network design and facility location fall in this category of problems. Some examples are Steiner trees, group Steiner trees, capacitated and uncapacitated facility location, and k-median. Since we are interested in connectivity, we can work with the metric induced by the shortest path distances in G for all these problems. One of the standard techniques for solving such a problem is to round a linear programming relaxation of the problem. Let Π be a graph optimization problem which can be expressed as an integer program with 0-1 variables $\bar{x}(e)$ for each edge and with an objective function expressible as $\sum_{e \in E} \bar{x}(e) \cdot c(e)$. We use the following series of steps to solve Π.

1. Solve the LP relaxation of the problem and let $x(e)$ be the fractional values assigned to the edges. Then $\sum_{e \in E} x(e) \cdot c(e)$ is a lower bound on the optimal.

2. Deterministically construct T such that $\sum_{(u,v) \in E} x(u,v) \cdot d_T(u,v) \leq \alpha \sum_{(u,v) \in E} x(u,v) \cdot c(u,v)$ is satisfied, where $d_T(u,v)$ is the distance between u and v in T. The LP solution carries over to the tree T.

3. Round the LP solution on the tree to an integral solution losing at most a factor of β.

4. Translate solution on T to G obtaining an $\alpha\beta$ approximation.

Observe that we preserve only a *specific* linear combination of edge lengths. This is what lets us create one tree in a deterministic fashion. In Section 10.2 we show how we can do step 2 of above with $\alpha = O(\log^2 n)$ using ideas from the paper of Garg et al. [57]. We can improve this to $O(\log n \log\log n)$ using an idea first developed by Seymour [124] and subsequently applied by Even et al. in their spreading metric paper [45], Klein et al. [87], and others. This was obtained by us after hearing about Bartal's improved result for probabilistic approximation of metric spaces [18]. We later discovered that the above mentioned tree construction has also been obtained independently by Bartal [18]. Step 3 is problem specific. We now describe two problems to which this paradigm can be applied – the group Steiner tree problem and the minimum communication spanning tree problem – and obtain deterministic rounding procedures for both of them.

The group Steiner tree problem is the following. Let G be an undirected graph with edge weights. Let $S = \{t_1, t_2, \ldots, t_k\}$ be a set of k subsets of V. The objective is to find a minimum weight tree which has at least one vertex from each of the groups. This problem is hard to approximate within logarithmic factors even on stars (via reduction from set cover [46]). The problem was introduced by Reich and Widmayer [121] and finds applications in VLSI design. Some related problems are discussed in [65]. Bateman et al. [20] gave the first non-trivial approximation algorithm with a ratio of $(1 + \ln k/2)\sqrt{k}$. Charikar et al. [29] using a reduction to directed Steiner trees gave an algorithm with a ratio of k^ϵ for any fixed $\epsilon > 0$. They also showed that it is possible to obtain an $O(\log^2 k)$ approximation in quasi-polynomial time. Garg et al. [56] gave a randomized approximation algorithm with a ratio of $O(\log^3 n \log k)$ using probabilistic approximation of metric spaces. They use a novel variant of randomized rounding to solve the problem on trees. We obtain a *deterministic* rounding scheme on trees. Plugging this rounding scheme into our paradigm, we obtain a deterministic algorithm with a ratio of $O(\log^2 n \log k \log\log n)$. We believe our deterministic rounding on the tree is of independent interest.

Finally we point out that the problem of minimum communication spanning trees can be derandomized directly by our tree construction procedure.

Approximating arbitrary metrics by polynomially many trees

A more general approach towards derandomizing Bartal's result would be to obtain a probability distribution on *polynomially* many trees such that the expected stretch of any edge is small. The following theorem guarantees the existence of such a distribution.

Theorem 10.1 *Let $G = (V, E)$ be an undirected graph with a weight function c on the edges that satisfies triangle inequality. For any such graph, there exists a $z^* = O(\log n \log\log n)$, $(m+1)$ trees (where m is the number of edges) $T_1, T_2, \ldots, T_{m+1}$, and positive real weights $\alpha_1, \alpha_2, \ldots, \alpha_{m+1}$ such that $\sum_{i=1}^{m+1} \alpha_i = 1$ and*

$$\sum_{i=1}^{m+1} \alpha_i \cdot d_{T_i}(u,v) \leq z^* \cdot c(u,v) \quad \textit{for each } (u,v) \in E(G)$$

where $d_{T_i}(u,v)$ is the distance between u and v in T_i.

Notice that Theorem 10.1 is existential; a proof of this theorem is presented in Section 10.3. Our main result is to demonstrate that such a distribution can indeed be constructed efficiently. More specifically, we present a polynomial time algorithm that given a weighted graph on n vertices constructs a probability distribution on $O(n \log n)$ trees such that the expected stretch of any edge of the graph in a tree chosen from the distribution is $O(\log n \log\log n)$. This is in contrast to Bartal's algorithms [17, 18] that generate trees from exponentially large distributions. Using our result we can derandomize applications to approximation algorithms by solving the problem at hand on each of the trees from the distribution, and picking the best of the solutions. This result builds upon the framework described above where we obtained deterministic approximation algorithms by preserving a fixed linear combination up to an $O(\log n \log\log n)$ factor; that framework applies to a restricted class of applications. Using our main result we obtain the first *deterministic* approximation algorithms for buy-at-bulk network design [8] and some variants of vehicle routing [33] that match the randomized approximation ratios. Further, our result establishes that future applications can assume a small distribution and hence be derandomized. Our derandomization is practical for two reasons: the small size of the distribution, and an efficient algorithm to construct the distribution. The small size of the distribution also makes it feasible to preprocess a graph and store the trees for repeated use.

Our main result is a consequence of viewing probabilistic approximations in a novel way. Let M be a finite metric space and $\mathcal{S}$ a collection of finite metric spaces that dominate M. We phrase the problem of finding the best (in the sense of least distortion) probabilistic approximation of M by $\mathcal{S}$ as a *fractional packing* problem (with $|\mathcal{S}|$ variables.) We then use the algorithm of Plotkin, Shmoys, and Tardos (PST) [116] for approximately solving this fractional packing problem. The PST framework requires a certain dual subroutine; when

$\mathcal{S}$ is the space of tree metrics the dual subroutine turns out to be the minimum communication cost spanning tree problem which is NP-hard [54, 136]. However an approximate dual subroutine can be obtained using graph partitioning techniques from [57, 124, 45], as shown in [18, 30] which give an $O(\log n \log \log n)$ guarantee on the approximation ratio. Since the best upper bound we have on the optimal value of the fractional packing problem is $O(\log n \log \log n)$, a naive application of the approximate dual in PST gives only an $O((\log n \log \log n)^2)$-probabilistic approximation. We modify the PST framework to exploit the fact that the dual subroutine gives a bound on the *integrality gap* as well as an approximation ratio of $O(\log n \log \log n)$; this results in an $O(\log n \log \log n)$ probabilistic approximation of a finite metric by a distribution on $O(n \log n)$ trees. We believe that our modification to PST (Theorem 10.20) is of independent interest. Using improved graph partitioning results, we show that planar graphs can be $O(\log n)$-probabilistically approximated (matching the result of [92] but using a small distribution), and points in d-dimensional Euclidean spaces can be $O(\sqrt{d} \log n)$-probabilistically approximated using a distribution on $O(n \log n)$ and $O(dn \log n)$ trees, respectively. For Euclidean spaces our result improves on the earlier known result of $O(d \log n)$ [111]. The improvement is obtained by a new graph partitioning procedure for Euclidean spaces that we describe below. One of our contributions, apart from obtaining a polynomial size distribution, are our new proofs for probabilistic approximation by tree metric spaces. Further, probabilistic approximation by any class of metric spaces $\mathcal{S}$ is reduced to an optimization problem, thus providing a concrete way to approach classes of metric spaces other than trees.

Low Diameter Partitioning of Real Normed Spaces

Clustering of metric spaces has applications in data analysis, classification, learning, facility location, and several other areas. Clustering objectives vary considerably depending on the application. One particular objective that has found several applications is that of low diameter clustering. The objective is to partition a metric space (induced usually by a weighted undirected graph) into subgraphs such that the diameter of each of the subgraphs is low. Peleg et al. and others [112, 7] studied such partitions motivated by applications in network routing and distributed computing. Leighton and Rao [97] pioneered the use of low diameter clustering for the design of approximation algorithms based on a divide and conquer approach. See [57, 86, 45] for related results. Bartal [17] used probabilistic partitioning of graphs into low diameter clusters to obtain tree approximations.

In this chapter we consider low diameter clustering when the graphs are induced by points in real normed spaces. Given a metric space G with a distance function c, let μ be a probability distribution over the partitions of G into clusters of diameter at most D. Let x_{uv} be the probability of vertices u and v belonging to different clusters in a partition picked according to μ. Define β to be the quantity $\max_{u \neq v}\{x_{uv} \cdot D/c_{uv}\}$. The probability of vertices u and v being "cut" is at most $\beta c_{uv}/D$. If β is small then vertices which are close together in G belong to the same cluster with high probability. Our objective is to give a probabilistic partitioning algorithm which guarantees a small value of β.

We obtain an improved graph partitioning algorithm when the graph is induced by points in $\Re_2^d$. For any $D > 0$ we show that a graph embedded in $\Re_2^d$ can be clustered into regions of diameter at most D, such that β is at most $2\sqrt{d}$. Notice that β is independent of n. Earlier partitioning algorithms achieved $\beta = O(d)$ for Euclidean graphs [111], and $O(\log n)$ for general metrics [17]. Since the dimension d can be reduced to $O(\log n)$ with only a small increase in distances [77], our upper bound gives at least an $\Omega(\min(\sqrt{d}, \sqrt{\log n}))$ improvement over the previous best results for graphs induced by $\Re_2^d$. We generalize our results to graphs induced by points in $\Re_p^d$ ($\Re^d$ equipped with the l_p norm). We prove a lower bound of $\Omega(d)$ on β for graphs induced by $\Re_1^d$. This lower bound and our upper bound for $\Re_2^d$ can be combined to obtain matching upper and lower bounds (upto constant factors) of $d^{1/p}$ for $1 \leq p \leq 2$; for $p > 2$, we have an upper bound of $O(d^{1-1/p})$. Let $f(d,p) = d^{1/p}$ for $1 \leq p \leq 2$ and $f(d,p) = d^{1-1/p}$ for $p > 2$. We use our partitioning algorithm to obtain a $O(f(d,p) \cdot \log n)$-probabilistic approximation for n point metric spaces induced by points in $\Re_p^d$.

The rest of the chapter is organized as follows. Section 10.2 describes the deterministic construction which preserves a given linear combination of edges to within an $O(\log n \log\log n)$ factor. Section 10.2.1 describes the rounding procedures for the group Steiner tree problem; Section 10.2.2 points out that the Minimum Communication Spanning Tree problem is exactly the same as that of embedding a metric into a graph while preserving a fixed linear combination of edge lengths. Section 10.3 details our main result. We describe how the problem of constructing a probability distribution on trees while preserving the edge lengths up to some factor α can be viewed as a fractional packing procedure which uses the Minimum Communication Spanning Tree problem as a dual procedure. In Section 10.3.1, we show how to approximately solve the fractional packing problem in polynomial time using the framework of [116]. We describe our partitioning results in Section 10.4.

10.2 Preserving a fixed linear combination of edge lengths

Definition 10.2 (Bartal [17]) *A k-hierarchically well-separated tree (k-HST) is defined as a rooted weighted tree with the following properties*

1. *The edge weight from any node to each of its children is the same.*
2. *The edge weights along any path from the root to a leaf are decreasing by a factor of at least k.*

Let $G = (V, E)$ be a undirected graph with two positive weight functions c and x defined on the edge set E. In the sequel, distance in the graph is defined with respect to the weight function c. We will assume that for any edge $(u, v) \in E$, that $c(u, v) = d_G(u, v)$. We will further assume that $x(e) > 0$. Let $\Delta(G)$ denote the diameter of G. The following theorem is a modified version of the ball growing theorem in the paper of Garg, Vazirani, and Yannakakis on multicuts [57].

Theorem 10.3 (Garg et al. [57]) *Given a graph G with two positive weight functions x and c on the edges, and a value $k > 1$, it is possible to partition G in two induced subgraphs G_1 and G_2, in polynomial time, such that*

1. $\Delta(G_1) \leq \Delta(G)/k$.
2. *Let* $E(G_1, G_2) = \{(u, v) | u \in G_1, v \in G_2\}$,

$$\sum_{e \in E(G_1, G_2)} x(e) \leq \frac{O(k \log n)}{\Delta(G)} \sum_{e \in E(G_1) \cup E(G_1, G_2)} x(e) \cdot c(e).$$

Define $\mathsf{vol}(G) = \sum_{e \in E(G)} x(e) \cdot c(e)$. The corollary below is a direct consequence of Theorem 10.3.

Corollary 10.4 *For any $k > 1$, we can partition $V(G)$ in to $V_1, V_2, \ldots, V_p$ such that*

1. *For $1 \leq i \leq p$, $\Delta(G_i) \leq \Delta(G)/k$ where G_i is the induced subgraph on V_i.*
2. $\displaystyle\sum_{e \in F} x(e) \leq \frac{O(k \log n)}{\Delta(G)} \mathsf{vol}(G)$,
   where $F = \{(u, v) \in E | u \in V_i \Rightarrow v \notin V_i\}$.

We use the above theorem to construct a k-HST for G which preserves the weighted sum of the edge lengths to within a $O(\log^2 n)$ factor.

Theorem 10.5 *For any $k > 1$ we can construct in polynomial time a k-HST, $T = (V', E_T)$ where every vertex of G occurs as a leaf of T and the following is true.*

- *For a node i in T let T_i be a subtree rooted at i and let G_i be the subgraph of G induced by the leaves of T_i. For every i, the following holds*
  1. *$d_{T_i}(u,v) \geq c(u,v)$ where $d_{T_i}(u,v)$ is the distance between u and v in T_i.*
  2. *The edges from i to its children have a length of $\Delta(G_i)/2$.*
  3. *The distance from i to any leaf is at most $\frac{k}{2(k-1)}\Delta(G_i)$.*
- $$\sum_{(u,v)\in E} x(u,v)\cdot d_T(u,v) \leq O\left(\frac{k^2}{k-1}\log_k n \log n\right) \sum_{(u,v)\in E} x(u,v)\cdot c(u,v).$$

Proof. We construct the tree as follows. We first contract all edges e such that $c(e) \leq \frac{\Delta(G)}{2kn}$. Let G' be the resulting graph. Using Corollary 10.4, we partition G' into $G'_1, \ldots, G'_p$ with a parameter $2k$ such that $\Delta(G'_i) \leq \Delta(G)/2k$. For $1 \leq i \leq p$ let G_i be the graph obtained by expanding the contracted edges of G'_i. It is easy to see that $\Delta(G_i) \leq \Delta(G'_i) + n \cdot \frac{\Delta(G)}{2kn} \leq \Delta(G)/2k + \Delta(G)/2k \leq \Delta(G)/k$. We recursively build the trees for each G_i. Let $T_1, \ldots, T_p$ be the respective trees. We create a tree T for G by adding a new root r and connecting it to the roots of $T_1, \ldots, T_p$ by edges of length $\Delta(G)/2$. Inductively the length of the longest path from the root to any leaf in each of T_i is at most $\frac{k}{2(k-1)}\Delta(G_i)$. Therefore the length of the longest path from the root in T is bounded by

$$\begin{aligned}
\frac{1}{2}\Delta(G) + \frac{k\Delta(G_i)}{2(k-1)} &\leq \frac{1}{2}\Delta(G) + \frac{\Delta(G)}{2(k-1)} \\
&\leq \frac{k}{2(k-1)}\Delta(G)
\end{aligned}$$

It is also easy to see that the distance in T between any two leaves u and v such that $u \in T_i$ and $v \in T_j$, $i \neq j$, is at least $\Delta(G)$ which is at least their distance in G. Since the diameter of the graphs falls by a factor of k at each level, the construction also ensures that the resulting tree is a k-HST. The only property left to verify is the bound on the sum of the distances in the tree T. Let $\mathsf{cost}(T) = \sum_{(u,v)\in E} x(u,v) \cdot d_T(u,v)$. Notice $V(G) \subseteq V(T)$. Let $F = \{(u,v) \in E | u \in V_i \Rightarrow v \notin V_i\}$. Then it is clear that

$$\begin{aligned} \mathsf{cost}(T) &= \sum_{i=1}^{p} \mathsf{cost}(T_i) + \sum_{(u,v)\in F} x(u,v) \cdot d_T(u,v) \\ &\leq \sum_{i=1}^{p} \mathsf{cost}(T_i) + \frac{k \cdot \Delta(G)}{k-1} \sum_{(u,v)\in F} x(u,v) \\ &\leq \sum_{i=1}^{p} \mathsf{cost}(T_i) + \frac{k}{k-1} O(k \log n) \mathsf{vol}(G'). \end{aligned}$$

Observe that we have G' instead of G in the above equation. From the above equation it follows that $\mathsf{cost}(T)$ is bounded by $O(\frac{k^2}{k-1} \log n)$ times the sum of the volumes of all graphs used for partitioning. Instead of summing up over the volumes of graphs, we can sum up over all edges e, the quantity $q(e) \cdot x(e) \cdot c(e)$ where $q(e)$ is the number of levels an edge participates in before it is cut. We claim that $q(e) = O(\log_k n)$ for all e. This immediately gives the desired bound on the cost of the tree T. To prove the claim observe that an edge stays contracted as long as the current diameter of the graph being partitioned is greater than $2kn \cdot c(e)$. Since the diameter of the graphs decreases geometrically by a factor of k, it follows that an edges contributes to only $O(\log_k n)$ levels. □

Remark 10.6 *The tree constructed in Theorem 10.5 can be modified to obtain a different tree T' whose vertex set is identical to that of G such that $d_{T'}(u,v) \geq c(u,v)$ for all $(u,v) \in E(G)$, and $\sum_{(u,v)\in E(G)} x(u,v) \cdot d_{T'}(u,v) \leq O(\log^2 n) \sum_{(u,v)\in E} x(u,v) \cdot c(u,v)$.*

After hearing about Bartal's [18] improvement to probabilistic approximations, we obtained the following theorem for preserving a fixed linear combination of edges of a graph to within an $O(\log n \log \log n)$ factor on a tree. We later discovered that this theorem has also been obtained independently by Bartal [18]. The tree construction is very similar to the one presented above and the details of the construction can be found in [18]. The idea for improving the analysis to obtain an improved ratio was introduced by Seymour [124] and has been subsequently applied by Even et al. [45] for spreading metrics, Klein et al. [87], and others. For completeness, we present a proof of this theorem in Appendix B.1.

Theorem 10.7 *A k-HST satisfying properties* 1 *to* 3 *of Theorem 10.5 can be constructed* *in polynomial time such that*

$$\sum_{u,v \in V(G)} x(u,v) \cdot d_T(u,v) = O(\frac{k^2}{k-1} \log n \log \log n) \sum_{u,v \in V(G)} x(u,v) \cdot c(u,v),$$

and further, for each pair of vertices (u,v)*,* $d_T(u,v) \leq \frac{2k^2 n}{k-1} c(u,v)$*.*

For planar graphs Konjevod et al. [92] give an $O(\log n)$ probabilistic approximation by tree metrics, as well as a procedure to approximate a fixed linear combination of edges within an $O(\log n)$ factor on a tree. Their result is based on a stronger partitioning procedure than that of [57] for graphs with shallow excluded minors presented by Klein et al. [86]. On Euclidean graphs in d dimensions, we show (Section 10.4) that the average stretch can be bounded by $O(\sqrt{d} \log n)$. Thus our approximation for the group Steiner tree problem improves by a factor of $\log \log n$ on planar graphs.

Lower Bounds: Consider the metric of the $n \times n$ grid, denoted by $G(n,2)$.

Definition 10.8 *A tree* T *is said to be a* super-spanning *tree of the grid* $G = G(n,2)$*, if* **(1)** *All vertices of* G *are leaves of* T*, and* **(2)** $d_T(x,y) \geq d_G(x,y)$ *for all* $x, y \in V(G)$*.*

Konjevod, Ravi and Salman [92] extended the results of Alon, Karp, Peleg and West [4] to prove the following theorem, which gives a lower bound of $\Omega(\log n)$ for the deterministic tree construction in Theorem 10.7, even for planar graphs.

Theorem 10.9 ([4, 92]) *If* T *is a super-spanning tree of* $G = G(n,2)$*, and an edge is chosen uniformly at random from E(G), then* $E[d_T(e)] \geq \Omega(\log n)$*.*

Consider the linear combination of edge lengths of $G(n,2)$ where each edge is given a weight of 1. The above theorem implies that every super-spanning tree of $G(n,2)$ must distort the value of this linear combination by a factor of $\Omega(\log n)$.

We now describe three applications of the framework developed above.

10.2.1 The group Steiner tree problem

The undirected group Steiner tree problem $G = (V, E, c, r, S)$ consists of a vertex set V, an edge set E, a cost metric c defined on E, and a set S of groups, where each group is a subset of V. Also, r is a distinguished vertex (the root) in G. A solution to the problem is a subtree T of G such that $T \cap t \neq \emptyset$ for all $t \in S$ and T contains r. The cost of the solution is $\sum_{e \in T} c(e)$ and the objective is to find a minimum cost solution. In this section we give a *deterministic* $O(\log^2 n \log k \log\log n)$ approximation algorithm for this problem, where $k = |S|$. Without loss of generality we can assume that G is a complete graph, and that c satisfies the triangle inequality.

A randomized $O(\log^3 n \log k)$ algorithm for this problem was given by Garg et al. [56]. They use Bartal's result [17] to obtain a probabilistic approximation of the metric space c by a tree metric T_G. Then they solve a LP relaxation of the group Steiner tree problem on the tree T_G. Finally, they give a randomized rounding procedure that rounds the solution of the LP relaxation to an integer solution that is within a factor of $O(\log n \log k)$ of the optimal LP value. We solve the same LP relaxation as [56], but we solve it on the original graph. We then invoke Theorem 10.7 to translate this LP to the tree (Section 10.2.1). Finally, we use the method of conditional probabilities (Section 10.2.1) to give a deterministic rounding procedure which produces a solution that is within a factor of $O(\log n \log k)$ of the LP relaxation.

Bartal's [18] recent improvement can be plugged into the result of Garg et al. [56] to obtain the same approximation ratio as we obtain here. However our result is deterministic, and we believe that our rounding technique is of independent interest.

Obtaining a tree LP for group Steiner trees

The group Steiner tree Problem can be modeled as the following 0-1 integer program. It will help to think of f as flows and x as capacities. The idea behind the formulation is that there is a flow of value 1 from the root to each group (f_t denotes flow for group t), in the graph with edge capacities defined by x. The flows are directed even though the graph is not. We relax the integer program to obtain the following LP.

$$\min \textstyle\sum_{(u,v)\in E} c(u,v)\cdot x(u,v)$$

$$\begin{array}{rcll}
\displaystyle\sum_{u\neq v} f_t(u,v) - \sum_{v\neq w} f_t(v,w) & = & 0 & \\
 & & t\in S, v\in V-\{r,t\} & \\
f_t(t,v) & = & 0 & t\in S, v\in V \\
\displaystyle\sum_{u\in t}\sum_{v\in V} f_t(v,u) & = & 1 & t\in S \\
f_t(u,v)+f_t(v,u) & \leq & x(u,v) & u,v\in V \\
x(u,v) & = & x(v,u) & \\
f_t(u,v) & \geq & 0 &
\end{array}$$

Using Theorem 10.7, we obtain a spanning tree T of G such that

$$\begin{array}{rcl}
\displaystyle\sum_{(u,v)\in E(G)} d_T(u,v)x(u,v) & \leq & O(\log n\log\log n)\cdot \\
 & & \displaystyle\sum_{(u,v)\in E(G)} c(u,v)x(u,v)
\end{array}$$

where $d_T(u,v)$ is the length of the path $P_T(u,v)$ between vertices u and v in the tree T. For each group $t\in S$ and any edge $e\in T$, define $f_t^T(e)=\sum_{e\in P_T(u,v)} f_t(u,v)$. Short circuit the flows f_t^T by eliminating cycles – this can only decrease $f_t^T(e)$ on any edge $e\in T$. Define $x_T(e)=\max_{t\in S} f_t^T(e)$. Since T is a spanning tree of G the flows f_t^T can also be looked upon as flows in G.

Lemma 10.10 *The flows f_t^T define a feasible solution to the LP for group Steiner trees. Further,* $\displaystyle\sum_{e\in T} x_T(e)c(e)\leq O(\log n\log\log n)\sum_{(u,v)\in E} c(u,v)x(u,v)$.

We define $f_t^\tau=\sum_{u:u\in t\cap\tau}\sum_{v:(v,u)\in T} f_t^T(v,u)$. Informally, f_t^τ is the amount of flow on tree τ for group t. Due to technical reasons we will need the following two properties of the LP: that for all $e\in T$, $t\in S$, the flow $f_t^T(e)$ is a multiple of $1/n$, and that $f_t^T=1$ for all $t\in S$. The first property is ensured by doubling all flows and then rounding them down to multiples of $1/n$. This doubles the cost of the LP solution by at most a factor 2. The

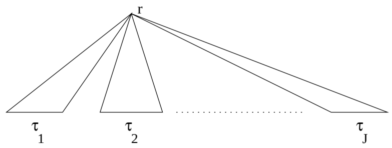

Figure 10.1: The trees $\tau_1 \dots \tau_J$ maintained by the algorithm.

second property can be obtained by reducing flows if necessary. Let z^* represent the cost of the above LP.

Rounding the LP on the tree

Our rounding scheme is essentially a derandomization of the rounding scheme give by Garg et al. [56]. We use the method of conditional probabilities in conjunction with a pessimistic estimator [118].

Let X be any subtree of T, such that $r \in X$. Define the *density* of the tree X rooted at r to be its cost divided by the number of groups satisfied by X. We will maintain a pessimistic estimator $\tilde{D}$ for the density that we hope to obtain. We will look at edges one by one, and either pick the edge or discard it, making sure that $\tilde{D}$ does not increase. After examining all the edges, we obtain a tree X with density no worse than the pessimistic estimator that we had at the very beginning. We show that our pessimistic estimator, and therefore the density of X, is at most $z^* \log(2n)/k$. Of course not all groups might get satisfied by X, so this procedure will be repeated till all groups get satisfied. Repeating the procedure will cost us an additional factor of $\log k$ in the approximation ratio. We will now focus on the construction of X.

The algorithm only examines edges that are currently incident on the root r. If an edge e gets chosen, it gets contracted into the root and all flows in the subtree attached to this edge get multiplied by $1/y_e$ where y_e is the *current* capacity of the edge (since the flows on edges change, so do the capacities. We use x and f for the initial values, and y and g for the current ones.). The algorithm maintains a set of disjoint trees $\tau_1 \dots \tau_J$ hanging from the root (Figure10.1). Splitting a tree into two is allowed, but merging two trees is not. Initially there is just one tree T hanging from r. Let f_t^τ and g_t^τ represent the initial and current flows, respectively, on tree τ for group t. To obtain an estimator $\tilde{D}$ for density, we define estimators for the cost and the profit.

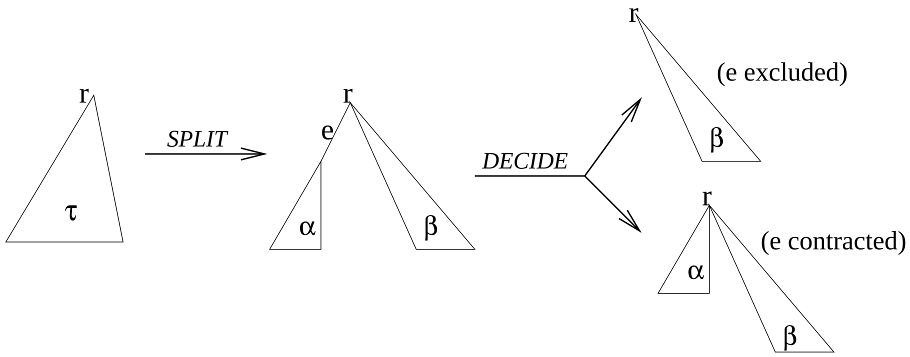

Figure 10.2: One iteration of ROUND.

Costs: Define $\tilde{C}^{\tau} = \sum_{e \in \tau} y(e)c(e)$, and let C^r be the cost of all edges that have been contracted into the root. $\tilde{C} = C^r + \sum_{i=1}^{J} \tilde{C}^{\tau}$ be the estimator for cost.
Profits: Define $\tilde{P}_t^{\tau} = g_t^{\tau} / \log_2(2nf_t^{\tau})$. Lemma 10.11 guarantees that $0 \leq g_t^{\tau} \leq 1$, and therefore $\tilde{P}_t^{\tau}$ can be looked upon as probabilities (recall that all flows are multiples of $1/n$). We are now going to pretend that $\tilde{P}_t^{\tau}$ indeed represents the probability of group t being satisfied in tree τ. Let $\mathbf{Prob}(\{a_1 \dots a_J\})$ be the probability of at least one of J independent events happening where a_i is the probability of the i-th event, *ie.* $\mathbf{Prob}(\{a_1 \dots a_J\}) = 1 - \prod_{i=1}^{J}(1 - a_i)$. Define $\tilde{P}_t = \mathbf{Prob}(\{\tilde{P}_t^{\tau_i} : 1 \leq i \leq J\})$. Let $\tilde{P} = \sum_{t \in S} \tilde{P}_t$ be the estimator for profit.

The intuition behind the above definition of profit is provided by the analysis of Garg et al. of their randomized rounding procedure [56]. Having defined estimators for cost and profit, we define $\tilde{D} = \tilde{C}/\tilde{P}$. The algorithm ROUND repeatedly picks any subtree τ out of $\{\tau_1 \dots \tau_J\}$, and picks any edge e in τ incident on r. It then does the following two steps (See Figure 10.2):

Step 1 –SPLIT: Partition the tree τ into α which is the tree hanging from r via edge e, and β, the rest of the tree. The quantity $\tilde{C}$ remains unchanged but the quantity $\tilde{P}$ needs to be recomputed, since the event corresponding to P_t^{τ} has now been split into two independent events.

Step 2 –DECIDE: Calculate $\tilde{C}^X$ and $\tilde{P}^X$, the estimators for cost and profit if e (and hence the entire subtree α) were to be excluded. Also calculate the estimators $\tilde{C}^I$ and $\tilde{P}^I$ if the edge e were to be chosen. Choosing e involves contracting it into the root r and multiplying the flows (and hence the capacities) on all the other edges in α by $1/y(e)$.

Let $\tilde{D}^X = \tilde{C}^X/\tilde{P}^X$ and $\tilde{D}^I = \tilde{C}^I/\tilde{P}^I$. Pick the smaller of $\tilde{D}^X$ and $\tilde{D}^I$ and perform the corresponding action. In case of a tie use $\tilde{D}^I$.

During each iteration at least one edge gets either contracted or removed. So the above algorithm terminates after n iterations. The cost and profit estimators can be computed in time $O(nk)$. After all the edges have been considered, the set of all contracted edges forms the tree X.

Lemma 10.11 $0 \le g_t^\tau \le 1$ *for all trees* τ *attached at the root and all groups* t.

Lemma 10.12 (a) $\tilde{D}$ *does not increase during the* SPLIT *step.* (b) $\tilde{D}$ *does not increase during the* DECIDE *step.*

Proof. *Part (a)* $\tilde{C}$ does not change during this step. To show that $\tilde{P}$ does not decrease, it suffices to show that

$$\tilde{P}_t^\tau \le \tilde{P}_t^\alpha + \tilde{P}_t^\beta - \tilde{P}_t^\alpha \cdot \tilde{P}_t^\beta. \tag{10.1}$$

because $\tilde{P}_t$ is monotonically increasing in each $\tilde{P}_t^\tau$. Let a be the larger of f_t^α and f_t^β and b be the smaller. Since the trees α and β were part of a single tree τ before the split, the flows on α and β have been increased by the same factor, say λ. Therefore $g_t^\alpha = \lambda f_t^\alpha$ and $g_t^\beta = \lambda f_t^\beta$. Our proof is by contradiction: assume that equation 10.1 gets violated. If $a = 0$ then the contradiction is quite straight forward. So assume $a > 0$. By Lemma 10.11, $a\lambda$ and $b\lambda$ are at most 1.

$$\begin{aligned}
\frac{\lambda(a+b)}{\log(2n(a+b))} &> \frac{\lambda a}{\log(2na)} + \frac{\lambda b}{\log(2nb)}\left(1 - \frac{\lambda a}{\log(2na)}\right) \\
&\ge \frac{\lambda a}{\log(2na)} + \frac{\lambda b}{\log(2n(a+b))}\left(1 - \frac{\lambda a}{\log(2na)}\right) \\
\Rightarrow \frac{a}{\log(2n(a+b))} &> \frac{a}{\log(2na)} - \lambda \cdot \frac{a}{\log(2na)} \cdot \frac{b}{\log(2n(a+b))} \\
\Rightarrow \log\frac{a+b}{a} &< \lambda b,
\end{aligned}$$

But $a \le b$ which implies that $(a+b)/a \ge 2$, or $\log_2((a+b)/a) \ge 1$. Also $\lambda b \le 1$, which gives a contradiction.

Part (b) We define $A \triangleq \tilde{C}^\alpha + y_e c_e$, and

$$B \triangleq \sum_t \left(1 - \mathbf{Prob}(\{\tilde{P}_t^{\tau_i} : \tau_i \ne \tau\} \cup \{P_t^\beta\})\right) \tilde{P}_t^\alpha.$$

Now

$$\begin{aligned}\tilde{D} &= (\tilde{C}^X + A)/(\tilde{P}^X + B),\\ \tilde{D}^I &= (\tilde{C}^X + A/y(e))/(\tilde{P}^X + B/y(e)).\end{aligned}$$

If $A/B > \tilde{C}^X/\tilde{P}^X$ then $\tilde{D} > \tilde{D}^X$. On the other hand, if $A/B \leq \tilde{C}^X/\tilde{P}^X$ then since $y(e) \leq 1$, $\tilde{D} \geq \tilde{D}^I$. Since the algorithm chooses the smaller of $\tilde{D}^X$ and $\tilde{D}^I$, the estimator for density does not increase. □

Lemma 10.13 *Let D be the density of the tree X, and $\tilde{D}$ the density estimator at the end of algorithm* Round. *Then $D \leq \tilde{D}$.*

Proof. Recall that C^r is the cost of all edges that have been contracted into the root. At the end of the algorithm, all edges have been either contracted into root or discarded. Therefore $\tilde{C} = C^R$. But C^R is just the cost of the tree X.

The profit of X is the number of groups that get satisfied in X. For any satisfied group t, the quantity $\tilde{P}^t$ is at most 1 (Lemma 10.11). All groups that are not chosen have zero flows in the final solution, and hence zero profit estimators. Therefore the profit of X is at least $\tilde{P}$. This completes the proof of the lemma. □

Lemma 10.14 *The initial estimator for the density is $z^* \log(2n)/k$.*

Theorem 10.15 *The cost of the final solution is at most $\log(2n)\log(2k) \cdot z^*$.*

Proof. The algorithm Round satisfies $i \geq 1$ groups and pays a cost of no more than $\dfrac{i \cdot \log(2n)}{k} \cdot z^*$. We repeat the algorithm Round till all groups get satisfied. After each invocation of Round we remove the flows for each satisfied group. Let there be N iterations of Round, and let k_j be the number of unsatisfied groups at the end of the j-th iteration. Now, $k = k_0 > k_1 > \ldots > k_N \geq 1$. The total cost is at most $\log(2n)z^* \sum_{j=1}^{N}(k_{j-1}-k_j)/k_{j-1}$. But this expression is at most $\log(2n)z^* \sum_{j=1}^{k} 1/j < \log(2n)\log(2k)z^*$. □

Theorem 10.16 *Combining Theorems 10.7 and 10.15 gives an $O(\log^2 n \log k \log\log n)$* deterministic *approximation algorithm for the group Steiner tree problem.*

10.2.2 The minimum communication cost spanning tree problem

The communication cost spanning tree problem is a minimization problem defined on an undirected weighted graph $G = (V, E)$. With each pair $(u, v) \in V \times V$ there is an associated positive weight w_{uv}. The objective is to find a spanning tree T that minimizes $\sum_{u,v} w_{uv} \cdot d_T(u, v)$ where $d_T(u, v)$ is the distance in the tree between u and v. We restrict ourselves to the case when the graph G is induced by an n point metric space (a *complete* graph on n vertices with the edge weights satisfying triangle inequality).

This problem has applications in routing with delays, and computational biology. For details see [136] where it is observed that an $O(\log^2 n)$ ratio is achievable for the metric case using probabilistic approximation of metric spaces (the ratio now improves to $O(\log n \log\log n)$ [18]). We can formulate the problem of finding a spanning tree as an integer program and relax it to obtain an LP with fractional values $x(e)$ on the edges. Since the graph is a metric, we observe that an alternative lower bound to the integer optimal is given by the trivial solution of setting $x(e) = 1$ for all edges (u, v). It is easy to see that the objective function is a linear combination of edge lengths and the tree we construct according to Theorem 10.7 is within a $O(\log n \log\log n)$ factor of the lower bound. From Remark 10.6 we obtain a tree whose vertex set is the same as that of G. We then use the fact that G is a metric to obtain a spanning tree.

Wu et al. [136] show that the general graph case can be reduced to the metric case if $w_{u,v} = 1$ for all pairs (u, v).

10.3 Probabilistic Approximation via Fractional Packings

In this section we show how the problem of probabilistic approximation by a class of metric spaces $\mathcal{S}$ can be formulated as a linear programming problem. We consider only the case when $\mathcal{S}$ is the class of tree metrics but the formulation and solution strategy generalize in a straight forward manner to other spaces as well. Let G be a weighted undirected graph on n vertices that induces a finite metric. Let $\mathcal{S} = \{T_1, T_2, \ldots, T_N\}$ be a set of tree metrics on the vertices of G. An important point to remember is that the trees T_i could have virtual vertices in addition to the vertices of G. We assume without loss of generality that $\mathcal{S}$ is finite although N could be an arbitrarily large function of n. We assume that each of the metrics in $\mathcal{S}$ *dominates* G, that is $d_{T_i}(u, v) \geq d_G(u, v)$ for every pair of vertices (u, v). Let $c(e)$ denote the length of an edge e in G. We assume that $c(u, v)$ is less than or equal to the distance between u and v in G. A probability distribution on $\mathcal{S}$ can be represented by assigning to each $T_i \in \mathcal{S}$ a real value $0 \le x_i \le 1$, with the constraint that the x_i sum to one. With this simple observation we obtain the following linear program to find the optimal probabilistic approximation of G by $\mathcal{S}$.

$$\begin{aligned}
\min \quad & \lambda \\
\sum_i d_{T_i}(e) \cdot x_i &\le \lambda \cdot c(e) \qquad \text{for every edge } e \\
\sum_i x_i &= 1 \\
x_i &\ge 0
\end{aligned}$$

In other words , we have to solve the problem $\min\{\lambda : Dx \le \lambda c, \sum_i x_i = 1\}$ where $D_{e,i} = d_{T_i}(e)$. Let D_e refer to the e-th row of D. The LP above has a large number of variables, but only $(m+1)$ constraints where m is the number of edges. We now show that there is an optimal solution to the above LP that has at most $(m+1)$ non-zero x_i. Thus, the existence of a small distribution is guaranteed from the formulation itself.

Theorem 10.17 (Same as Theorem 10.1) *Let $G = (V, E)$ be an undirected graph with a weight function c on the edges that satisfies triangle inequality. For any such graph, there exists a $z^* = O(\log n \log\log n)$, $(m + 1)$ trees (where m is the number of edges) $T_1, T_2, \ldots, T_{m+1}$, and positive real weights $\alpha_1, \alpha_2, \ldots, \alpha_{m+1}$ such that $\sum_{i=1}^{m+1} \alpha_i = 1$ and*

$$\sum_{i=1}^{m+1} \alpha_i \cdot d_{T_i}(u, v) \le z^* \cdot c(u, v) \quad \textit{for each } (u, v) \in E(G)$$

where $d_{T_i}(u, v)$ is the distance between u and v in T_i.

Proof. Let us look at the space of all spanning trees of G. We will express as an LP the optimization of the parameter z. This will have exponential number of variables, one corresponding to each spanning tree of G. Let α_i denote the weight of the ith spanning tree T_i.

$$\begin{aligned}
\text{Minimize} \quad & z \\
\textstyle\sum_i \alpha_i \cdot d_{T_i}(u, v) &\le z \cdot w(u, v) \\
\alpha_i &\ge 0
\end{aligned}$$

Notice that the above LP has $m+1$ constraints apart from the non-negativity constraints. There exists an optimal solution to the above LP which corresponds to a vertex of the polyhedron defined by the constraints. At that vertex solution at most $m+1$ of the α_i's will be non-zero since at least n constraints are tight. Let us consider those trees with non-zero α_i. There are polynomial number of them and they satisfy the required constraint. The fact that $z^* = O(\log n \log\log n)$ follows from the result of Bartal [18] where he gives a probability distribution on the spanning trees satisfying the inequality with $z = O(\log n \log\log n)$. This fact can also be deduced from the $O(\log n \log\log n)$ approximation for the minimum communication cost spanning tree problem that we give in Section 10.2.2, since this problem forms a separation oracle for the dual of the above LP. □

In addition, the formulation can be looked upon as a fractional packing problem, and we will exploit this structure. Let $\rho_e = \max_i d_{T_i}(e)/c(e)$ be the maximum stretch of edge e in any of the trees, and let $\rho = \max_e \rho_e$ be the maximum stretch of any edge. It follows that $D_e x \le \rho c_e$ for all x satisfying $\sum_i x_i = 1$. The parameter ρ is important in solving the LP and we will show that we can restrict ourselves to a space of trees with ρ polynomial in n.

The key to solving the above LP is to view it as a packing problem. A packing linear program is formulated as $\min\{\lambda : Ax \le \lambda b, x \in P\}$, where P is a convex polytope, and $Ax \ge 0$ for $x \in P$. Plotkin, Shmoys, and Tardos [116] showed that a packing problem can be solved to ϵ optimality (that is find a λ such that $\lambda \le (1+\epsilon)\lambda^*$) given the following subroutine.

- A DUAL procedure that given a vector $y \ge 0$ (y is an $m \times 1$ column vector if A is an $m \times n$ matrix) finds an $\tilde{x} \in P$, such that $c\tilde{x} = \min_{x \in P} cx$ where $c = y^t A$.

A crucial parameter for their algorithm is ρ, the *width* of the polytope P with respect to the constraint $Ax \le \lambda b$. The width is defined as $\max_i \max_{x \in P} \frac{A_i x}{b_i}$. The following theorem states the main result of [116].

Theorem 10.18 (pst:pack95) *There is an algorithm that solves a packing problem to ϵ optimality with $O(\epsilon^{-2}\rho\log(\rho/\epsilon))$ calls to the* DUAL *procedure.*

The running time of the above algorithm depends only on the DUAL procedure and the width. In particular there is no explicit dependence on the number of variables. Our goal in the rest of the section is to show how we can adapt the algorithm of [116] to solve our LP formulation. It turns out that the DUAL procedure as required by the PST algorithm is NP-hard for our program. We will show however that we can approximately solve the dual problem and also modify the proof of [116] to show that an approximate solution is sufficient.

10.3.1 Solving the Fractional Packing Problem

As we saw, our LP formulation is easily viewed as a packing problem. In particular the polytope P is defined by the constraint $\sum_i x_i = 1$ and $x_i \geq 0$. There are two important consequences of the simplicity of P.

1. The DUAL procedure is required to return a $\tilde{x} \in P$ that minimizes $y^t Ax$ over P. Since any point in our polytope is a convex combination of trees, it follows that there exists a *single* tree that achieves the minimum. This implies that a bound on the number of calls to the DUAL procedure gives a bound on the number of trees generated in the approximate solution.

2. The space of trees $\mathcal{S}$ and the width ρ are implicitly defined by the DUAL procedure. This follows from the fact that the only trees considered by the PST algorithm are those returned by the DUAL procedure.

We now interpret the DUAL procedure for our formulation. The vector y of dual variables corresponds to assigning weights $y(e) \geq 0$ for each edge e. Let $C^*(y) = \min_{T \in \mathcal{S}} \sum_e y(e) \cdot d_T(e)$. Thus the DUAL procedure is to find a tree that minimizes the average y-weighted edge lengths of G. This is exactly the Minimum Communication Spanning Tree problem discussed in Section 10.2.2. The following theorem is a restatement of Theorem 10.7 with $k = 2$. The stronger results for the planar case follow from the work of Klein et al. and Konjevod et al. [86, 92]; the stronger results for Euclidean graphs follows from the results in Section 10.4 of this chapter.

Theorem 10.19 *Let G be a weighted graph on n vertices that induces a metric space. Given positive weights $y(e)$ on the edges, there is a polynomial time algorithm that finds a tree T that dominates G such that $\sum_e y(e) \cdot d_T(e) \leq O(\log n \log \log n) \sum_e y(e) \cdot c(e)$. Further, for every edge e, $d_T(e)/c(e) = O(n)$. For planar graphs and for graphs induced by points in d dimensional Euclidean space the $O(\log n \log \log n)$ term improves to $O(\log n)$ and $O(\sqrt{d} \log n)$ respectively, with $d_T(e)/c(e) = O(n)$ for planar graphs, and $d_T(e)/c(e) = O(dn)$ for Euclidean graphs.*

Notice that the above theorem guarantees that the width of the polytope is $O(n)$ for general metrics as well as planar graphs, and $O(dn)$ for Euclidean graphs.

As mentioned before, we need a modified version of the PST algorithm [116] that gives a guarantee when the Dual procedure is approximate. Since every tree $T \in \mathcal{S}$ dominates G, $\sum_e y(e) \cdot c(e)$ is a lower bound on $C^*(y)$ and thus trivially $C(y) \leq O(\log n \log\log n)C^*(y)$. The upper bound we have on λ^*, the optimal of our formulation, is $O(\log n \log\log n)$. Theorem 10.19 viewed as providing an approximation algorithm for $C^*(y)$ with a ratio of $O(\log n \log\log n)$ guarantees only a $O((\log n \log\log n)^2)$-probabilistic approximation. We will exploit the stronger guarantee of Theorem 10.19 to obtain a $O(\log n \log\log n)$-probabilistic approximation. We restrict ourselves to packing problems in the formalism of Plotkin et al. [116] that satisfy the additional condition that $Ax \geq b$ for $x \in P$. This guarantees that $C^*(y) \geq y^t b$ for all $y \geq 0$. We obtain the following theorem by a careful restatement of the algorithm and proof in [116]; the details of these modifications are presented in Appendix B.2. We believe that the theorem is of independent interest and might find future applications.

Theorem 10.20 *Consider a packing problem with $Ax \geq b$ for $x \in P$. If the* Dual *procedure guarantees $C(y) \leq \beta y^t b$, the algorithm in [116] finds a $\tilde{x} \in P$ such that $A\tilde{x} \leq (1+\epsilon)\beta b$ in $O(\epsilon^{-2}\rho\log(\rho/\epsilon))$ calls to the* Dual *procedure. If the* Dual *procedure has the property that $C(y) \leq \beta C^*(y)$, $\tilde{x}$ satisfies $A\tilde{x} \leq (1+\epsilon)\beta\lambda^* b$.*

With Theorems 10.19 and 10.20 in place we put the pieces together to state and prove our main result.

Theorem 10.21 *Every finite metric on n points can be $O(\log n \log\log n)$-probabilistically approximated by a probability distribution on $O(n \log n)$ trees. The trees and the distribution can be computed in polynomial time.*

Proof. We apply Theorem 10.20 to our packing formulation of the problem. Our packing problem satisfies the condition of Theorem 10.20 since every tree in $\mathcal{S}$ dominates G. From Theorem 10.19, we have a dual procedure that guarantees that $C(y) = O(\log n \log\log n)y^t c$. Therefore we obtain an $\tilde{x}$ that satisfies $D\tilde{x} \leq O(\log n \log\log n) \cdot c$. Since the width guaranteed by the Dual procedure of Theorem 10.19 is $O(n)$, it follows that PST algorithm will terminate in $O(\epsilon^{-2} n \log(n/\epsilon))$ calls to Dual . Each call to the Dual procedure results in one tree, therefore $\tilde{x}$ has at most $O(\epsilon^{-2} n \log(n/\epsilon))$ non-zero entries. Choosing some small but fixed ϵ we obtain the desired result. $\square$

We obtain the following corollary using stronger results for special cases.

Corollary 10.22 *Planar graphs can be $O(\log n)$-probabilistically approximated and points in $\Re_2^d$ can be $O(\sqrt{d} \log n)$-probabilistically approximated by a distribution on $O(n \log n)$ and $O(dn \log n)$ trees respectively.*

We also obtain results for points in $\Re_p^d$ where the distance is measured using the l_p norm instead of the standard Euclidean norm. We describe the generalization in Section 10.4.

We end this subsection with some comments.

- There is a lower bound of $O(\log n)$ on the expected distortion provided by tree metrics [92] even for planar graphs. However a specific metric space might have a better approximation. This is captured by the optimal solution to our linear programming formulation. The second part of Theorem 10.20 shows that an improved approximation to the minimum communication cost spanning tree can result in relating the distortion we guarantee to the best possible for the given metric space.

- The tree metrics constructed by Bartal [17, 18] called HSTs satisfy a hierarchical separation property. An HST is a rooted tree such that the edge lengths along each root to leaf path decrease by a constant factor. This additional property is useful in certain applications (see [17] for details). We note that the trees guaranteed by Theorem 10.19 are also HSTs.

- It is possible to use the parallel algorithm for positive linear programming of Luby and Nisan [99] in place of [116] at the cost increasing the number of trees generated by a poly-logarithmic factor. However we do not have a parallel tree construction procedure to take advantage of the algorithm of [99].

10.4 Low Diameter Partitioning of Real Normed Spaces

Let G be a graph induced by n points in $\Re_p^d$ i.e, the vertices correspond to points in d-dimensional space, and the distances are induced by the l_p norm (the l_p distance between two d dimensional vectors u and v, denoted as $\|u-v\|_p$, is defined to be $(\sum_{i=1}^{d} |u_i - v_i|^p)^{1/p}$).

Equivalently, the graph G is said to be embedded in $\Re_p^d$. Let c_{uv} represent the distance between the vertices u and v.

We solve the following problem: given a graph $G(V)$ embedded in $\Re_p^d$, construct a probability distribution over partitions of G into disjoint clusters of diameter $\leq D$, such that the quantity $\beta(G) = \max_{u \neq v}\{(D/c_{uv}) \cdot x_{uv}\}$ is small, where x_{uv} is the probability of vertices u and v belonging to different clusters. Define $\beta_p(d)$ to be the maximum value of $\beta(G)$ over all graphs embedded in $\Re_p^d$. We will omit the subscript p and the argument d from $\beta_p(d)$ where these values are clear from the context. An algorithm that solves the above problem can be used to construct an $O(\beta \log n)$-probabilistic approximation of G using tree metrics [17].

For any graph G there is a trivial lower bound of $\Omega(1)$ on β. For arbitrary graphs Bartal [17] gave an algorithm which guarantees $\beta = O(\log n)$, and also showed the existence of graphs for which $\Omega(\log n)$ is a lower bound. For planar graphs, an algorithm for obtaining $\beta = O(1)$ is implicit in [86] and was used by [92] to obtain an $O(\log n)$-probabilistic approximation of planar graphs by trees.

Less is known about partitioning graphs that are embeddable in geometric spaces. We give tight upper and lower bounds for the value of β as a function of d, for all $1 \leq p \leq 2$. The bound is $\Theta(d^{1/p})$ if $1 \leq p \leq 2$; we obtain an upper bound of $O(d^{1-1/p})$ if $p \geq 2$. The upper bound follows from the partitioning algorithm for graphs embedded in $\Re_2^d$, presented in Section 10.4.1. The lower bound follows by interpolating our lower bound for $\Re_1^d$ (Section 10.4.2). Our upper bound is a significant improvement over the bound of $O(\log n)$ for general graphs. Specifically, if $p = 2$ then the dimension of the space can be reduced to $O(\log n)$ without distorting edge lengths by more than a constant factor[77]. Thus for Euclidean graphs, it is possible to beat the lower bound for general graphs by at least a factor of $\Omega(\sqrt{\log n})$. Peleg et al. [111] have recently given a partitioning procedure that results in $\beta = O(d)$ for Euclidean graphs.

10.4.1 Upper Bound: Partitioning Using Spheres

We restrict ourselves to graphs embedded in $\Re_2^d$. The partitioning algorithm is the following: draw spheres of radius $R = D/2$ around each vertex. Repeatedly pick points uniformly at random from the region defined by the union of all these spheres. Each time a point is picked, look at all the vertices within a distance R from this point. Put all these vertices into a cluster of their own, delete these vertices from the graph, and proceed. The algorithm

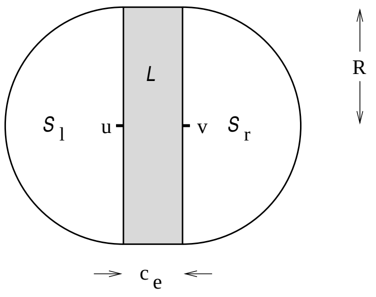

Figure 10.3: The shaded region, L, is an upper bound on the volume of $S(u) - S(v)$.

terminates after at most n iterations.

Theorem 10.23 *The above algorithm guarantees $\beta \leq 2\sqrt{d}$ for graphs embedded in $\Re_2^d$.*

Proof. Consider any two vertices u and v in G. Let $S(x)$ denote the Euclidean sphere of radius R centered at x. Let $V_k(r)$ be the volume of a sphere of radius r in k dimensions. Further, let $V_k(r) = C_k \cdot r^k$.

The two points u and v go into different clusters if a point in $(S(u)-S(v)) \cup (S(v)-S(u))$ gets picked before a point in $S(u) \cap S(v)$. But

$$\mathsf{vol}\,(S(u) - S(v)) = \mathsf{vol}\,(S(u) \cup S(v)) - \mathsf{vol}\,(S(v))\,.$$

Draw two hyperplanes orthogonal to the line (u, v) and passing through the points u and v respectively (Figure 10.3). Let S_l denote the hemisphere to the left of u and S_r the hemisphere to the right of v. Further, let L denote the cylindrical region bounded by the base of the hemispheres. The area of the base of L is $V_{d-1}(R)$ and the height of L is c_{uv}; therefore the volume of L is $c_{uv}.V_{d-1}(R)$. The region $S_l \cup S_r \cup L$ contains the region $S(u) \cup S(v)$. Therefore

$$\begin{aligned} &\mathsf{vol}\,(S(u) \cup S(v)) - \mathsf{vol}\,(S(v)) \\ &\leq \mathsf{vol}(L) + \mathsf{vol}(S_l) + \mathsf{vol}(S_r) - \mathsf{vol}\,(S(v)) \\ &\leq c_{uv} \cdot V_{d-1}(R). \end{aligned}$$

$$\text{i.e.}\quad x_{uv} \leq 2c_{uv}V_{d-1}(R)/V_d(R) = (2c_{uv}/R) \cdot (C_{d-1}/C_d).$$

The diameter of each cluster is at most $2R$. By definition of β, we now have $\beta \leq 4C_{d-1}/C_d$. But $C_{d-1}/C_d = \dfrac{\Gamma(1+d/2)}{2\Gamma(1/2+d/2)\Gamma(3/2)}$ [115]. It is easy to verify that this expression is at most $\sqrt{d}/2$, which implies that $\beta \leq 2\sqrt{d}$. □

The same algorithm can also be used to cluster spaces with any l_p norm where $p \geq 1$. Therefore the partitioning algorithm does not need to know the underlying norm associated with the space – it just needs to know the diameter D of the spheres that are used for partitioning.

We will use the following elementary fact repeatedly.

Fact 10.24 *Let x be a vector in $\Re^d$, and let $p > q$. Then $\|x\|_p \leq \|x\|_q \leq d^{1/q-1/p}\|x\|_p$.*

Theorem 10.25 *For $1 \leq p \leq 2$, $\beta \leq 2d^{1/p}$, and for $p \geq 2$, $\beta \leq 2d^{1-1/p}$.*

Proof. Consider the points u and v. Let $c_p = \|u - v\|_p$. Further, let D_p be the l_p diameter of the convex region defined by a Euclidean ball of l_2 diameter D. Let x be the probability of u and v falling into different clusters. Assume that the vertices u and v are the ones that maximize the ratio x/c_p. Then $\beta = xD_p/c_p = (xD_2/c_2) \cdot (D_p/D_2) \cdot (c_2/c_p)$. From Theorem 10.23, $xD_2/c_2 \leq 2\sqrt{d}$. Using Fact 10.24, if $1 \leq p < 2$ then $c_2/c_p \leq 1$ and $D_p/D_2 \leq d^{1/p-1/2}$. Therefore $\beta \leq 2d^{1/p}$. And if $p > 2$ then $c_2/c_p \leq d^{1/2-1/p}$ while $D_p/D_2 \leq 1$, which implies that $\beta \leq 2d^{1-1/p}$. □

10.4.2 Lower Bound on β

We show that the upper bound on β in Theorem 10.25 is tight up to constant factors for all $1 \leq p \leq 2$ if the underlying graph is the d-dimensional mesh of side n.

Lemma 10.26 *Consider the d-dimensional mesh of side r and with distances defined by the l_1 norm. Then $\beta = \Omega(d)$ for any probabilistic partitioning of this mesh into clusters of l_1 diameter $\delta, d < \delta < d \cdot r/24$.*

Proof. Two vertices in the mesh are said to be *adjacent* if they agree on $d-1$ coordinates and differ by one in the remaining coordinate; an edge is said to be *basic* if it is between two adjacent vertices. Consider any probabilistic partitioning of the mesh with clusters of l_1 diameter at most δ. Let S be the cluster with the largest number of points. Let $|S| = k^d$

(k need not be integral). Let A be the number of basic edges that are incident on exactly one vertex in S. Alon et al. [4] prove the following result:

$$A \geq dk^{d-1}(1 - k/r).$$

The number of basic edges within the cluster is at most $2dk^d$. Thus the probability of a basic edge being cut in this partitioning is $\Omega((1/k)(1 - k/r))$. If $\delta > d$ then the number of points in a cluster of l_1 diameter δ is at most $(12\delta/d)^d$ [115]. This implies $k \leq 12\delta/d$. Since $\delta < d \cdot r/24$, we have $1 - k/r > 1/2$. Thus the probability of a basic edge being cut is $\Omega(d/\delta)$. Since each basic edge has length 1, this gives a lower bound of $\Omega(d)$ on β for graphs embedded in $\Re_1^d$. □

We combine Lemma 10.26 and Fact 10.24 to obtain the following theorem.

Theorem 10.27 *Consider the d-dimensional mesh of side n and with distances defined by the l_p norm. Then $\beta = \Omega(d^{1/p})$ if $1 \leq p \leq 2$ for any probabilistic partitioning of this mesh into clusters of l_p diameter $\delta > d$.*

Thus we have matching upper and lower bounds on β for all l_p norms, $1 \leq p \leq 2$. Interestingly, for all norms, the upper bound is achieved by the same algorithm (independent of p) and the lower bound (for $1 \leq p \leq 2$) is achieved on the same graph.

10.4.3 Approximating Graphs in $\Re_p^d$ with Tree Metrics

In this section we show how to solve the minimum communication cost spanning tree problem where the underlying graph is embedded in $\Re_2^d$ – the construction for $\Re_p^d$ follows from the construction for $\Re_2^d$ using Fact 10.24. We also show that the width of the tree constructed is at most $d \cdot n$. Using this as the dual procedure in the algorithm of Section 10.3 gives us the result claimed in Corollary 10.22.

Consider a graph G consisting of n points embedded in $\Re_2^d$, and let c_{uv} denote the l_2 distance between vertices u and v. Let y_{uv} be the weight assigned to the pair (u, v). Clearly $L = \sum_{u,v \in G} c_{uv} \cdot y_{uv}$ is a lower bound on the cost of the minimum communication cost spanning tree. Let the diameter of the graph be Δ. We cluster the graph with spheres of radius $R = \Delta/4$ using the algorithm in Section 10.4.1. Let C be the set of all pairs of vertices such that the two vertices belong to different clusters. Call a partitioning *good* if it satisfies the following two criteria:

1. $R \sum_{(u,v)\in C} y_{uv} \leq 4\sqrt{d} \cdot L$.

2. No two vertices i and j with $c_{ij} \leq R/(8dn)$ belong to different clusters.

Condition 1 ensures that the total cost of the tree is low; condition 2 guarantees that the width of the polytope is small.

Lemma 10.28 *A partitioning is good with probability at least* $1/2$.

Proof. Let E_1 be the expected value of $R \sum_{(u,v)\in C} y_{uv}$ for a partitioning. Also, let x_{uv} be the probability of vertices u and v belonging to C. From linearity of expectations,

$$E_1 = R \sum_{u\in G, v\in G} x_{uv} y_{uv}.$$

But $x_{uv} \leq 2\sqrt{d} c_{uv}/(2R)$ (Theorem 10.23). Therefore

$$E_1 \leq \sqrt{d} \sum_{u\in G, v\in G} c_{uv} y_{uv} = \sqrt{d} L.$$

Using Markov's inequality, the quantity $R \sum_{(u,v)\in C} y_{uv}$ is greater than $4\sqrt{d}L$ with probability at most $1/4$, and condition (1) above gets violated with probability at most $1/4$.

Draw a subgraph H with the same vertex set as G and with an edge between all vertices i and j such that $c_{ij} \leq R/(8dn)$. Let P be a maximal connected component of H with k vertices; embed P back into $\Re_2^d$ such that each vertex in P coincides with the corresponding vertex in G. There must exist some vertex v in P such that a ball around v of radius $kR/(16dn)$ will cover all the vertices in P. P is said to be cut if this ball gets partitioned during the partitioning. Draw two concentric balls of radius $R - kR/(16dn)$ and $R + kR/(16dn)$ with their center at v. P gets cut if and only if a sphere with a center in the shell defined by the two spheres gets chosen before a sphere with a center in the inner sphere. Let x_P be the probability of P being cut.

$$\begin{aligned} x_P &\leq \frac{(R + kR/(16dn))^d - (R - kR/(16dn))^d}{(R + kR/(16dn))^d} \\ &< (1 + \frac{k}{16dn})^d - (1 - \frac{k}{16dn})^d \end{aligned}$$

Since $(1+\alpha)^d - (1-\alpha)^d \leq (1+2\alpha)^d - 1$ for all $\alpha > 0$

$$\begin{aligned} x_P &\leq (1+\frac{k}{8dn})^d - 1 \\ &\leq e^{k/(8n)} - 1 \\ &\leq k/(4n) \qquad [e^\alpha - 1 \leq 2\alpha \text{ for all } \alpha \leq 1] \end{aligned}$$

Summing over all maximal connected components of H, the probability of one or more connected components being cut is at most 1/4. But if none of the connected components in H gets cut then no two vertices i and j with $c_{ij} \leq R/(8dn)$ belong to different clusters. Therefore, condition (2) gets violated with probability at most 1/4.

Therefore both conditions are simultaneously satisfied with probability at least 1/2. □

We can use the partitioning guaranteed by the above lemma in the tree construction algorithm described in Theorem 10.5 to obtain the following theorem (the proof follows from the proofs of tree constructions in [18, 30, 92]). Corollary 10.30 then follows using the general framework described in Section 10.3.

Theorem 10.29 *There is a randomized polynomial time algorithm for the minimum communication cost spanning tree problem on a graph embedded in $\Re_2^d$ that, with probability at least half, produces a tree whose cost is at most $O(\sqrt{d}\log n)$ times the lower bound L. In addition the distortion of any edge in the tree is $O(dn)$.*

Corollary 10.30 *A graph embedded in $\Re_p^d$ can be α-probabilistically approximated by a a distribution on $O(dn\log n)$ trees, where $\alpha = O(d^{1/p})$ if $1 \leq p \leq 2$ and $\alpha = O(d^{1-1/p})$ if $p \geq 2$. Such a distribution can be constructed in randomized polynomial time.*

Acknowledgements

I would like to thank Piotr Indyk for several helpful discussions.

Appendix A

The Necessity and Sufficiency of a Speedup of 2-1/N for Emulating FIFO

A.1 Necessity

With a speedup of two, the algorithms described in Chapter 2 (CCF and GBVOQ) exactly mimic an arbitrary size OQ switch. The next natural question to ask is whether it is possible to emulate output queueing using a CIOQ switch with a speedup less than 2. This section presents a lower bound of $2 - 1/N$ on the speedup of any CIOQ switch that emulates OQ switching, even when the OQ switch uses FIFO. Hence the algorithms that presented in this paper are almost optimal. In fact, the difference of $1/N$ can be ignored for all practical purposes.

Since a speedup between 1 and 2 represents a non-integral distribution of phases, it is not clear how scheduling phases are distributed. A speedup of $2 - 1/N$ corresponds to having a truncated time slot out of every N time slots; the truncated time slot has just one scheduling phase, whereas the other time slots have two scheduling phases each. For the purposes of our lower bound, assume that the scheduling algorithm does not know in advance whether a time slot is truncated.

Recall that a cell is represented as P-TL, where P represents which output port the cell is destined to, and TL represents the time to leave for the cell. For example, the cell C-7

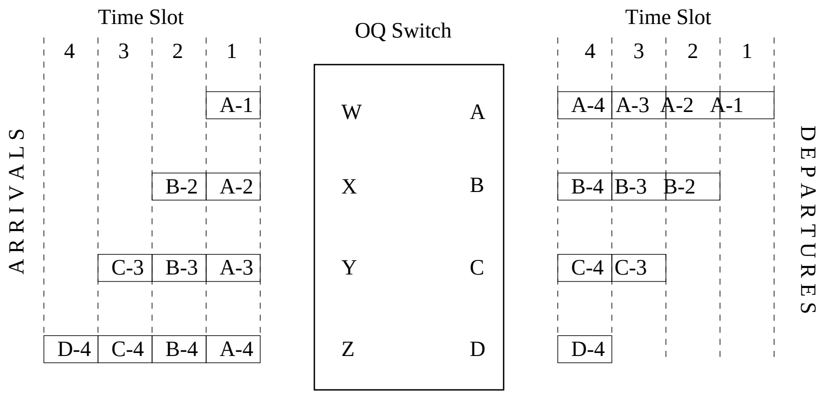

Figure A.1: Lower bound Input Traffic Pattern for a 4×4 switch.

must be scheduled for port C before the end of time slot 7.

The input traffic pattern that provides the lower bound for an $N \times N$ CIOQ switch is given below. The traffic pattern spans N time slots, the last of which is truncated.

1. In the first time slot, all input ports receive cells destined for the same output port, P_1.

2. In the second time slot, the input port that had the lowest time to leave in the previous time slot does not receive any more cells. In addition, the rest of the input ports receive cells destined for the same output port, P_2.

3. In the j-th time slot, the input ports that had the lowest time to leave in each of the previous time slots do not receive any more cells. In addition, the rest of the input ports must receive cells destined for the same output port, P_j.

The above traffic pattern can be repeated as many time as required to create arbitrarily long traffic patterns. Figure A.1 shows the above sequence of cells for a 4×4 switch. The departure events from the OQ switch are depicted on the right, and the arrival events are on the left. For simplicity, the proof of the lower bound is presented on this switch instead of a general switch.

Figure A.2 shows the only possible schedule for transferring these cells across in seven phases. Of the four time slots, the last one is truncated, giving a total of seven phases. Cell A-1 must leave the input side during the first phase, since the CIOQ switch does not know whether the first time slot is truncated. Similarly, cells B-2, C-3, and D-4 must leave

Phase	PA	PB	PC	PD
1	A-1			
2				
3		B-2		
4				
5			C-3	
6				
7				D-4

(a)

Phase	PA	PB	PC	PD
1	A-1			
2	A-2			
3		B-2		
4		B-3		
5			C-3	
6			C-4	
7				D-4

(b)

Phase	PA	PB	PC	PD
1	A-1			
2	A-2			
3	A-3	B-2		
4		B-3		
5		B-4	C-3	
6			C-4	
7				D-4

(c)

Phase	PA	PB	PC	PD
1	A-1			
2	A-2			
3	A-3	B-2		
4	A-4	B-3		
5		B-4	C-3	
6			C-4	
7				D-4

(d)

Figure A.2: Scheduling Order for the lower bound input traffic pattern in Figure A.1.

during the third, fifth, and seventh phases, respectively (see Figure A.2(a)). Cell A-2 must leave the input side by the end of the third phase. But it cannot leave during the first or the third phase because of contention. Therefore, it must depart during the second phase. Similarly, cells B-3 and C-4 must depart during the fourth and sixth phases, respectively (see Figure A.2(b)). Continuing this elimination process (Figures A.2(c), A.2(d)), there is only one possible scheduling order. For this input traffic pattern, the switch needs all seven phases in four time slots which corresponds to a minimum speedup of $7/4$ (or $2-1/4$).

Theorem A.1 *A minimum speedup of* $2-1/N$ *is necessary for an CIOQ switch operating under any algorithm which is not allowed to consider the number of scheduling phases in a time slot.*

The proof of the above theorem is a straight-forward extension of the CIOQ switch example.

A.2 Sufficiency

This section shows that it is possible to emulate a FIFO OQ switch using a speedup of $2-1/N$. Specifically, this emulation can be achieved by a CIOQ switch which follows the general framework described in Section 2.2, using a scheme called "Last In Highest Priority" (LIHP) to determine input priorities for incoming cells. As the name suggests, LIHP places a newly arriving cell right at the front of the input priority list. The analysis in this section borrows heavily from ideas described in Section 2.4.

A slightly different time slot structure is used in this section. A "normal" time slot has an arrival phase followed by two scheduling phases and then a departure phase, whereas a "truncated" time slot has an arrival phase, a scheduling phase, and then a departure phase. Since the speedup is $2-1/N$, we assume that there are at least $N-1$ normal phases between two truncated phases. The CIOQ switch does not need to know which phases are truncated.

At any time instant, and for any cell X, let $\eta(X)$ denote the number of truncated time slots between now and the time when this cell leaves the OQ switch, inclusive. Recall from Section 2 that $\mathrm{L}(X) = \mathrm{OC}(X) - \mathrm{IT}(X)$ is the slackness of cell X, where $\mathrm{OC}(X)$ and $\mathrm{IT}(X)$ refer to the output cushion and input thread of the cell, respectively.

The following lemma holds for CIOQ switches that use LIHP and operate at a speedup of $2 - 1/N$.

Lemma A.2 *If the OQ switch being emulated is FIFO, then $L(X) \geq \eta(X)$ after the first scheduling phase and just before the arrival phase, for all cells X waiting on the input side.*

Theorem A.3 is a consequence of Lemma A.2 – the proof of the lemma itself is deferred to the end of this section.

Theorem A.3 *A speedup of $2-1/N$ suffices for a CIOQ switch that uses LIHP to emulate a FIFO OQ switch.*

Proof. Suppose it is time for cell X to leave the OQ switch, and suppose that the CIOQ switch has successfully mimicked a FIFO OQ switch so far. Clearly, OC(X) must be zero. If X has already crossed over to the output side then it can leave the switch in time. So suppose X is still queued at its input port. If the current time slot were truncated then L(X) would be at least one (Lemma A.2). But then the input thread would be negative, which is not possible. Therefore, the current time slot has two scheduling phases. Invoking Lemma A.2 again, L(X) must be at least zero after the first scheduling phase. Since OC(X)is zero, the input thread of X must be zero too. Cell X, therefore, is at the front of both its input and its output priority lists, and will cross the switch in the second scheduling phase, just before the departure phase. This completes the proof of the theorem. □

Proof of Lemma A.2: Suppose the lemma has been true till the beginning of time slot $t-1$. The following argument will show that the lemma holds at the end of the first scheduling phase and at the end of the departure phase in time slot t, completing a proof by induction.

First consider the end of the first scheduling phase. Cells which were already present on the input side at the beginning of time satisfy L(X) $\geq \eta(X)$, as $\eta(X)$ does not change (a property of FIFO – the departure time of a cell from the OQ switch gets fixed upon arrival, and does not change), and L(X) can only go up (see Lemma 2.10 for an explanation of why L(X) can not decrease) during the arrival phase and the first scheduling phase together. Now consider a cell Y which arrives during time slot t. Let $k = \eta(Y)$. Since the slackness of a cell is at least zero upon arrival (remember that the input thread of an arriving cell is zero in LIHP), the slackness at the end of the first scheduling phase must be at least one. Therefore Y trivially satisfies the lemma if $k \leq 1$. Suppose $k > 1$. At most N cells could have arrived during the current time slot, and therefore, there must have been a cell Z in the system with $\eta(Z) = k - 1$ and the same output port as Y, at the beginning of time t (the fact that the truncated time slots are spaced at least apart is used here). If Z is waiting on the input side, then OC(Z) $\geq$L(Z) $\geq k - 1$. Since the OQ switch is FIFO, OC(Y) $\geq$OC(Z). But the input thread of the arriving cell Y must be zero. Hence, the slackness of is at least $k - 1$ after the arrival phase, and consequently, at least k after the first scheduling phase. The case where Z is waiting at the output side is similar.

Now concentrate on the end of time slot t. If this time slot turns out to be normal, then the slackness of any cell does not decrease during the second scheduling phase and the departure phase. Else, the slackness of any cell can go down by at most one. But the η value goes down by one for all cells in the system, and the lemma continues to hold.

Appendix B

Spreading Metrics and Packing

B.1 Trees from Spreading Metrics

B.1.1 Spreading Metrics

Even et al. [45] developed the idea of using spreading metrics as a divide-and-conquer paradigm for approximating NP-Complete graph optimization problems. They showed how to obtain improved results for several problems involving cuts in graphs like linear arrangement, embedding graphs in meshes, minimizing storage time product and so on. We will apply their spreading metric ideas in our tree construction. Below we describe their paradigm.

A problem instance, Π is denoted by a pair of graphs (G, H) where $G = (V, E)$ is a graph with edge capacities $x(e) \geq 1$ (In their exposition in [45] the capacities are denoted by $w(\cdot)$, we use $x(\cdot)$ as the same syntax as the rest of this chapter.) The graph $H = (V, E_H)$ called an *auxiliary graph*, is defined on the same vertex set. The paradigm of Even et al. is applicable for a problem if it satisfies the following two properties.

Problem Π can be solved by using a divide-and-conquer approach either by removing edges or vertices. We will use the edge version.

Divide-and-conquer applicability. Let $F \subseteq E$ be a set of edges whose removal partitions the graph in to p connected components $G_1, \ldots, G_p$, induced by vertex sets $V_1, \ldots, V_p$. Let $x(F)$ denote the sum of the capacities of the edges in F. The applicability of divide-and-conquer means that

- Π can be solved on G by solving the subproblems on $G_1, \ldots, G_p$.

- The cost of solving Π on G satisfies the following property.

$$\mathsf{cost}(G,H) \leq \begin{cases} 0 & \text{if } E_H \text{ is empty.} \\ \sum_{i=1}^{p} \mathsf{cost}(G_i,H_i) + \mathsf{scaler}(H)\cdot x(F) & \text{otherwise} \end{cases}$$

  The coefficient $\mathsf{scaler}(H)$ is a cost-scaler of the contribution of F to the solution and it is a function of H only. It is assumed that E_H is not empty implies $\mathsf{scaler}(H) \geq 1$.

Spreading metric computability. A spreading metric is a function $l : E \to Q$ that assigns rational lengths to every edge $e \in E$. It should be computable in polynomial time, and satisfies the following two properties:

- The volume of the spreading metric, defined by $\sum_{e\in E} x(e)l(e)$, is a lower bound on $\mathsf{cost}(G,H)$.

- Let $d(u,v)$ denote the length of the shortest path from u to v , where distances are measured with respect to edge lengths $l(e)$. Let $U \subseteq V$ be any subset of the vertex set. Denote by G_U and H_U the subgraphs of G and H respectively, induced by U. The diameter guarantee implies that for every subset $U \subseteq V$, if H_U has at least one edge, then there exist two vertices $u,v \in U$ that are connected in H_U, such that $d(u,v) \geq \mathsf{scaler}(H_U)$.

Theorem B.1 (Even et al. [45]) *For any optimization problem Π that satisfies the above paradigm, it is possible to find an approximation algorithm with approximation factor $O(\min\{\log\tau\log\log\tau, \log k\log\log k\})$, where τ is the value of an optimal (fractional) solution, and $k \leq n$ is the number of vertices that are not isolated in H.*

Remark B.2 *(Remark 4 in Theorem 1 in [45]) If the scaler function satisfies the weaker condition that there should exist vertices $u,v \in H_U$ such that $d(u,v) \geq \alpha\cdot\mathsf{scaler}(H_U)$, then the above theorem is true with an additional factor of $1/\alpha$ in the approximation ratio.*

B.1.2 Constructing Trees via Spreading Metrics

Let $G = (V,E)$ be a complete undirected graph with two positive weight functions c and x defined on the edge-set E where c satisfies the triangle inequality. We will further assume that $x(e) > 0$. By scaling we can assume without loss of generality that $x(e) \geq 1$. Distance is defined with respect to the weight function c.

Definition B.3 (see [17]) *A k-hierarchically well-separated tree (k-HST) is defined as a rooted weighted tree with the following properties*

- *The edge weight from any node to each of its children is the same.*
- *The edge weights along any path from the root to a leaf decrease by a factor of at least k.*

Let $u, v \in V$ be two non-isolated vertices for which $\mathsf{dist}(u, v) \geq \Delta/k$. $N(u, r)$ is the set of all vertices whose distance from u is at most r under the metric $c(\cdot)$. Define $N^{<}(u, r)$ to be the set of all vertices whose distance from u is strictly less than r. Denote by $E(u, r)$ the set of edges for which both endpoints belong to $N(u, r)$, and by $C(u, r)$ the set of all edges belonging to the cut $(N(u, r), V - N(u, r))$. Define $\mathsf{cap}(S, V - S)$, the *capacity* of the cut $(S, V - S)$ to be $\mathsf{cap}(S, V - S) \triangleq \sum_{e \in (S, V-S)} x(e)$. Define $\mathsf{vol}(e)$ the *volume* of edge e to be $\mathsf{vol}(e) \triangleq x(e)c(e)$. Define $\mathsf{vol}(u, r)$, the *volume* of $N(u, r)$ to be

$$\mathsf{vol}(u, r) \triangleq \sum_{e \in E(u,r)} \mathsf{vol}(e) + \sum_{e=(y,z) \in C(u,r)} x(e)(r - \mathsf{dist}(u, y))$$

Let vol^* denote the volume of the whole graph, that is $\mathsf{vol}^* = \mathsf{vol}(u, \infty)$. Without loss of generality we assume that $\mathsf{vol}(u, \Delta/2k) \leq \mathsf{vol}^*/2$; otherwise, swap u and v.

Even et al. [45] proved the following lemma, however they proved the lemma for the case $k = 1$.

Lemma B.4 *Let $G = (V, E)$ denote a graph with edge capacities $x(e) \geq 1$ and non-negative edge lengths $c(e)$. Let $\mathsf{dist}(u, v) \geq \Delta/k$ for $u, v \in V$. Suppose that $\mathsf{vol}(u, \Delta/2k) \leq \mathsf{vol}^*/2$. Then, there exists a radius $\Delta/4k \leq r \leq \Delta/2k$ such that,*

$$\mathsf{cap}(N^{<}(u, r), V - N^{<}(u, r)) \leq \frac{2k\mathsf{vol}(u, r)}{\Delta} \ln\left(\frac{e \cdot \mathsf{vol}^*}{2\mathsf{vol}(u, r)}\right) \ln\ln\left(\frac{e \cdot \mathsf{vol}^*}{2\mathsf{vol}(u, \Delta/4k)}\right),$$

where either $r = \mathsf{dist}(u, z)$ for some $z \in V$, or $r = \Delta/2k$.

The reader is referred to Theorem 2, in [45]; where the proof uses a quantity M, the proof of the lemma observed above follows from their proof, substituting $M' = Mk$ in place of M in their construction. We use Lemma B.4 (and modify the proof slightly) to obtain the following lemma:

Lemma B.5 *Let $G = (V, E)$ denote a graph with edge capacities $x(e) \geq 1$ and non-negative edge lengths $c(e)$. Let $\mathsf{dist}(u, v) \geq \Delta/k$ for $u, v \in V$. Suppose that $\mathsf{vol}(u, \Delta/2k) \leq \mathsf{vol}^*/2$. Then, there exists a set S, $u \in S$ such that the diameter of S is at most Δ/k, $\mathsf{vol}(S) \leq \mathsf{vol}^*/2$ and*

$$\mathsf{cap}(S, V - S) \leq \frac{4k\mathsf{vol}(S)}{\Delta} \ln\left(\frac{e \cdot \mathsf{vol}^*}{2\mathsf{vol}(S)}\right) \ln\ln\left(\frac{e \cdot \mathsf{vol}^*}{2\mathsf{vol}(u, \Delta/8k)}\right), \tag{B.1}$$

and no edge of length less than $\Delta/4kn$ belongs to the cut $(S, V - S)$. $\mathsf{vol}(S)$ denotes the volume taken out of G.

Proof. Contract all edges in G with length less than $\Delta/4kn$, i.e. change their length to 0. Let G' be the resulting graph. The contraction operation does not change the volume of an edge, i.e. all zero length edges retain their original volume. $N(u, r), N^{<}(u, r), E(u, r), C(u, r)$ are defined as before with distances measured in G'. Define $\mathsf{vol}'(u, r)$ to be

$$\mathsf{vol}'(u, r) = \sum_{e \in E(u,r)} \mathsf{vol}(e) + \sum_{e=(x,y) \in C(u,r)} x(e)(r - \mathsf{dist}_{G'}(u, x))$$

Note that $\mathsf{dist}_{G'}(u, v) \geq \Delta/2k$ and $\mathsf{vol}'(u, \Delta, 4k) \leq \mathsf{vol}^*/2$. By Lemma B.4, there exists a radius $\Delta/8k \leq r \leq \Delta/4k$ such that,

$$\mathsf{cap}(N^{<}(u, r), V - N^{<}(u, r)) \leq \frac{4k\mathsf{vol}'(u, r)}{\Delta} \ln\left(\frac{e \cdot \mathsf{vol}^*}{2\mathsf{vol}'(u, r)}\right) \ln\ln\left(\frac{e \cdot \mathsf{vol}^*}{2\mathsf{vol}'(u, \Delta/8k)}\right), \tag{B.2}$$

where either $r = \mathsf{dist}(u, x)$ for some $x \in V$, or $r = \Delta/4k$.

Let S be the set of vertices in $N^{<}(u, r)$ (including contracted vertices). Then $\mathsf{vol}(S) = \mathsf{vol}'(u, r)$, which is the volume of the subgraph cut out of G. This is the volume of the edges in the subgraph induced by the vertex set S and the edges which were cut under the distance metric $\mathsf{dist}_{G'}(\cdot)$. Notice that since the volume of the edges were not changed but only their lengths, the volume of as computed in G' is the same as in G. The only difference is that the set S does not represent a sphere (Not all vertices are within a distance r from u.) in the resulting cut.

The cut $(N^{<}(u, r), V - N^{<}(u, r))$ is the same as the cut $(S, V - S)$. In G', the distance from u to any vertex in S is at most $r \leq \Delta/4k$. Expanding the contracted edges could increase this distance by at most $n \cdot \Delta/4kn = \Delta/4k$. Thus every vertex in S is at a distance of at most $\Delta/2k$ from u. It follows that the diameter of S is at most Δ/k. Further

$S \subseteq N(u, \Delta/2k)$. Hence $\mathsf{vol}(S) \leq \mathsf{vol}(u, \Delta/2k) \leq \mathsf{vol}^*/2$. Observe that no edge of length less than $\Delta/4kn$ belongs to the cut $(S, V - S)$. Since distances in G are not less than distances in G', $\mathsf{vol}'(u, \Delta/8k) \geq \mathsf{vol}(u, \Delta/8k)$.

This proves the lemma. □

In order to find a set S that satisfies Lemma B.5, we first contract all small edges as in the proof of Lemma B.5. We then grow a region $N(u, r)$ using a single source shortest path algorithm until the capacity of the cut $(N^<(u, r), V - N^<(u, r))$ is bounded by the inequality (B.2). This gives us the set S. We use this to construct a tree using the following procedure:

Build-Tree(G, Δ)

If $G = \emptyset$ return singleton root.
If $\Delta = 0$ return singleton tree labeled $V(G)$.
Find S according to Lemma B.5.
Let G_1 be the graph induced on S and G_2 be the graph induced on $V(G) - S$.
$T_1 =$ Build-Tree$(G_1, \Delta(G_1))$.
$T_2 =$ Build-Tree(G_2, Δ).
Connect the root(T_1) to root(T_2) by an edge of length $\Delta/2$.
return T_2.

Let $T(G, \Delta)$ denote the tree returned by Build-Tree(G, Δ). By induction on the call graph of Build-Tree(G, Δ), we can claim the following lemma,

Lemma B.6 *By construction of the tree $T(G, \Delta)$,*

1. *The vertices of G are the leaves of $T(G, \Delta)$.*
2. *Every edge from* root$(T(G, \Delta))$ *to its children has length $\Delta/2$.*
3. *$T(G, \Delta)$ is a k-HST. (Since $\Delta(G_1) \leq \Delta/k$).*
4. *Any leaf of $T(G, \Delta)$ is within a distance of $k\Delta/2(k-1)$ of the root.*

Lemma B.7 *In the course of executing* Build-Tree$(G, \Delta(G))$, *any recursive call to* Build-Tree(G, Δ) *satisfies $\Delta \geq \Delta(G)$.*

Proof. By induction on the call graph of $\mathsf{Build\text{-}Tree}(G, \Delta(G))$. Clearly the property is true at the root. Consider the call to $\mathsf{Build\text{-}Tree}(G, \Delta)$. By the inductive hypothesis $\Delta \geq \Delta(G)$. The property holds for the call $\mathsf{Build\text{-}Tree}(G_1, \Delta(G_1))$. The property holds for the call $\mathsf{Build\text{-}Tree}(G_2, \Delta)$ as $\Delta \geq \Delta(G) \geq \Delta(G_2)$ (because $G_2 \subseteq G$). $\square$

Lemma B.8 *Let (u, v) be an edge in G.*

$$c(u,v) \leq \mathsf{dist}_{T(G,\Delta)}(u,v) \leq \frac{4k^2 n}{k-1} \cdot c(u,v) \tag{B.3}$$

Proof. By induction on the call graph of $\mathsf{Build\text{-}Tree}(G, \Delta)$. Consider the call to $\mathsf{Build\text{-}Tree}(G, \Delta)$. If $u, v \in G_1$ or $u, v \in G_2$, (B.3) follows by the inductive hypothesis. Suppose not. Then the edge (u, v) belongs to the cut $(S, V - S)$. Let $r = \mathsf{root}(T(G, \Delta))$. The vertices u and v belong to subtrees rooted at different children of r. The path from u to v in $T(G, \Delta)$ must pass through r. But the distance of r to any of its children is $\Delta/2$. Hence $\mathsf{dist}_{T(G,\Delta)}(u, v) \geq \Delta \geq \Delta(G) \geq c(u, v)$. Since the edge (u, v) is in the cut $(S, V - S)$, $c(u, v) \geq \Delta/4kn$. (This is guaranteed by Lemma B.5). By Lemma B.6, each of u and v are at a distance of at most $k\Delta/2(k-1)$ from r. Hence $\mathsf{dist}_{T(G,\Delta)}(u, v) \leq k\Delta/(k-1) \leq 4k^2 n/(k-1) \cdot c(u, v)$. $\square$

Define $\mathsf{cost}(T) \triangleq \sum_{u,v \in T} x(u, v) \cdot \mathsf{dist}_T(u, v)$. Consider the execution of $\mathsf{Build\text{-}Tree}(G, \Delta)$. Let S, G_1, G_2 be as in the description of $\mathsf{Build\text{-}Tree}(G, \Delta)$. T_1 is the tree returned by the call to $\mathsf{Build\text{-}Tree}(G_1, \Delta(G_1))$ and T_2 is the tree returned by the call to $\mathsf{Build\text{-}Tree}(G_2, \Delta)$. Let T be the tree returned by the procedure. Then,

Lemma B.9

$$\mathsf{cost}(T) \leq \mathsf{cost}(T_1) + \mathsf{cost}(T_2) + \frac{k}{k-1} \Delta \cdot \mathsf{cap}(S, V - S). \tag{B.4}$$

Proof. Consider a pair of vertices $u, v \in T$. Its contribution to $\mathsf{cost}(T)$ is $x(u, v) \cdot \mathsf{dist}_T(u, v)$. We will prove that the contribution of this pair to the LHS of the inequality (B.4) is bounded by the contribution to the RHS.

If both $u, v \in T_1$, the contribution of the pair to $\mathsf{cost}(T_1)$ is $x(u, v) \cdot \mathsf{dist}_{T_1}(u, v) = x(u, v) \cdot \mathsf{dist}_T(u, v)$. Similarly, if both $u, v \in T_2$, the contribution of the pair to $\mathsf{cost}(T_2)$ is $x(u, v) \cdot \mathsf{dist}_{T_2}(u, v) = x(u, v) \cdot \mathsf{dist}_T(u, v)$. On the other hand, if $(u, v) \in (S, V - S)$, both u and

v are at a distance of at most $k\Delta/2(k-1)$ from the root of T. Thus $\mathsf{dist}_T(u,v) \leq k\Delta/(k-1)$. The contribution of (u,v) to the LHS is $x(u,v)\cdot \mathsf{dist}_T(u,v) \leq x(u,v)\cdot k\Delta/(k-1)$ which is the contribution to the RHS. □

Lemma B.10

$$\sum_{u,v\in V(G)} x(u,v)\cdot \mathsf{dist}_T(u,v) = O(\frac{k^2}{k-1}\log n \log\log n)\sum_{u\neq v} x(u,v)\cdot c(u,v)$$

Proof. Note that the LHS is simply $\mathsf{cost}(T(G,\Delta))$. Lemma B.9 establishes that $\mathsf{cost}(T)$ satisfies the divide and conquer recurrence (B.4). This coupled with the cut procedure guaranteed by Lemma B.5 and the analysis of Even et al. [45] proves the lemma. The analysis in [45] does not depend on the subgraph (specifically the guarantee that all of the vertices are within a certain distance from a special vertex) found by the cut, but only on the volume of the subgraph. □

We summarize the results of the previous lemmas in the following theorem, which is identical to Theorem 10.7 in the main text.

Theorem B.11 *Let T be the tree returned by* Build-Tree(G,Δ)*. Then,*

1. *T is a k-HST.*
2. *Any leaf of T is within a distance $k\Delta/2(k-1)$ of the root.*
3. *Let (u,v) be an edge in G. Then $c(u,v) \leq \mathsf{dist}_T(u,v) \leq \frac{2k^2n}{k-1}\cdot c(u,v)$.*
4. $\sum_{u,v\in V(G)} x(u,v)\cdot \mathsf{dist}_T(u,v) = O(\frac{k^2}{k-1}\log n \log\log n)\sum_{u\neq v} x(u,v)\cdot c(u,v)$

Remark B.12 *The tree constructed in Theorem B.11 can be modified to obtain a different tree T' whose vertex set is identical to that of G such that $c(u,v) \leq \mathsf{dist}_{T'}(u,v) \leq \frac{2k^2n}{k-1}\cdot c(u,v)$ for all $u,v\in G$, and $\sum_{u,v\in V(G)} x(u,v)\cdot \mathsf{dist}_T(u,v) = O(\frac{k^2}{k-1}\log n\log\log n)\sum_{u\neq v} x(u,v)\cdot c(u,v)$*

B.2 Using the PST framework with an approximate dual procedure

Consider the following LP, where $\{T_1,\ldots,T_N\}$ is the set of all spanning trees of G which satisfy the property $d_{T_i}(e) \leq \rho c(e)$ for all edges $e\in E$. Since $c(e)$ defines a metric space,

$d_T(e)/c(e) \geq 1$ for all trees T and all edges e. The quantity ρ is called the width of the set $\{T_1, \ldots, T_N\}$. N may be as large as n^{n-2} so the LP has an exponential number of variables. The number of constraints, however, is m, the number of edges in G.

$$\begin{array}{lrcl} \text{Minimize} & \lambda & & \\ \text{such that} & \sum_i d_{T_i}(e) x_i & \leq & \lambda c(e) \\ \text{and} & \sum_i x_i & = & 1 \end{array}$$

In other words , we have to solve the problem $\min\{\lambda : Dx \leq \lambda c, \sum_i x_i = 1\}$ where $D_{e,i} = d_{T_i}(e)$. Let D_e refer to the e-th row of D. Notice that $D_e x \leq \rho c_e$ for all x satisfying $\sum_i x_i = 1$.

For now, we assume that ρ is some polynomial in n. Invoking Theorem 10.7, we have a polynomial time subroutine Build-Tree (Section 10.2) that takes demands y_e and finds a solution T to the minimum communication cost spanning tree with the following properties:

1. T is one of $T_1 \ldots T_N$, where the width ρ of the trees $T_1 \ldots T_N$ is $O(n)$..
2. $\sum_e d_T(e) y_e \leq O(\log n \log\log n) \sum_e y_e c_e$.
3. $\sum_e d_T(e) y_e \leq \beta \sum_e d_{T^*}(e) y_e$ for any tree T^* (not necessarily in $T_1 \ldots T_N$), where $\beta = O(\log n \log\log n)$.

While we do not use the fact explicitly in our construction or proof, it is worthwhile to point out that the minimum communication cost spanning tree problem constitutes a separation oracle for the dual of the linear program we are trying to solve. Let $C(y) = \sum_e d_T(e) y_e$ refer to the cost of the solution produced by Build-Tree , and let $C^*(y) = \min_T \sum_e D_e y_e$. No convex combination of trees can result in a lower value than $C^*(y)$. We now use the subroutine Build-Tree to obtain an approximately optimal solution (x, λ) to the fractional packing LP defined above.

The quantity $\sum_e c_e y_e$ is just $y^t c$ and the quantity $\sum_e \sum_i x_i d_{T_i}(e)$ is just $y^t Dx$. We call the variables x primal and the variables y dual. Throughout this section we will only concern ourselves with primal solutions for which some inequality $D_e x \leq \lambda c_e$ is tight. Therefore we will sometimes call x the primal solution, defining λ implicitly.

Lemma B.13 *Let (x, λ) be any feasible primal solution, and let $y > 0$ be positive values associated with the edges. Then $\lambda \geq C(y)/(\beta \sum_e c_e y_e)$. In particular, $\lambda^* \geq C(y)/(\beta \sum_e c_e y_e)$.*

Proof. Since $\lambda c \geq Dx$ and $y > 0$ we can claim $y^t \lambda c \geq y^t Dx$. But $y^t Dx \geq C^*(y) \geq C(y)/\beta$. Therefore, $\lambda \geq C(y)/(\beta y^t c)$. □

Lemma B.14 *Suppose* $(1+\epsilon)C(y)/(y^t c) \geq \lambda$ *for some feasible solution* (x, λ) *and some* $y > 0$. *Then* $\lambda \leq (1+\epsilon)O(\log n \log\log n)$ *and further,* $\lambda \leq (1+\epsilon)\beta\lambda^*$.

Proof. The subroutine Build-Tree guarantees that $C(y)/(y^t c) \leq O(\log n \log\log n)$. Therefore, $\lambda \leq (1+\epsilon)O(\log n \log\log n)$. The second part follows from Lemma B.13. □

If the precondition in Lemma B.14 is true then we have the approximation guarantee we are looking for. The rest of the proof shows that the precondition of the above lemma can be satisfied after finding $\tilde{O}(\rho/\epsilon^2)$ trees (ie. with only $\tilde{O}(\rho/\epsilon^2)$ components of x being strictly positive). We first define relaxed optimality constraints $\tilde{P}_1$ and $\tilde{P}_2$ which together imply the precondition we require.

$\tilde{P}_1$**:** $(1-\epsilon)\lambda y^t c \leq y^t Dx$;

$\tilde{P}_2$**:** $y^t Dx - C(y) \leq \epsilon(y^t Dx + \lambda y^t c)$.

Lemma B.15 $\tilde{P}_1$ *and* $\tilde{P}_2$ *together imply that* $\lambda \leq \dfrac{C(y)}{(1-3\epsilon)y^t c}$.

If $\epsilon < 1/6$ then $1/(1-3\epsilon) < 1+6\epsilon$. In the rest of this section we will assume that we are looking for a $1+6\epsilon$ approximation and that $\epsilon < 1/6$.

The algorithm runs in phases, attempting to improve the primal solution in each phase. At the beginning of the first phase, the primal solution has one of the x_i's (chosen arbitrarily) set to 1 and the rest to 0. In each phase, let λ_0 be the value of λ at the beginning of that phase. Then let $\alpha = \dfrac{4\ln(2m/\epsilon)}{\epsilon\lambda_0}$ and $\sigma = \epsilon/(4\alpha\rho)$. Given primal variables x we define the dual variables y as $y_e = \dfrac{1}{c_e} \cdot e^{\alpha \cdot \frac{D_e x}{c_e}}$.

Claim B.16 *The relaxed optimality condition* $\tilde{P}_1$ *is satisfied for any feasible primal solution* x *with* $\lambda \geq \lambda_0/2$ *and the dual variables* y *as defined above.*

Within each phase, there are several iterations to improve the primal solution. During each iteration the dual variables y are computed as outlined above and then subroutine Build-Tree is called with these values of y to yield a tree T_j and a cost $C(y)$. The tree corresponds to a trivial primal solution $\hat{x}$ with $\hat{x}_j = 1$ and $\hat{x}_i = 0$ if $i \neq j$. Let the old primal solution be x. Now if any of the following two conditions is true, the phase terminates.

1. $\lambda \le \frac{1}{1-3\epsilon} \cdot C(y)/(y^t c)$.

2. $\lambda \le \lambda_0/2$.

If the first condition is true then we have satisfied the precondition for Lemma B.14 and we are done; if the second is true, we set λ_0 to the current λ and start the next phase. If neither condition holds then the new primal solution is set to $(1-\sigma)x + \sigma\hat{x}$ and the next iteration within the current phase starts.

We now show that the number of iterations within a phase is small.

Lemma B.17 *The number of iterations within a phase is* $O\left(\frac{\rho \ln(m/\epsilon)}{\epsilon^3 \lambda_0}\right)$. *Also if* $\lambda_0 \le 1 + \gamma\epsilon$ *then the number of iterations is* $O\left(\frac{\gamma\rho \ln(m/\epsilon)}{\epsilon^2}\right)$ *for any* $\gamma > 1$.

Proof. During a phase $\tilde{P}_1$ is always satisfied and therefore $\tilde{P}_2$ is always violated (since $\tilde{P}_1$ and $\tilde{P}_2$ together imply $1/(1-3\epsilon)$-optimality which allows the phase to terminate. We define a potential function

$$\Phi = y^t c = \sum_e e^{\alpha \cdot \frac{D_e x}{c_e}}.$$

To prove this lemma we show a large decrease in the potential during each iteration. Let Φ be the potential at the beginning of an iteration and Φ' at the end.

$$\begin{aligned} \Phi - \Phi' &= \sum_e (e^{\alpha \frac{D_e x}{c_e}} - e^{\alpha \frac{D_e((1-\sigma)x + \sigma\hat{x})}{c_e}}) \\ &= \sum_e y_e c_e (1 - e^{\alpha\sigma \frac{D_e(\hat{x}-x)}{c_e}}) \end{aligned}$$

We know that $1 \le D_e x/c_e \le \rho$ and $1 \le D_e \hat{x}/c_e \le \rho$. Therefore $|\alpha\sigma(D_e\hat{x} - D_e x)/c_e| \le \epsilon/4$. For $-1 \le \delta \le 1$ it is easy to show that $e^\delta \le 1 + \delta + \delta^2$. Therefore

$$\begin{aligned} \Phi - \Phi' &\ge \sum_e y_e \left(\alpha\sigma(D_e x - D_e\hat{x})\right) - \sum_e y_e (\epsilon/4)\alpha\sigma D_e(x + \hat{x}) \\ &= \alpha\sigma \Big((y^t D x - C(y)) - (\epsilon/4)(y^t D x + C(y))\Big) \end{aligned}$$

Since $\tilde{P}_2$ does not hold, $y^t D x > C(y)$. Therefore $\Phi - \Phi' > \alpha\sigma(y^t Dx - C(y) - \epsilon y^t D x)$. Again using the fact that $\tilde{P}_2$ does not hold, we obtain $y^t Dx - C(y) - \epsilon y^t D x > \epsilon\lambda y^t c$, which implies that $\Phi - \Phi' > \alpha\sigma\epsilon\lambda y^t c = \alpha\sigma\epsilon\lambda\Phi$. Therefore, Φ must halve every $O(1/(\alpha\sigma\epsilon\lambda)) = O(\frac{\rho}{\epsilon^2\lambda})$ iterations.

During the phase $e^{\alpha\lambda_0/2} \le \Phi \le me^{\alpha\lambda}$. Also, if $\lambda = 1 + \gamma\epsilon$ then $e^{\alpha} \le \Phi \le e^{\alpha(1+\gamma\epsilon)}$. These two facts, combined with the large decrease in Φ in each iteration, suffice to prove the lemma. $\square$

Since the number of phases is $O(\log \lambda)$ and $\lambda \le \rho$, we can immediately conclude that the total number of iterations is $O(\frac{\rho \log \rho \log(m/\epsilon)}{\epsilon^3})$. Using the ϵ-scaling trick due to [116] we can decrease the number of iterations significantly.

Theorem B.18 *The total number of iterations over all phases is* $O(\rho \ln(m/\epsilon)/\epsilon^2)$.

The algorithm terminates only when $\lambda \le \frac{1}{1-3\epsilon} \cdot C(y)/(y^t c)$. But Theorem B.18 guarantees termination after a small number of iterations. Further the subroutine Build-Tree finds one tree in each iteration. This allows us to state the following theorem; Theorem 10.20 in the main text follows using exactly the same argument.

Theorem B.19 *The above algorithm results in a* $(1+\epsilon)O(\log n \log\log n)$*-probabilistic approximation of a graph metric using* $O(\rho \log(m/\epsilon)/\epsilon^2)$ *trees. Further, if the best possible probabilistic approximation for this metric is* λ^**, the above algorithm finds a* $\beta\lambda^*$*-probabilistic approximation.*

The rationale behind β:

The quantity β does not play any role in our algorithm. Nor does it benefit us to have a small ϵ. The reason we introduced these quantities is the following. Suppose a PTAS/constant factor approximation could be found for the minimum communication cost spanning tree problem. Then we would automatically have a PTAS/constant factor approximation algorithm for λ^*, again using polynomially many trees. While this may not give a better guarantee on the expected stretch of an edge, it will guarantee that out of all possible probabilistic approximations of the given metric space using trees, we will find one which is close to best. Even Max-SNP hardness is not known for the minimum communication cost spanning tree problem. Also, Wu et al. [136] recently gave a PTAS for a special case.

Bibliography

[1] A. Agrawal, P. Klein, and R. Ravi. When trees collide: An approximation algorithm for generalized steiner tree problem on networks. *23rd ACM Symposium on Theory of Computing*, pages 134–144, 1991.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1976.

[3] N. Alon and Y. Azar. On-line steiner trees in the euclidean plane. *Discrete and Computational Geometry*, 10(2):113–121, 1993.

[4] N. Alon, R.M. Karp, D. Peleg, and D. West. A graph-theoretic game and its application to the k-server problem. *SIAM J. Comput.*, 24(1):78–100, 1995.

[5] M. Andrews, B. Awerbuch, A. Fernandez, J. Kleinberg, T. Leighton, and Z. Liu. Universal stability results for greedy contention-resolution protocols. *37th IEEE symposium on Foundations of Computer Science*, pages 380–389, 1996.

[6] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line load balancing with applications to machine scheduling and virtual circuit routing. *25th ACM Symposium on Theory of Computing*, pages 623–31, 1993.

[7] A. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg. Improved routing strategies with succinct tables. *Journal of Algorithms*, 11(3):307–41, 1990.

[8] B. Awerbuch and Y. Azar. Buy-at-bulk network design. *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science*, pages 542–47, 1997.

[9] B. Awerbuch, Y. Azar, A. Blum, and S. Vempala. Improved approximation guarantees for minimum weight k-trees and prize-collecting salesmen. *27th ACM Symposium on Theory of Computing*, pages 277–283, 1995.

[10] B. Awerbuch, Y. Azar, A. Fiat, and T. Leighton. Making commitments in the face of uncertainty: How to pick a winner almost every time. *28th ACM Symposium on Theory of Computing*, pages 519–530, 1996.

[11] B. Awerbuch, Y. Azar, and S. Plotkin. Throughput competitive online routing. *34th IEEE symposium on Foundations of Computer Science*, pages 32–40, 1993.

[12] B. Awerbuch, A. Baratz, and D. Peleg. Cost-sensitive analysis of communication protocols. *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pages 177–87, 1990.

[13] B. Awerbuch and T. Singh. Online algorithms for selective multicast and maximal dense trees. *29th ACM Symposium on Theory of Computing*, 1997.

[14] Y. Azar, A. Broder, and A. Karlin. On-line load balancing. *Proc. 33rd IEEE Annual Symposium on Foundations of Computer Science*, pages 218–225, 1992.

[15] A. Ballardie, P. Francis, and J. Crowcroft. Core based trees(cbt) - an architecture for scalable inter-domain multicast routing. *ACM SIGCOMM*, pages 85–95, 1993.

[16] J. Bar-Ilan, G. Kortsarz, and D. Peleg. Generalized submodular cover problems and applications. *Fourth Israeli Symposium on Theory of Computers and Systems*, pages 110–118, 1996.

[17] Y. Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. *37th IEEE symposium on Foundations of Computer Science*, pages 184–193, 1996.

[18] Y. Bartal. On approximating arbitrary metrics by tree metrics. *30th ACM Symposium on Theory of Computing*, 1998.

[19] Y. Bartal, S. Leonardi, and A. Fiat. Lower bounds for on-line graph problems with application to on-line circuit and optical routing. *Proc. of the 28th Symposium on Theory of Computation*, pages 531–540, 1996.

[20] C.D. Bateman, C.S. Helvig, G. Robins, and A. Zelikovsky. Provably good routing tree construction with multi-port terminals. *Proceedings of the 28th ACM/SIGDA International Symposium on Physical Design*, 1997.

[21] J.E. Beck and D.P. Siewiorek. Modeling multicomputer task allocation as a vector packing problem. *9th International Symposium on Systems Synthesis*, pages 115–120, 1996.

[22] M. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. *Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 270–279, 1998.

[23] P. Berman and V. Ramaiyer. Improved approximation algorithms for the steiner tree problem. *J. of Alg.*, 17:381–408, 1994.

[24] M. Bern and P. Plassmann. The steiner problems with edge lengths 1 and 2. *Information Processing Letters*, 32:171–176, 1989.

[25] K. Bharath-Kumar and J.M. Jaffe. Routing to multiple destinations in computer networks. *IEEE Transactions on Communications*, 31(3):343–51, 1983.

[26] A. Blum, R. Ravi, and S. Vempala. A constant factor approximation for the k-mst problem. *28th ACM Symposium on Theory of Computing*, pages 442–448, 1996.

[27] A. Borodin, J. Kleinberg, P. Raghavan, A. Sudan, and D. Williamson. Adversarial queueing theory. *28th ACM Symposium on Theory of Computing*, pages 376–385, 1996.

[28] S. Chakrabarti, C. Phillips, A. Schulz, D. Shmoys, C. Stein, and J. Wein. Improved scheduling algorithms for minsum criteria. *23rd International Colloquium on Automata, Languages, and Programming*, 1996.

[29] M. Charikar, C. Chekuri, T. Cheung, A. Goel, S. Guha, and M. Li. Approximation algorithms for directed steiner problems. *Ninth Annual ACM-SIAM Symposium on Discrete Algorithm*, pages 192–200, 1998.

[30] M. Charikar, C. Chekuri, A. Goel, and S. Guha. Rounding via trees: Deterministic approximation algorithms for group steiner trees and k-median. *30th ACM Symposium on Theory of Computing*, 1998.

[31] M. Charikar, C. Chekuri, A. Goel, S. Guha, and S. Plotkin. Approximating a finite metric by a small number of tree metrics. *39th IEEE Symposium on Foundations of Computer Science*, 1998.

[32] M. Charikar, S. Guha, D. Shmoys, and E. Tardos. A constant-factor approximation algorithm for the k-median problem. *Proceedings of the 31st ACM Symposium on Theory of Computing*, pages 1–10, 1999.

[33] M. Charikar and B. Raghavachari. The finite capacity dial-a-ride problem. *Proceedings of the 39th IEEE Symposium on Foundations of Computer Science*, pages 458–467, 1998.

[34] Charny. Providing qos guarantees in input-buffered crossbar switches with speedup. *PhD dissertation, MIT*, 1998.

[35] A. Charny, P. Krishna, N. Patel, and R. Simcoe. Algorithms for providing bandwidth and delay guarantees in input-buffered crossbars with speed up. *Sixth IEEE/IFIP IWQoS*, 1998.

[36] C. Chekuri and S. Khanna. On multi-dimensional packing problems. *Tenth ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[37] C. Chekuri, R. Motwani, B. Natarajan, and C. Stein. Approximation techniques for average completion time scheduling. *Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 609–618, 1997.

[38] J.S.-C. Chen and T.E. Stern. Throughput analysis, optimal buffer allocation, and traffic imbalance study of a generic nonblocking packet switch. *IEEE J. Select. Areas Commun.*, 9(3):439–49, 1991.

[39] S. Chuang, A. Goel, M. McKeown, and B. Prabhakar. Matching output queuing with a combined input output queued switch. *To Appear in Journal on Selected Areas in Communication*, 1999.

[40] D.D. Clark. The design philosophy of the darpa internet protocols. *Computer Communication Review*, 18(4):106–14, 1988.

[41] S. Deering, D.L. Estrin, D. Farinacci, C. Jacobson, V. Liu, and L. Wei. The pim architecture for wide-area multicast routing. *IEEE/ACM Trans on Networking*, 4(2):153–152, April 1996.

[42] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *Internetworking: Research and Experience*, 1(1):3–26, 1990.

[43] M. Doar and I. Leslie. How bad is naive multicast routing. *IEEE INFOCOM*, pages 82–89, 1992.

[44] Y. Dodis and S. Khanna. Designing networks with pairwise bounded distances. *Proceedings of the 31st ACM Symposium on Theory of Computing*, pages 750–59, 1999.

[45] G. Even, J. Naor, S. Rao, and B. Schieber. Divide-and-conquer approximation algorithms via spreading metric. *Proceedings of IEEE 36th Symposium on Foundations of Computer Science*, pages 62–71, 1995.

[46] U. Feige. A threshold of $\ln n$ for approximating set-cover. *Proceedings of the 28th ACM Symposium on Theory of Computing*, pages 314–18, 1996.

[47] U. Feige and J. Kilian. Zero knowledge and the chromatic number. *Proceedings of the Eleventh Annual Conference on Complexity Theory, IEEE*, 1996.

[48] A. Feldmann, M. Kao, J. Sgall, and S. Teng. Optimal online scheduling of parallel jobs with dependencies. *J. of Combinatorial Optimization*, 1(4):393–411, 1998.

[49] D. Gale and L.S. Shapely. College admissions and the stability of marriage. *American Mathematical Monthly*, 69:9–15, 1962.

[50] D. Gamarnik. Stability of adversarial queues via fluid models. *39th IEEE symposium on Foundations of Computer Science*, 1998.

[51] D. Gamarnik. Stability of adaptive and non-adaptive packet routing policies in adversarial queueing networks. *Proceedings of the 31st ACM Symposium on Theory of Computing*, 1999.

[52] J. Garay and I. Gopal. Call preemption in communication networks. *IEEE INFOCOM*, pages 1043–1050, 1992.

[53] J. Garay, I. Gopal, S. Kutten, Y. Mansour, and M. Yung. Efficient on-line call control algorithms. *2nd Annual Israel Conference on Theory of Computing and Systems*, 1993.

[54] M.R. Garey and D.S. Johnson. *Computers and Intractability: A guide to the theory of NP-completeness.* Freeman, San Francisco, 1978.

[55] N. Garg. A 3-approximation for the minimum tree spanning k vertices. *37th IEEE Symposium on Foundations of Computer Science*, pages 302–309, 1996.

[56] N. Garg, G. Konjevod, and R. Ravi. A polylogarithmic approximation algorithm for the group steiner tree problem. *Proceeding of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 253–59, 1998.

[57] N. Garg, V. Vazirani, and M. Yannakakis. Approximate max-flow min-(multi)cut theorems and their applications. *Proceedings of 25th Annual ACM Symposium on the Theory of Computing*, pages 698–707, 1993.

[58] M. Garofalakis and Y. Ioannidis. Scheduling issues in multimedia query optimization. *ACM Computing Surveys*, 27(4):590–592, 1995.

[59] A. Goel, M. Henzinger, and S. Plotkin. Online throughput-competitive algorithm for multicast routing and admission control. *9th ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[60] A. Goel, M. Henzinger, S. Plotkin, and E. Tardos. Scheduling data transfers in a network and the set scheduling problem. *31st ACM Synposium on Theory of Computing*, 1999.

[61] A. Goel, D. Logothetis, D. Kataria, and K.G. Ramakrishnan. An efficient algorithm for constraint-based routing in data networks. *Manuscript*, 1999.

[62] A. Goel and K. Munagala. Extending greedy multicast routing to delay sensitive applications. *Stanford University Tech Note STAN-TN-CSD-99- . Also available from http://cs.stanford.edu/~agoel/papers/steiner_weight_delay_technote.ps*, June 1999.

[63] M. Goemans and J. Kleinberg. An improved approximation ratio for the minimum latency problem. *7th ACM-SIAM Symposium on Discrete Algorithms*, pages 152–158, 1996.

[64] M. Goemans and D. Williamson. A general approximation technique for constrained forest problems. *SIAM J. Comput.*, 24(2):296–317, April 1995.

[65] S. Guha and S. Khuller. Approximation algorithms for connected dominating sets. *Algorithmica*, 20:347–87, 1998.

[66] S. Guha and S. Khuller. Improved methods for approximating node weighted steiner trees and connected dominating sets. *Information and Computation*, 150:57–74, 1999.

[67] A.L. Gupta and N.D. Georganas. Analysis of a packet switch with input and output buffers and speed constraints. *IEEE Infocom*, pages 694–700, 1991.

[68] L.A. Hall, D.B. Shmoys, A.S. Schulz, and J. Wein. Scheduling to minimize average completion time: off-line and on-line approximation algorithms. *Mathematics of Operations Research*, 22:513–544, 1997.

[69] L.A. Hall, D.B. Shmoys, and J. Wein. Scheduling to minimize average completion time: off-line and on-line algorithms. *7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 142–151, 1996.

[70] R. Hassin. Approximation schemes for the restricted shortest path problem. *Mathematics of Operations Research*, 17:36–42, 1992.

[71] D. Hochbaum. *Approximation algorithms for NP-hard problems.* PWS Publishing Company, 1997.

[72] Sung-Pil Hong, Heesang Lee, and Bum Hwan Park. An efficient multicast routing algorithm for delay-sensitive applications with dynamic membership. *Proceedings IEEE INFOCOM*, pages 1433–40, 1998.

[73] I. Iliadis and W.E. Denzel. Analysis of packet switches with input and output queuing. *IEEE Transactions on Communications*, 41(5):731–40, 1993.

[74] M. Imase and B. Waxman. Dynamic steiner tree problem. *SIAM J. Discrete Math.*, 4(3):369–384, August 1991.

[75] V. Jacobson. Congestion avoidance and control. *Computer Communication Review*, 18(4):314–29, 1988.

[76] K. Jain. A factor 2 approximation algorithm for the generalized steiner network problem. *39th IEEE symposium on Foundations of Computer Science*, pages 448–57, 1998.

[77] W. Johnson and J. Lindenstrauss. Extensions of lipschitz mappings into a hilbert space. *Contemporary Mathematics*, 26:189–206, 1984.

[78] A. Kamath, O. Palmon, and S. Plotkin. Routing and admission control in general topology networks with poisson arrivals. *7th ACM-SIAM Symposium on Discrete Algorithms*, pages 269–278, 1996.

[79] D¿ Karger and S. Plotkin. Adding multiple cost constraints to combinatorial optimization problems, with applications to multicommodity flows. *Proc. 27th Annual ACM Symposium on Theory of Computing*, pages 18–25, 1995.

[80] O. Kariv and S.L. Hakimi. An algorithmic approach to network location problems, part ii: p-medians. *SIAM J. Appl. Math.*, 37:539–60, 1979.

[81] M. Karol, M. Hluchyj, and S. Morgan. Input versus output queueing on a space division switch. *IEEE Trans. Comm*, 35(12):1347–1356, Dec 1987.

[82] M. Karpinsky and A. Zelikovsky. New approximation algorithms for the steiner tree problem. *Technical Report, Electronic Colloquium on Computational Complexity,(ECCC): TR95-030*, 1995.

[83] H. Kellerer, T. Tautenhahn, and G.J. Woeginger. Approximability and non-approximability results for minimizing total flow time on a single machine. *28th ACM Symposium on Theory of Computing*, pages 418–426, 1996.

[84] Kelly. Blocking probabilities in large circuit-switched networks. *Advances in Appl. Prob*, 18:473–505, 1986.

[85] S. Khuller, B. Raghavachari, and N. Young. Balancing minimum spanning and shortest path trees. *Algorithmica*, 14(4):305–321, 1994.

[86] P. Klein, S. Plotkin, and S. Rao. Shallow excluded minors and improved graph decompositions. *25th ACM Symposium on Theory of Computing*, 1993.

[87] P. Klein, S. Plotkin, S. Rao, and E. Tardos. Approximation algorithms for steiner and directed multicuts. *Journal of Algorithms*, 22(2):241–269, 1997.

[88] P. Klein, S. Plotkin, C. Stein, and E. Tardos. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. *SIAM Journal on Computing*, 23(3):466–487, June 1994.

[89] P. Klein and R. Ravi. A nearly best-possible approximation algorithm for node-weighted steiner trees. *J. of Algorithms*, 19(1):104–14, 1995.

[90] J. Kleinberg and E. Tardos. Disjoint paths in densely embedded graphs. *36th IEEE symposium on Foundations of Computer Science*, pages 52–61, 1995.

[91] V.P. Kompella, J.C. Pasquale, and G.C. Polyzos. Multicast routing for multimedia communication. *IEEE/ACM Transactions on Networking*, 1(3):286–92, June 1993.

[92] G. Konjevod, Ravi.R., and F.S. Salaman. On approximating planar metrics by tree metrics. *Manuscript*, page 1997, 1997.

[93] G. Kortsarz and D. Peleg. Approximating shallow-light trees. *Proceeding of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 103–110, 1997.

[94] L. Kou, G. Markowsky, and L. Berman. A fast algorithm for steiner trees. *Acta Informatica*, 15:141–45, 1981.

[95] P. Krishna, N. Patel, A. Charny, and R. Simcoe. On the speedup required for work-conserving ceossbar switches. *Sixth IEEE/IFIP IWQoS*, 1998.

[96] T. Leighton, F. Makedon, S. Plotkin, C. Stein, E. Tardos, and S. Tragoudas. Fast approximation algorithms for multicommodity flow problems. *J. Comp. and Syst. Sci.*, 50:228–43, 1995.

[97] T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with application to approximation algorithms. *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 422–31, 1988.

[98] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. *Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 110–119, 1997.

[99] M. Luby and N. Nisan. A parallel approximation algorithm for positive linear programming. *Proceedings of 25th Annual Symposium on the Theory of Computing*, pages 448–57, 1993.

[100] Q. Ma and P. Steenkiste. Quality-of-service routing for traffic with performance guarantees. *Proc. IFIP Fifth International Workshop on Quality of Service*, pages 115–26, 1997.

[101] Y. Ma and S. Plotkin. Improved lower bounds for load balancing of tasks with unknown duration. *Information Processing Letters*, 62, 1997.

[102] M. Macedonia and D. Brutzman. Mbone provides audio and video across the net. *IEEE Computer*, pages 30–36, April 1994.

[103] G. Malkin (Editor). Internet users' glossary. *IETF Request For Comments 1392*, August 1996.

[104] S. McCanne. Scalable compression and transmission of internet multicast video. *PhD Thesie, Univ. of Calif. Berkeley*, 1996.

[105] N. McKeown, V. Anantharam, and J. Walrand. Achieving 100% throughput in an input-queued switch. *IEEE INFOCOM*, pages 296–302, 96.

[106] N. McKeown, M. Izzard, A. Mekkittikul, W. Ellersick, and M. Horowitz. The tiny tera: A packet switch core. *Hot Interconnects V, Stanford University*, August 1996.

[107] Bay Networks. Private network network interface routing software design. 1996.

[108] Y. Oie, M. Murata, K. Kubota, and H. Miyahara. Effect of speedup in nonblocking packet switch. *Proc. ICC*, pages 410–14, 1989.

[109] M. Parsa, Qing Zhu, and J.J. Garcia-Luna-Aceves. An iterative algorithm for delay-constrained minimum-cost multicasting. *IEEE/ACM Transactions on Networking*, 6(4):361–74, 1998.

[110] C. Partridge et al. A 50-gb/s ip router. *IEEE/ACM Transactions on Networking*, 6(3):237–48, June 1998.

[111] D. Peleg and E. Reshef. Deterministic polylog approximation for minimum communication spanning tree. *25th International Colloquium on Automata, Languages, and Programming*, 1998.

[112] D. Peleg and E. Upfal. A tradeoff between space and efficiency for routing tables. *JACM*, 36(3):510–30, 1989.

[113] C. Phillips, C. Stein, E. Torng, and J. Wein. Critical scheduling via resource augmentation. *29th ACM Symposium on Theory of Computing*, 1997.

[114] C. Phillips, C. Stein, and J. Wein. Scheduling jobs that arrive over time. *Lecture Notes in Computer Science 955*, pages 290–301, Springer, Berlin.

[115] G. Pisier. *The Volume of Convex Bodies and Banach Space Geometry.* Cambridge University Press, 1989.

[116] S. Plotkin, D. Shmoys, and E. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Math of Oper. Research*, pages 257–301, 1994.

[117] B. Prabhakar and N. McKeown. On the speedup required for combined input and output queued switching. *To appear in Automatica*, 99.

[118] P. Raghavan. Probabilistic construction of deterministic algorithms: Approximating packing integer problems. *Journal of Comp. Sys. Sci.*, 37:130–43, 1988.

[119] P. Raghavan and C. Thompson. Randomized rounding. *Combinatorica*, 7:365–374, 1987.

[120] S. Ramanathan. Multicast tree generation in networks with asymmetric links. *Proc. of IEEE INFOCOM*, pages 558–68, 1996.

[121] G. Reich and P. Widmayer. Beyond steiner's problem: A vlsi oriented generalization. *Proc. of the 15-th Int. Workshop on Graph-Theoretic Concepts in Computer Science, LNCS 411*, 1989.

[122] N. Robertson and P.D. Seymour. Recent results on graph minors. *Surveys in Combinatorics*, Cambridge University Press, 1985.

[123] H.F. Salama. Evaluation of multicast routing algorithms for real-time communication on high-speed networks. *IFIP Sixth International Conference on High Performance Networking*, pages 27–42, 1995.

[124] P.D. Seymour. Packing directed circuits fractionally. *Combinatorica*, 15(2):281–88, 1995.

[125] J. Sgall. Randomized on-line scheduling of parallel jobs. *J. of Algorithms*, 21:149–175, 1996.

[126] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Comm. of the ACM*, 28(2):202–208, 1985.

[127] H. Takahashi and A. Matsuyama. An approximate solution for the steiner problem in graphs. *Math.Japonica*, 24:573–77, 1980.

[128] Y.T. Tsai and C.Y. Tang. An on-line algorithm for computing low distance spanning trees on euclidean space. *Computer Science and Informatics*, 22(2):3–16, 1992.

[129] D. Waitzman, C. Partridge, and S. Deering. Distance vector multicast routing protocol. *Internet RFC 1075*, 1988.

[130] N.B. Wang and R. Chang. An upper bound for the average length of the euclidean minimum spanning tree. *International Journal of Computer Mathematics*, 30(1-2):1–12, 1989.

[131] Z. Wang and J. Crowcroft. Quality-of-service routing for supporting multimedia applications. *JSAC*, 14(7):1228–34, 1996.

[132] B. Waxman. Routing of multipoint connections. *IEEE J. on Sel. Areas in Commun.*, 6(9):1617–22, 1988.

[133] L. Wei and D. Estrin. The trade-offs of multicast trees and algorithms. *International Conference on Computer Communications and Networks*, 1994.

[134] D. Williamson, M. Goemans, M. Mihail, and V. Vazirani. A primal-dual approximation algorithm for generalized steiner network problems. *25th ACM Synposium on Theory of Computing*, pages 708–717, 1993.

[135] P. Winter. Steiner problem in networks: a survey. *Networks*, 17:129–167, 1987.

[136] B.Y. Wu, G. Lancia, V. Bafna, K. Chao, R. Ravi, and C.Y. Tang. A polynomial time approximation scheme for minimum routing cost spanning trees. *Proceeding of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 21–32, 1998.

[137] A. Zelikovsky. An 11/6 approximation algorithm for the network steiner problem. *Algorithmica*, 9:463–70, 1993.

[138] A. Zelikovsky. A polynomial-time subpolynom-approximation scheme for the acyclic directed steiner tree problem. *Tech. Report 85113-CS, University of Bonn*, 1994.