Parallel and Dynamic Shortest-Path Algorithms for Sparse Graphs

Sairam Subramanian

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-95-04
May 1995

Parallel and dynamic shortest-path algorithms for sparse graphs

by

Sairam Subramanian

M. Sc.(Tech), Computer Science, Birla Institute of Technology and Science, 1988

M. Sc.(Hons), Mathematics, Birla Institute of Technology and Science, 1988

Sc. M, Computer Science, Brown University, 1990

Thesis

Submitted in partial fulfillment of the requirements for the

Degree of Doctor of Philosophy in the Department of Computer Science

at Brown University.

May 1995

This dissertation by Sairam Subramanian

is accepted in its present form by the Department of

Computer Science as satisfying the

dissertation requirement for the degree of Doctor of Philosophy.

Recommended to the Graduate Council

Date ______________ ______________________________

Philip N. Klein

Date ______________ ______________________________

Roberto Tamassia

Date ______________ ______________________________

Jeffrey S. Vitter

Approved by the Graduate Council

Date ______________ ______________________________

Vita

Sairam Subramanian was born in New Delhi, India on the third of December 1965. He completed his schooling at the D.T.E.A. Sr. Secondary School, Mandir Marg, New Delhi in May 1983. He received his M.Sc.(Tech) in Computer Science and M.Sc.(Hons) in Mathematics from Birla Institute of Technology and Science, Pilani, in May 1988. He received his Sc.M. in Computer Science from Brown University in May 1990. He is currently a postdoctoral researcher at the Department of Computer Science in University of Texas at Austin.

Acknowledgments

There are many people at Brown and elsewhere who have been instrumental in my reaching this goal. First and foremost I would like to thank my mother Meenakshi for believing that her children were capable of anything and instilling in us a desire to be the best in whatever we did. I would also like to thank my high school teachers Mr. Jaypal Chandra and Ms. Bhuvaneshvari for showing me that education could be fun, and Professors. M.V. Tamhankar, and H. Subramanian for some truly inspiring courses in mathematics.

At Brown, I would like to thank Professors Philip Klein, Roberto Tamassia, and Jeff Vitter for advising this thesis and for teaching me much of what I know. I would like to thank Prof. Vitter for introducing me to research and for his confidence in my abilities. His constant encouragement kept me motivated during times when the going was tough. I would like to thank Prof. Tamassia for encouraging my interest in dynamic graph algorithms and for suggesting the problem solved in Chapter 5. A large portion of the results in this thesis were obtained in joint work with Prof. Phil Klein. I would like to thank him for his boundless enthusiasm for research and for the innumerable creative insights that led to the discovery of the results in this thesis. My thanks also to Satish Rao and Monika Rauch with whom I collaborated on the work presented in Chapter 6.

I would also like to thank Professors. Paris Kanellakis, Robert Netzer, and John Savage, and R. Ravi, Bob Cohen, P. Krishnan, and R. Sridhar for many stimulating discussions. My thanks also to Ajit, Uma, Mala, and Shilpa who made my stay at Providence a truly delightful one. My thanks to Mary, Dawn, Michele, Janet, Trina, and the people at SPOC who kept everything at Brown moving smoothly and gave the students no reason to complain.

Finally I would like to thank my wife Bharathi without whose support this thesis would never have been completed. She has been a colleague, a friend, wife, and my biggest source of encouragement. At every step in my research she has helped me weed out the bad ideas and has been instrumental in the discovery of a number results presented in this thesis.

Contents

Vita iii

Acknowledgments iv

1 Introduction 1

1.1 Computing shortest paths in graphs with nonnegative edge weights . . . 1

1.1.1 The parallel shortest-path problem 2

1.1.2 The dynamic shortest path problem 2

1.1.3 Finding shortest paths in parallel and dynamic settings 3

1.2 Computing shortest paths in graphs with negative edge weights 6

1.3 Background and overview 7

1.3.1 Models of computation 8

1.4 Planar separators, cluster decompositions, and the decomposition tree . 9

1.4.1 Separators 9

1.4.2 One-way separators 10

1.4.3 Cluster partitions 11

1.4.4 The decomposition tree 14

1.5 The Ullman-Yannakakis random-sampling technique 15

1.6 Path-detour: A new technique for finding and representing paths that cross a separating path 17

2 Fully dynamic data structures for shortest paths and reachability in

planar graphs **21**

2.1 Face-boundary shortest-path substitutes: sparse representation of approximate shortest paths . . . 26

2.1.1 Path-detour: A basic sparsification technique . . . 26

2.1.2 Face-boundary shortest-path substitutes . . . 28

2.2 Face-boundary reachability substitutes: sparse representation for all-pairs reachability . . . 32

2.3 Fully dynamic data structures for approximate shortest paths and reachability . . . 39

3 A randomized parallel algorithm for computing shortest paths in directed graphs **43**

3.1 Computing exact shortest paths by repeated approximation . . . 46

3.2 Parallel BFS and the Ullman-Yannakakis algorithm . . . 48

3.2.1 Limited search . . . 49

3.2.2 The Ullman-Yannakakis algorithm . . . 49

3.2.3 Approximating k-limited shortest paths . . . 51

3.3 Procedure FullSearch . . . 52

3.4 Using procedure FullSearch to solve the $(1-1/2)$-approximate shortest-path problem . . . 54

4 A linear-processor polylog-time algorithm for shortest paths in planar digraphs **58**

4.1 The ingredients . . . 61

4.1.1 Limited Bellman-Ford search from a source-path . . . 61

4.1.2 Summaries . . . 63

4.1.3 Applying limited Bellman-Ford to a summary . . . 64

4.1.4 Parallel Path-detour: Fast parallel approximation of k-hop paths that cross P . . . 65

4.1.5 The derivation dag . . . 68

4.2 Near-covers 69
4.2.1 Finding a near-cover in parallel 71
4.3 The shortest-path approximation algorithm 75
4.3.1 Reducing the size and diameter of a summary 76
4.3.2 Constructing a graph with small diameter 84
4.4 Extending the algorithm to general graphs 87
4.4.1 Counting intersections in summaries of general graphs 87
4.4.2 Solving the shortest-path problem on general graphs 92
4.5 Extending the parallel algorithm to handle negative edges 93
4.5.1 Limited Bellman-Ford in a graph with nonpositive edge-weights . 94
4.5.2 Finding approximate shortest paths with nonpositive edge weights 98
4.5.3 Constructing a graph with small diameter when the underlying edges have nonpositive weight 100
4.5.4 Computing a maximum flow in an undirected planar graph . . . 102

5 An efficient parallel algorithm for shortest paths in planar layered digraphs 108
5.1 Preliminaries 110
5.2 Shortest paths in a planar layered *st*-graph 114
5.3 The Tube minima problem and a more efficient solution 123
5.4 Remarks 124

6 Shortest paths in the presence of negative edge weights 126
6.1 A three-phase algorithm for single-source shortest-paths 131
6.1.1 Preliminaries 131
6.1.2 The three-phase algorithm 132
6.2 A fully-dynamic data structure for shortest paths in planar graphs when edges can have arbitrary integral values 137
6.3 Comments 140

Chapter 1

Introduction

Computing single-source shortest paths is a fundamental and ubiquitous problem in network analysis. Aside from the importance of this problem in its own right, often the problem arises in the solution of other problems (e.g. network flow and matching).

In this thesis we consider the problem of computing shortest paths in parallel, dynamic, and sequential realms. We also consider the related problem of graph-reachability in these settings.

1.1 Computing shortest paths in graphs with nonnegative edge weights

The single-source shortest-path problem is defined as follows: "Given a graph G with with weighted edges and a distinguished source node s, determine the lowest weight path from s to all the other nodes." The all-pairs shortest-path problem requires us to determine the least-weight path between every pair of nodes.

For the case when all the edge-weights of the graph are restricted to be nonnegative, the known sequential algorithms for the single-source problem are quite efficient; they require only slightly more than linear time [3,31,32,53]. Unfortunately, in the parallel and dynamic settings this problem is not well understood.

1.1.1 The parallel shortest-path problem

In the parallel setting algorithms that run in polylogarithmic time and use a polynomial number of processors are known, but the processor bounds are large. These algorithms use repeated squaring of $n \times n$ matrices, and hence require about n^3 processors. Thus the *work* done by these algorithms (work = time $\times$ number of processors, essentially the total number of operations) far exceeds the work done by the sequential algorithms. An *efficient* parallel algorithm for a problem is one that does work exceeding the sequential time by no more than a polylog factor. This polylog factor represents the overhead in going from a sequential to a parallel solution.

It is a longstanding open problem to find an efficient polylog-time algorithm for shortest paths. Our inability to find efficient parallel algorithms for this problem and related ones has been termed the *transitive-closure bottleneck*. The gap between what we can and cannot achieve is particularly evident when we consider sparse graphs, graphs where the number of edges is a small factor times the number of nodes. For such graphs, the work required by known polylog-time algorithms is essentially the cube of the sequential time required. Put another way, using p processors yields a speed-up of roughly $p^{1/3}$ instead of p.

1.1.2 The dynamic shortest path problem

In the dynamic setting our aim is to build a data structure that can answer queries of the form: *"What is the weight of the shortest path from u to v?"* The goal is to build a data structure such that queries can be answered much more efficiently than performing a single-source computation from u. At the same time we would like our data structure to be dynamic. Whenever the weight of some edge changes (or edges are added or deleted), we would like to be able to modify our data structure quickly. In other words a small change to the input graph should not force us to recompute the entire data structure. Thus the challenge of constructing a dynamic data structure is to satisfy both these requirements simultaneously.

Though there are many algorithms for the dynamic problem [7,27,33] (see also [24]), none of them can simultaneously handle both updates and queries in time that is sublinear in the input size.

1.1.3 Finding shortest paths in parallel and dynamic settings

In this thesis we give efficient exact and approximate algorithms for solving shortest-path problems in parallel and dynamic settings for graphs with nonnegative edge weights.

In the dynamic setting we focus on building a fully dynamic data structure for maintaining approximate all-pairs shortest paths. We say that a path is an ϵ-approximate shortest-path if its length is at most $1+\epsilon$ times the distance between its endpoints. In Chapter 2 we show that if we are willing to settle for approximate answers then substantial improvements are possible in both the query and update times for maintaining shortest paths in a planar graph. In particular, we give a fully dynamic data structure that maintains ϵ-approximate shortest-paths in undirected planar graphs. Both query and update times for maintaining our data structure are sublinear in n (for moderate values of ϵ).

In Chapter 2 we describe a new approximation technique called *path-detour*. The path-detour technique helps in approximately representing the shortest-path information in a set of paths all of which intersect a single separating path. Using this technique we show how to construct an approximate representation for these shortest paths. This representation, called the *crossing substitute* is much sparser than representing each path by a single edge (with the appropriate weight) and forms the basis of our dynamic data structure.

A variation of our technique for representing shortest paths in undirected graphs can also be used to sparsely represent reachability information in directed planar graphs. This variant can be used to develop a fully dynamic data structure for all-pairs reachability in planar digraphs. Our parallel shortest-path algorithm of Chapter 4 relies on a directed version of the path-detour technique.

In Chapters 3, 4, and 5 we consider the shortest-path problem in the parallel setting.

In this setting we show that the problem of finding exact shortest paths can be reduced to a series of approximate computations. Using this we give efficient randomized parallel algorithms for shortest paths in sparse graphs.

In Chapter 3 we give a parallel algorithm for the single-source shortest-path problem for graphs with polynomially bounded edge-weights. Our algorithm uses m processors and runs in $\tilde{O}(\sqrt{n})$ time.[1] Here m and n respectively denote the number of edges and nodes in the graph. Our algorithm works by reducing the exact problem to a series of approximate problems (see Section 3.1) by using a scaling technique similar to the one used by Gabow [34]. We then show how to solve the approximate problem by extending a breadth first search algorithm of Ullman and Yannakakis [96].

In Chapter 4 we investigate the single-source shortest-path problem in planar graphs. We give a parallel randomized algorithm that works in polylogarithmic time and uses a linear number of processors. Thus our algorithm is nearly work-optimal. The algorithm for planar graphs uses the scaling idea of Chapter 3, and a parallel approximation procedure that is essentially a directed version of the path-detour technique of Chapter 2.

The techniques used in our algorithm for planar graphs can also be used to find single-source shortest paths in general directed graphs. The resulting algorithm, although not as efficient as the one in Chapter 3, is considerably faster. It works in polylogarithmic time and uses m^2 processors (where m is the number of edges).

In Chapter 5 we continue our investigation of the parallel shortest-path problem by considering a restricted class of planar graphs called *planar layered digraphs*. For this class of graphs we present a parallel algorithm that runs in $O(\log n \log\log n)$ time using $n/\log\log n$ processors. For the class of *planar layered digraphs* the specialized algorithm performs much better than the algorithm for general planar digraphs presented in Chapter 4. Our algorithm for planar layered digraphs generalizes a similar algorithm presented for grid digraphs in [5].

Our algorithm for planar digraphs presented in Chapter 4 does not depend upon

[1] We use the $\tilde{O}(\cdot)$ notation to elide polylog factors.

planarity and can be generalized to apply also to any minor-closed family of graphs for which a separator decomposition tree using $\tilde{O}(\sqrt{n})$-node separators is given as part of the input. For example we can use our algorithm to find single-source shortest paths in constant-genus graphs if we are given as part of the input a separator decomposition tree of the underlying graph.

The best previously known polylog-time algorithm for single-source shortest paths in planar undirected graphs is that of Pan and Reif [82,83], which does $\tilde{O}(n^{1.5})$ work. Cohen [14] has given an algorithm with similar bounds for the directed case. The previous bounds are no better for breadth-first search, the special case where every edge-length is one.

Our shortest-path algorithm applied to planar graphs in particular has several further applications. It can be used to obtain an efficient polylog-time algorithm for minimum s-t cut in planar graphs, using Johnson's parallel version of an algorithm of Reif [54,55,87]. In the case when s and t belong to the same face, we can also efficiently obtain a maximum s-t flow, using the algorithm of Hassin [48]. Best previously known parallel algorithms for these problems relied on the algorithm of Pan and Reif, and hence required $\tilde{O}(n^{1.5})$ work.

Breadth-first search also has several applications. For example, using our algorithm in conjunction with the method of Miller [76], we obtain the first polylog-time, linear-processor algorithm for finding $O(\sqrt{n})$ separators in planar graphs. The best NC algorithm previously known was that of Gazit and Miller [40], which takes $O(\log^2 n)$ time and requires $O(n^{1+\epsilon})$ processors for any given constant $\epsilon > 0$. Their algorithm can also be used to find separators of size $O(\sqrt{n}\,\text{polylog } n)$ using only n processors. Indeed, we use this in our algorithm. Another application of parallel breadth-first search is in implementing in parallel a method due to Baker [8] for approximation schemes for a variety of problems in planar graphs. Her method involves doing a breadth-first search on a graph derived from the planar input graph. Since the graph still has small separators, our method applies. The second stage of her method involves solving the problem exactly on k-outerplanar graphs. This stage can be done efficiently in parallel using the algorithm of Lagergren [67]

Our shortest-path algorithm can be generalized to handle negative lengths as long as no negative-length edge appears in a directed cycle. Using this algorithm, we can solve the longest-path problem in a dag. By using the technique of Hassin and Johnson [49] we can find the maximum flow in an undirected network where s and t are not necessarily on the same face, in polylog time using a linear number of processors.

1.2 Computing shortest paths in graphs with negative edge weights

In Chapter 6 we investigate the problem of computing shortest paths when the edge-weights are allowed to have negative values. In this chapter we give efficient sequential, parallel, and dynamic algorithms for computing single-source shortest paths in planar digraphs. We also give improved algorithms for several related problems like maximum flow, perfect matching etc.

In the sequential setting we obtain an algorithm that takes time $O(n^{4/3} \log(nL))$, where the lengths are integers greater than $-L$. For general graphs, the best bound known is $O(n^{1/2} m \log L)$ time, due to Goldberg [44] which yields $O(n^{3/2} \log L)$ time on sparse (e.g. planar) graphs. For undirected planar graphs, Pan and Reif [82,83] showed how to achieve $O(n^{3/2})$ time using separators. Cohen [14] showed how to achieve the same bound for directed planar graphs. Previously no algorithm handling negative lengths was known that ran faster than $O(n^{3/2})$. Our algorithm overcomes this apparent barrier, improving the time by a (fractional) polynomial factor when the negative lengths are not too large in magnitude.

For planar graphs, shortest-path computation is closely related to network flow. Given a planar digraph and source and sink nodes s and t respectively on different faces of the graph, Miller and Naor [77] (see also Johnson and Venkatesan [56]) show how to solve the max-flow problem by performing a single-source shortest-path computation in a graph with negative lengths. They also use the same approach in solving the following problem: Given multiple sources and multiple sinks, each with a specified supply or demand, find a feasible flow in the network. Bipartite perfect matching in a planar graph, for example, can be formulated this way. The previous best algorithms for all of these problems required $\Omega(n^{3/2})$ time. Our shortest-path algorithm thus yields polynomial-factor improvements for these problems. In particular, we obtain an $O(n^{4/3} \log n)$-time algorithm for planar bipartite perfect matching. We obtain an $O(n^{4/3} \log n \log C)$-time algorithm for maximum flow, where C is the sum of (integral) capacities. Recently Weihe [100] has given an $O(n \log n)$-time algorithm for computing max-flow in a directed planar graph. His bounds improve upon our bounds for computing the max-flow and the related problem of bipartite perfect matching. However, his algorithm relies on the use of dynamic trees [89] while our algorithm does not use any complicated data structures. Thus, our algorithm may perform better in practice.

The approach used in obtaining the shortest-path algorithm for arbitrary lengths also enables us to obtain parallel and dynamic algorithms for this problem and for the related problems of max-flow and matching.

1.3 Background and overview

A word about terminology in order to forestall confusion. For a graph G, we use $V(G)$ to denote the node-set of G and $E(G)$ to denote the edge-set of G. For any set X we use $|X|$ to denote the number of elements in X. We use $|G|$ as a shortahand for $|E(G)|$.

Unless otherwise stated, when we speak of the *length* of a path, we mean the sum of the lengths of its edges (rather than the number of its edges). Similarly, *distance* is measured according to edge-lengths. The *size* of a path, on the other hand, shall signify the number of edges in the path, and the *minimum path-size* is the shortest distance measured in number of edges traversed. An *r-edge path* is one with at most r edges. Unless otherwise stated, n and m denote, respectively the number of nodes and the number of edges in the graph under consideration.

1.3.1 Models of computation

In sequential and dynamic settings we use a standard random-access-memory (RAM) model of computation [2]. In the parallel setting we use the PRAM [60] or the parallel random-access-memory model of computation. Different algorithms use different PRAM models. However, unless otherwise stated the parallel algorithms presented assume an ARBITRARY CRCW PRAM as the underlying model of computation. In an ARBITRARY CRCW PRAM multiple processors are allowed to read-from or write-to the same location in one step. When multiple processors attempt to write in the same location an arbitrarily chosen processor succeeds.

For the algorithm presented in Chapter 3 we assume a model of parallel computation called the OR CRCW PRAM [9] in which multiple processors can simultaneously read and write to a shared memory. If multiple processors attempt to write multiple values to a single location, the value written is the bitwise OR of the values. This model enables us to do processor allocation using a randomized constant-time algorithm due to Hagerup and Raman [47] for approximate prefix summation. Their algorithm runs in constant time with probability $1 - 2^{-n^{\delta}}$ for some positive constant δ.

Assuming the slightly weaker ARBITRARY CRCW PRAM, in which an arbitrary processor succeeds in writing, would entail using an $O(\log^* n)$-time algorithm [42] for processor allocation. This change multiplies our time bounds of Theorem 6 by $O(\log^* n)$.

In the rest of the chapter we give a brief overview of some of the techniques that have proved useful in addressing the shortest-path problem in different settings. Some of the techniques are well known while others are new. First we review some basic work on separators for planar graphs and discuss two types of decompositions that have proved useful both for our work and for solving a number of other problems. We then discuss a random-sampling technique due to Ullman and Yannakakis [96] that is at the core of the parallel shortest-path algorithms in Chapters 3 and 4. Finally we give a preview of the path-detour technique: a method for approximating shortest paths that cross a given separating path. This technique, in different guises, forms the foundation for our results on parallel and dynamic computation of shortest paths in Chapters 2 and 4.

1.4 Planar separators, cluster decompositions, and the decomposition tree

1.4.1 Separators

A *separating set*, as the name indicates, is a set of nodes whose removal disconnects the graph.

Definition 1 Given a graph $G = (V, E)$ with n vertices we call a set $X \subset V$ an $f(n)$-separator that δ-splits G if $|X| \leq f(n)$ and the vertices in $V - X$ can be partitioned into sets $\{A_1, \ldots A_k\}$ such that there are no edges between any two sets A_i and A_j and for all A_i we have $|A_i| \leq \delta n$.

Separators can also be used to separate a subset of the nodes of G.

Definition 2 For a graph G and a node-subset S, an *S-balanced node-separator* is a set of nodes whose removal breaks G into two pieces such that each piece contains at most an α fraction of the nodes of S (for some suitable constant $1/2 \leq \alpha < 1$).

Several families of graphs have such separators of size $O(\sqrt{n})$ where n is the number of nodes in the graph. Examples include planar graphs [74], bounded-genus graphs [43], two-dimensional overlap graphs [78], and graphs excluding a fixed graph as a minor [4]. Of these families, for all but the last there are polylog-time parallel algorithms for finding such separators. If one is willing to settle for separators of size $O(\sqrt{n}\,\text{polylog}\ n)$, the algorithm of Gazit and Miller [40] for planar graphs requires only a linear number of processors. Separators of size $O(\sqrt{n})$ can be found for overlap graphs by an algorithm using only a linear number of processors [78].

The concept of a separator has played an important role in the development of algorithms; and starting with the planar separator theorem of Lipton and Tarjan [74] it has led to a number of applications in VLSI [68,69,73,95,98] and numerical analysis [72, 82].

Lipton and Tarjan [74] first proved that all planar graphs have $2\sqrt{2}\sqrt{n}$ separators that 2/3-split the graph. Djidev [21,22,23] reduced the size of the separator (required to 2/3-split the graph) to $\sqrt{6n}$. This result and other extensions to it have paved the way to divide and conquer solutions for many problems in planar graphs. A parallel algorithm for finding a cycle separator for biconnected graphs was given by Miller [76] which uses n processors if the breadth first search tree of the graph is already known. An improved version of the algorithm was given by Gazit and Miller [40], which uses randomization to find a cycle separator with $n^{1+\epsilon}$ processors. A version of their algorithm can be used to find $O(\sqrt{n}\,\text{polylog}\ n)$ separators using n processors. Gazit and Miller [38,39] have also given slower but more work-efficient parallel algorithms for finding $O(\sqrt{n})$ planar separators. Recently Klein [66] has given a more work-efficient (but not polylog-time) algorithm for finding planar separators. Algorithms for finding small separators in more general graphs are given in [4,78,79,80,99].

1.4.2 One-way separators

The separators we have examined so far separate the underlying undirected graph. All they guarantee is that any path from one side to the other must pass through the separator. They do not preclude a path from going back-and-forth many times across the separator. We now describe a new kind of separator called a *one-way separator* that is especially useful in implementing divide-and-conquer solutions to the shortest-path problem. A one-way separator is a separator that cuts the graph in such a manner that any simple directed path in the graph crosses it at most once. In particular, shortest paths cannot go back-and-forth across the separator. Such a separator (as we will see in Chapter 5) makes it much easier to combine the shortest-path solutions in two subgraphs to get the answer in the union.

Definition 3 Let X be a separator of a digraph $G = (V, E)$ that divides $V - X$ into sets $A_1, \ldots A_k$, and let p be a simple directed path in G. We say that p crosses X r

times if there are disjoint subpaths $p_1, \ldots, p_r$ of p such that the end points of each p_i are in different sets A_j and A_l. We call X a *one-way separator* if any directed path p between two vertices in G crosses X at most once. A division of G into one or more pieces is a *one-way* division if the separator X used to divide G is a one-way separator.

Grid digraphs, for instance, have one-way separators of size $O(\sqrt{n})$ that 1/2-split the graph. In fact the shortest-path algorithm in [5] uses such separators to construct a divide-and-conquer solution. In Chapter 5 our main result is that planar layered digraphs also admit small one-way separators. We use these separators to implement a fast and efficient divide-and-conquer solution to the single-source shortest-path problem on planar layered digraphs.

1.4.3 Cluster partitions

Frederickson [29] first used planar separators to construct what we call a *cluster partition* (our terminology is similar to the one used by Galil, Italiano, and Sarnak [37]). Using such a partition he gave a more efficient sequential algorithm for finding single-source shortest paths in planar digraphs.

Definition 4 An *k-cluster partition* of an n-node planar graph G is a partitioning of G into edge-induced subgraphs $G_1, G_2, \cdots, G_j$ such that the following properties hold.

1. Each subgraph G_i contains no more than k edges.

2. For any i, the number of *boundary nodes* in G_i is $O(\sqrt{k})$, where a node is a boundary node if it is in more than one subgraph. A node u is a boundary node if and only if edges of more than one cluster are incident on it. Note that u can be a boundary node in G_i only if it is an end point of some edge in G_i.

3. The total number of pieces j is $O(n/k)$.

Given a cluster partition the *parent region* G_u of a node u is the subgraph G_i that contains u. If u is a boundary node then we arbitrarily set G_u to be one of the subgraphs G_i that contains edges incident to u.

Lemma 1 (Frederickson [29]) *Given an n-node planar graph G a k-cluster partition can be obtained in $O(n \log n)$ time.*

Frederickson [29] introduced the idea of such a partition and showed how to obtain one in $O(n \log n)$ time. A cluster partition is obtained by starting with G and repeatedly dividing it using a planar separator [74] until all the pieces have at most k edges. His algorithm also makes sure that none of the pieces contain too many boundary nodes by redividing pieces when the ratio of boundary nodes to non boundary nodes crosses a threshold.

Definition 5 A *face-restricted* k-cluster partition is a partition in which all of the above properties hold. In addition for each subgraph G_i the boundary nodes are present on a constant number of faces (for a suitable constant f).

In order to construct a face-restricted cluster partition we make use of different separator algorithm by Miller [76]; one that constructs *cycle separators.* A cycle separator X is a separator such that all the nodes of X lie on a simple undirected cycle in the underlying planar graph G. The size of a separator in Miller's algorithm depends the number of edges on the maximum sized face. Therefore we need to keep the size of the faces small.

To build the cluster partition we first two-connect and triangulate G by adding dummy nodes and edges. This is done by placing a zero-weight dummy node in each face and connecting it by dummy edges to all the nodes on the boundary of the face. By applying Miller's algorithm to this graph, we obtain a cycle separator. The set X of non-dummy nodes of the separator need not form a cycle in the original graph. However, it does divide the graph into two pieces such that, in the subgraph G' induced on one piece together with the nodes of X, the nodes of X all lie on the boundary of a single face. As in the previous case we repeatedly use cycle separators to divide the subgraphs until we get pieces of size k. We retriangulate subgraphs when faces get too big so that the pieces are guaranteed to have small cycle separators. Using techniques due to Frederickson [29] we can also make sure that none of the pieces have too many boundary nodes, and in each G_i the boundary nodes are on a constant number of faces. This process takes $O(n \log n)$ time. These dummy nodes and edges are transient and do not play any role in the data structures of Chapter 2. They are only required to find small cycle separators using Miller's algorithm. We therefore have the following lemma.

Lemma 2 *Given an n-node planar graph G, a face-restricted k-cluster partition can be obtained in $O(n \log n)$ time.*

Frederickson [28,30] used the idea of partitioning a graph into clusters to develop dynamic data structures for maintaining minimum spanning trees, connected components, and two-edge connected components. Galil and Italiano [36] used a k-cluster partition to develop a fully dynamic data structure for two- and three-connectivity in planar graphs. Later Galil, Italiano, and Sarnak [37] used such a decomposition for designing a dynamic planarity-testing algorithm.

The basic idea behind these dynamic data structures is the following.

> We first divide the underlying graph into suitably sized pieces. We then precompute partial solutions to the problem on each of the pieces. Whenever a query is asked the partial solutions are quickly combined to give the answer to the query, resulting in a fast query-response time. When the underlying graph changes, only some of the clusters are affected. We therefore need only recompute our partial solution in a constant number of subgraphs (instead of recomputing everything). Thus our update time for the data structure remains small.

In Chapter 2 we use the face-restricted k-cluster partition to develop a fully dynamic data structure for maintaining approximate all-pairs shortest paths in undirected planar graphs with nonnegative edge weights. We also show how to use such a partition to build a dynamic data structure for all-pairs reachability in planar digraphs.

In Chapter 6 we will show how to use a k-cluster partition to develop faster sequential, parallel, and dynamic algorithms for shortest paths in planar digraphs that have both negative and nonnegative edges.

1.4.4 The decomposition tree

A cluster partition is a two level decomposition of the graph wherein the graph is decomposed into $O(n/k)$ pieces. We now describe a recursive version of this decomposition where we repeatedly decompose the graph using separators until we reach pieces of constant size. Such a recursive decomposition has proved useful in constructing a divide-and-conquer solution to many problems on planar graphs [14,72,82].

Let G be a constant-degree graph all of whose subgraphs have $O(\sqrt{n})$ separators. By repeatedly finding separators, one obtains a decomposition of G. Following tradition, we find it useful to represent the structure of the decomposition by means of a rooted tree, the decomposition tree. We refer to *vertices* of the decomposition tree instead of *nodes* in order to avoid confusion with the nodes of G. Each vertex v of the decomposition tree is labeled with a node-induced subgraph $G(v)$ of G, called the *pertinent graph* of v, and a subset $S(v)$ of the nodes of $G(v)$, called the *splitting set* of v. Essentially $S(v)$ consists of the nodes used to separate $G(v)$, together with the nodes of $G(v)$ used in separators belonging to ancestors of v.

We define the decomposition tree recursively as follows. Let S be a subset of G's nodes such that $|S| = \tilde{O}(\sqrt{n})$. If G contains only one node, then the decomposition tree for G and S consists of a single vertex v labeled with pertinent graph G and splitting set S. Otherwise, let X be a separator for G that breaks G into up to three pieces such that each piece contains at most an α fraction of the nodes of G and S. Such a separator can be obtained by first finding a $V(G)$-balanced separator and then finding an S-balanced separator in the piece that contains more than half the nodes of S. Let G_1, G_2, G_3 be the pieces with the separator nodes X added to each. For $i = 1, 2, 3$, let $S_i = (X \cup S) \cap V(G_i)$. Let T_i be the decomposition tree for G_i and S_i. The decomposition tree for G and S is obtained from $T_1 \cup T_2 \cup T_3$ by adding a new vertex v labeled with G and with splitting set $S \cup X$, and having as its three children the roots of T_1, T_2, and T_3.

The point of the splitting sets is as follows. Let G_0 be the pertinent graph of the root. A path in G_0 cannot "escape" a pertinent graph $G(v)$ without first passing through a node of the splitting set of v's parent $p(v)$. More formally, $S(p(v))$ separates $G(v) - S(p(v))$ from the rest of G_0. We call this the *splitting property* of the decomposition tree with respect to the graphs $G(\cdot)$.

A vertex v in the tree is said to be in level i if the distance of v from the root of the tree is i. Let $T(G)$ be the decomposition tree for G and the empty set. Let $n = |V(G)|$. It is not hard to show that $T(G)$ has the following properties:

1. The number of levels in the decomposition tree is $O(\log n)$, and

2. For each level i, the sum of the sizes of the pertinent graphs at that level is $O(n)$, and the sum of the squares of the sizes of the splitting sets is $\tilde{O}(n)$.

Recently in joint work with Klein, Rao, and Rauch [62] we have given a linear-time algorithm for finding single-source shortest paths in planar digraphs. Our algorithm uses a recursive decomposition tree similar to the one described above.

Our parallel algorithm for single-source shortest paths in Chapter 4 uses the decomposition tree described here. It starts by finding approximate shortest paths (within each piece) between boundary nodes at the bottommost level of the decomposition tree. The algorithm then works its way up the decomposition tree. At the level i vertex v_i in the decomposition tree we use the approximate shortest-path information from the level $i-1$ children of v_i to find approximate shortest paths in $G(v_i)$ the pertinent graph of v_i. The exact problem is solved by repeatedly finding approximations and using a scaling technique similar to the one by Gabow [34] (see Section 3.1).

1.5 The Ullman-Yannakakis random-sampling technique

We now describe a random-sampling technique due to Ullman and Yannakakis [96] that is at the core of our parallel algorithms for shortest paths (see Chapters 3 and 4).

Definition 6 Let G be an n-node graph and let S be a subset of the nodes in G that are marked as *distinguished*. Given an integer $0 < k \leq n$, a path $\pi \in G$ is said to have a *k-gap* with respect to S if it contains a subpath of size at least k without any distinguished nodes. We will not specifically mention the set S whenever there is no ambiguity.

Theorem 1 (The sampling theorem of Ullman and Yannakakis) *Let G be an n-node graph and let Π be a collection of paths in G such that $|\Pi| = O(n^{\alpha})$ for some constant α. Given a nonnegative integer $0 < k \leq n$ and an accuracy parameter $c > 0$, if we select a set S of $\frac{\beta n}{k} \log n$ nodes uniformly at random from G then with probability $1 - n^{-c}$, no path in Π has a k-gap with respect to S. The value of β depends only on c and α, and is independent of n.*

Ullman and Yannakakis [96] used this idea to devise an elegant parallel algorithm for computing single-source reachability and breadth-first search. The basic idea is the following.

> Let us fix in advance the set Π of paths from the source s to all the other nodes contained in some breadth-first-search tree. A sequential search from the source node to find these paths would require a long time because we might have to traverse long paths. Thus such a search could require $\Omega(n)$ time. To get a faster algorithm we instead, randomly select $O(\frac{n}{k} \log n)$ nodes and mark them as distinguished. We also mark the source node s as distinguished. We then simultaneously search from all the distinguished nodes for k-stages; i.e. we only look for paths with at most k edges. Such a search is called *k-limited* (see Chapter 3). We use the information gathered from these searches to compute the BFS-tree from s. Since we look for k-limited paths from the distinguished nodes, our search time is roughly proportional to k instead of n. By suitably selecting k we can design a fast parallel algorithm that is much more work-efficient than using parallel matrix multiplication.

By Theorem 1 *with high probability* (probability exceeding $1 - n^{-c}$ for some constant c) none of the paths in Π contains a k-gap with respect to the distinguished nodes. Therefore, with high probability the information contained in the k-limited search trees of the selected nodes is enough to derive the search tree of s. Thus the algorithm will output the correct answer.

For more details on the Ullman-Yannakakis algorithm see Section 3.2. Our parallel shortest-path algorithms of Chapters 3 and 4 use this random-sampling technique at various stages so that long paths can be found quickly in parallel by simultaneously searching from some randomly selected nodes. The algorithm in Chapter 3 uses the random-sampling technique exactly as used by the Ullman-Yannkakis algorithm. In Chapter 3 we first show how to extend their algorithm for computing approximate shortest-paths. We then use this in conjunction with a scaling technique to find exact shortest paths. In Chapter 4 we use this random-sampling technique to construct graphs that approximate the shortest-path distances by paths containing a polylogarithmic number of edges. Cohen [13,16] has also used this random-sampling technique to find approximate shortest paths in undirected graphs.

1.6 Path-detour: A new technique for finding and representing paths that cross a separating path

In this section we describe a new technique called *path-detour* that is the basis for our dynamic and parallel shortest-path algorithms of Chapters 2 and 4. A simpler version of this technique is also used in building a fully dynamic data structure for all-pairs reachability in planar digraphs. The basic idea is as follows.

Let G be an undirected graph with nonnegative edge weights. Let $d > 0$ be a distance parameter and $\epsilon > 0$ an error parameter. Let Π be a collection of paths in G such that all of them have length at least d. Also, let P be a path of length at most $\frac{\epsilon}{2}d$ that intersects all the paths in Π. The aim of the path-detour technique is to find a sparse representation for the paths in Π such that given any path $\pi \in \Pi$ the representation should have a path between the same endpoints with length at most $1+\epsilon$ times $\ell(\pi)$. In other words we are interested in a sparse $(1+\epsilon)$-factor approximation of the paths in Π.

One way to represent the paths in Π is to represent each path π by an edge between

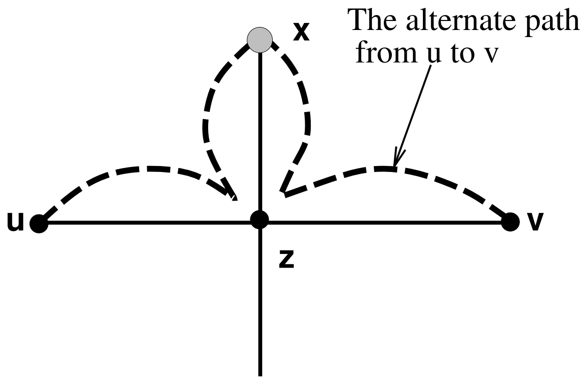

Figure 1.1: Approximating a path that passes through P

the same endpoints with length equal to $\ell(\pi)$. This preserves all the distance information in Π but is very dense since there is one edge for every path in Π. In order to find a sparser representation we construct a *substitute graph* that approximately preserves the distance information in Π.

Let x be one of the end-nodes of P. We call x the *detour node* of P. Let $\pi \in \Pi$ be a path that crosses P at a point z (see Figure 1.1). In order to sparsely represent π we consider the alternate path π' that goes via the detour node x on P (see Figure 1.1). The length of the alternate path is at most $\ell(\pi) + 2\ell(P) = \ell(\pi) + \epsilon d$. Since all the paths in Π have length $\geq d$ we see that $\ell(\pi') \leq (1+\epsilon)d$. Let u and v be the endpoints of π. In the substitute graph we represent π by two edges ux and vy such that $\ell(ux) + \ell(xy) \leq \ell(\pi')$.

In order to approximate all the paths in Π we find the shortest paths from x to all the endpoints u in Π. We then construct a substitute which contains x as a center of a *star* and edges xu with lengths corresponding to the shortest path from x to u.[2] If the number of endpoints in Π is k then the size of the substitute graph is k, even though there could be as many as k^2 paths in Π.

We can use a similar technique to represent the reachability information in a collection Π of directed paths all of which intersect a separating path P. Our substitute consists of a modified copy of P along with edges to and from the endpoints of Π to P.

[2] A star centered at x is simply a depth-one tree with x at its root.

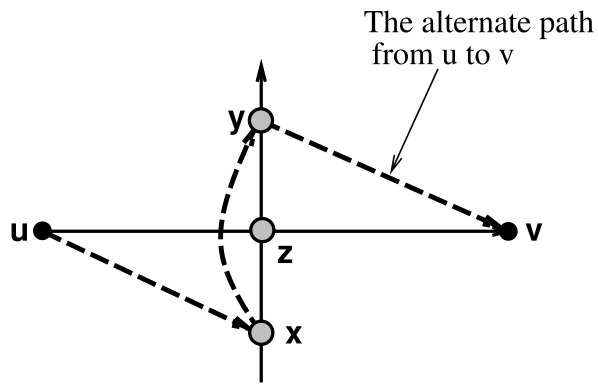

Figure 1.2: Approximating a path that passes through P

Let $\pi \in \Pi$ be a path from u to v that intersects P at z. See Figure 1.2 for an example. The substitute contains edges ux and yv such that z occurs between x and y on P. Thus the alternate path goes from u to v via the x-to-y subpath of P and represents the same reachability information as π. The edges ux and vy are constructed by finding the lowest point x on P that can be reached by u and the highest point y on P that can reach v. In Section 2.2 we show that the substitute constructed in this fashion has at most $2k$ edges thus resulting in a sparse representation for the reachability information in Π.

The path-detour technique helps us in constructing sparse substitutes for reachability and approximate shortest paths and thus forms the basis for the dynamic data structures in Chapter 2.

We can also use the path-detour technique for quickly computing, in parallel, paths of length roughly d that intersect a separating path P. As earlier, suppose we have a collection Π of paths (with lengths $\geq d$) that intersect a separating path P of length $\leq \epsilon d/2$. Further, suppose that all the paths in Π have $\leq k$ edges.

If we are dealing with an undirected graph we can quickly find a representation that approximates the paths of Π accurately. This is done by running the Bellman-Ford shortest-path algorithm from the detour node x of P for k stages. As observed by Cohen [14], running the Bellman-Ford algorithm for k stages is sufficient to find paths that contain $\leq k$ edges.

In a directed setting the approximation technique outlined in Figure 1.1 is invalid because the edges of P go only in one direction. In Chapter 4 we show that path-detour is possible even in a directed setting. We also show how to do such an approximation quickly in parallel. In particular, Lemmas 9 and 11 of Section 4.1 show how to find a good representation for intersecting paths quickly in parallel. The parallel shortest-path algorithm of Chapter 4 uses the path-detour idea to find paths that have few edges and uses the random sampling idea of Theorem 1 to construct longer paths from these short paths. This combination of random sampling and parallel path-detour forms the basis for the two main procedures COMPACT and SMALLPATH of Chapter 4.

The remainder of the thesis is organized as follows: In Chapter 2 we give our dynamic data structures for approximate shortest paths and reachability. In Chapters 3, 4, and 5 we explore the parallel shortest-path problem in various settings, and in Chapter 6 we investigate the complexity of the shortest-path problem when edges are allowed to have negative integral weights.

Chapter 2

Fully dynamic data structures for shortest paths and reachability in planar graphs

In this chapter we explore the dynamic shortest-path problem for planar graphs with nonnegative edge weights. We also explore the related problem of building a dynamic data structure that maintains all-pairs reachability in a planar digraph.

An algorithm for a given graph problem is said to be *dynamic* if it can maintain the solution to the problem as the graph undergoes changes. These changes could be additions or deletions of edges, or a change in the cost of some edge (if applicable). In such a setting an *update* denotes an incremental change to the input and a *query* is a request for some information about the current solution. We expect the dynamic algorithm to handle both queries and updates quickly, i.e., in time that is substantially less than it would take to solve the problem from scratch every time the input changes. An algorithm is said to be *fully dynamic* if it supports both additions and deletions of edges while it is said to be *semi dynamic* if it supports only one of them.

Unfortunately designing fully dynamic algorithms seems to be considerably harder than designing their sequential counterparts, and very few graph problems have fully dynamic solutions. See [28,30,36,37] for fully dynamic data structures to various graph problems. See [81] for a complexity theoretic approach to dynamic computation.

As we discussed in Chapter 1 the shortest-path problem is not very well understood in the dynamic realm. Though there are many algorithms for the dynamic problem [7, 24,27,33], none of them can simultaneously handle both updates and queries in time that is sublinear in the input size.

We say that a path is an ϵ-approximate shortest-path if its length is at most $1 + \epsilon$ times the distance between its endpoints. In this chapter we show that if we are willing to settle for approximate answers then substantial improvements are possible in both the query and update times for maintaining shortest paths in a planar graph. In particular, we give a fully dynamic data structure that maintains ϵ-approximate shortest paths. Both query and update times for maintaining our data structure are sublinear in n (for moderate values of ϵ). For maintaining all-pairs shortest paths in an undirected planar graph we have the following bounds.

Theorem 2 *For any $0 < \epsilon \leq 1$, there exists a fully dynamic data structure to maintain ϵ-approximate all-pairs shortest-path information in planar undirected graphs with nonnegative edge lengths. The time per operation is $O(\epsilon^{-1} n^{2/3} \log^2 n \log D)$ where D is the sum of the lengths of all the edges and n is the number of nodes in the graph. The time for queries, edge-deletion, and changing lengths is worst-case, while the time for adding edges is amortized.*

We also consider the related problem of maintaining all-pairs reachability in a planar digraph. The reachability problem on a directed graph G consists of answering queries of the type "Is there a directed path from u to v in G?" The digraph reachability problem is a fundamental research problem and has both theoretical and practical applications. In the complexity-theoretic setting this problem is complete for the class NLOGSPACE while on the practical side the related problem of computing the transitive closure of a digraph is an important subroutine for answering database queries.

Given a digraph G with n nodes and m edges, a reachability query between two nodes u and v can be easily answered in $O(m)$ time by doing a depth-first search from

u (to determine if there is a directed path from u to v). However, in the dynamic realm the problem seems much harder. The best previously known dynamic results for general digraphs are semi-dynamic data structures with $O(n^2)$ space, constant query time, and $O(n)$ update time. Data structures that support insertions only are given in [12,50,84]. Data structures that are restricted to acyclic digraphs and support only deletions can be found in [12,51].

We can obtain better results if we restrict ourselves to specific classes of graphs. For series-parallel digraphs, Italiano, Marchetti-Spaccamela, and Nanni [52] gave an $O(n)$-space data structure that handles both queries and updates in $O(\log n)$ time. For the class of planar st-graphs (acyclic planar digraphs with one source and one sink both of which are on the same face) Tamassia and Preparata [92] give an $O(n)$-space fully dynamic data structure that supports both updates and queries in $O(\log n)$ time. However, their algorithm is embedding-specific; i.e., only those changes that are consistent with the current embedding of the graph are allowed. Tamassia and Tollis [93] have extended the results of [92] to get a fully dynamic data structure for answering reachability queries in spherical st-graphs and planar digraphs with a single source and a single sink. Here too the updates are restricted to be embedding specific. To our knowledge there is no previous work that handles general planar digraphs even if updates are restricted to be embedding-specific.

The results in [92] and [93] are based on the well known result in dimension theory which (in graph theoretic terms) states: "Given an n-node planar directed acyclic graph G with one source and one sink, we can find two orderings $<_L$ and $<_R$ of the nodes in G such that a node v is reachable from another node u in G if and only if v occurs after u in both the orderings [11,58]." Unfortunately it seems unlikely that these methods can be extended to all planar digraphs, since Kelly [61] has shown that for general planar digraphs we might need a large number of orderings to represent all-pairs reachability information as an intersection of these orderings.

In this chapter we give a fully dynamic data structure to maintain the all-pairs reachability information in planar digraphs. Unlike previous algorithms our algorithm handles the entire class of planar digraphs and at the same time allows all updates as long as the graph still remains planar. The time bounds for maintaining the all-pairs reachability are as follows.

Theorem 3 *Given an n-node planar directed graph G, a fully dynamic $O(n)$ space data structure that maintains the all-pairs reachability information in G can be constructed in $O(n \log n)$ time. Any reachability query in G can be answered in $O(n^{2/3} \log n)$ time using this data structure. Updates to the underlying digraph G, like addition and deletion of edges and nodes, can also be reflected in the data structure within the same time bounds. The time required to modify the data structure for delete operations is worst-case while the time for additions is amortized.*

Our dynamic data structures for shortest paths and reachability are based upon a novel technique for compactly representing approximate all-pairs shortest paths or all-pairs reachability information, among a set of k *selected* nodes, by a sparse substitute graph. The substitute graph that approximates shortest paths has the following properties.

- Each edge uv in the substitute graph corresponds to a path π from u to v in G.
- Each shortest path between selected nodes in G is approximated to within a $1+\epsilon$ factor by a two-edge path in the substitute graph.

The substitute graph that represents all-pairs reachability among the selected nodes has the following properties.

- Each edge uv in the substitute graph corresponds to a directed path π from u to v in G.
- For any pair of selected nodes u, v such that v is reachable from u in G, there exists a directed path from u to v in the substitute graph.

The size of the substitute graph depends upon the location of the selected nodes in G but is independent of the number of nodes in G. When the selected nodes lie on the boundaries of a constant number of faces we obtain the following bounds for representing approximate shortest paths and all-pairs reachability between them.

Theorem 4 (Face-boundary shortest-path substitute) *Let G be an n-node planar undirected graph with nonnegative edge weights that sum up to at most D. Also, let S be a set of k selected nodes in G all of which lie on the boundaries of a constant number of faces. Given G and S we can construct a sparse substitute graph that approximately represents the all-pairs shortest paths between the selected nodes. The substitute graph has $O(\epsilon^{-1} k \log k \log D)$ edges and approximates the all-selected-node-pair shortest paths in G to within a $1+\epsilon$ factor. Furthermore, the substitute graph can be constructed in $O(\epsilon^{-1} n \log n \log D)$ time.*

Theorem 5 (Face-boundary reachability substitute) *Let G be a planar directed graph with n nodes and let S be a set of k selected nodes in G all of which lie on the boundaries of a constant number of faces. Given G and S we can construct a substitute graph that represents the all-pairs reachability information between the selected nodes. The substitute graph has $O(k \log k)$ edges and can be constructed in $O(n \log n)$ time.*

To construct both these substitutes we use a technique called *path-detour* (introduced in Section 1.4), to represent the information present in a collection of paths all of which cross a single path. In the shortest-path scenario this helps in getting a sparse $(1+\epsilon)$-approximate representation for the original collection of paths. In representing reachability, the path-detour technique gives us a sparse representation that contains the same reachability information as the original collection of paths.

Sparse substitute graphs have been used earlier in [36], [37], and [26] for maintaining various properties in undirected planar graphs. However, our substitute for representing all-pairs reachability is the first such substitute for directed planar graphs. Furthermore, while constructing the substitute, we also need to concern ourselves with the distribution of the selected nodes. This implies that the partitioning of G into clusters has to be done more carefully than was the case before. To build our dynamic data structures we use the face-restricted cluster partition described in Section 1.4.

The rest of the chapter is organized as follows. In Section 2.1 we describe the construction of the *face-boundary shortest-path substitute* for representing all-pairs shortest paths among a set of selected nodes. In Section 2.2 we show how to construct the *face-boundary reachability substitute* for representing all-pairs reachability among selected nodes. Finally, in Section 2.3 we describe the dynamic data structures for maintaining shortest paths and all-pairs reachability.

2.1 Face-boundary shortest-path substitutes: sparse representation of approximate shortest paths

In this section we address the issue of constructing a face-boundary shortest-path substitute to approximately represent the all-pairs shortest paths in G among a set Q of $O(\sqrt{n})$ selected nodes. We will assume that the selected nodes are distributed over a constant number of faces.

In Subsection 2.1.1 we detail the path-detour technique introduced in Section 1.4. Path-detour is the key to the construction of our substitutes. In Subsection 2.1.2 we give a simple divide-and-conquer procedure that repeatedly uses the path-detour technique to construct a face-boundary shortest-path substitute.

2.1.1 Path-detour: A basic sparsification technique

Let G be a planar undirected graph with nonnegative edge weights and let ϵ be an error parameter and d a distance parameter. Also, let P be a path in G of length $O(d)$. Let Q be a set of k selected nodes that lie on the boundary of a constant number of faces. In this subsection we show how to sparsely represent selected node-pair shortest paths that intersect P, and are between d and $2d$ in length. In particular, we show that there exists a substitute graph with $O(\epsilon^{-1}k)$ edges that approximates these shortest-paths to within a $1+\epsilon$ factor. We call this substitute a *crossing substitute.*

To construct a crossing substitute we first divide P into $O(\epsilon^{-1})$ node-disjoint segments of length at most $\epsilon d/2$ each. The first node x_i in each segment s_i is called

```
procedure Path-detour
    [Dividing the separating paths]
        for each path ρ ∈ Z do
        1. Partition it into O(ε^{-1}) segments s_1, s_2, ..., s_k of length at most εd/2.
        2. let x_i be the first node of segment s_i.
        3. for 1 ≤ i ≤ k do

            [Sparsifying paths that cross segment s_i]
            a. Compute shortest paths from x_i.
            b. For each boundary node u add the edge ux_i (to the substitute),
                   with the shortest-path length found in Step a.
               end [ Sparsification]
end [procedure Path-detour]
```

Figure 2.1: The Path-detour technique

the *segment node* of that segment. To approximate the paths that cross segment s_i we use the approximation idea introduced in Section 1.6 (see Figure 1.1). To do this we perform single-source shortest-path computations in G from each of the $O(\epsilon^{-1})$ segment nodes. Our crossing substitute consists of a collection of stars, one for each segment, as discussed in Section 1.6. The star for segment s_i has x_i as its center and the selected nodes as its leaves. The edge between a selected node u and x_i is labeled with the length of the shortest path from x_i to u. A pseudocode version of the basic sparsification procedure is given in Figure 2.1.

As discussed in Section 1.6, these stars approximately represent d-length shortest paths between selected nodes that intersect P. For each segment s_i, a path π from u to v that intersects s_i is approximated in our substitute by the two-edge path ux_i, x_iv that detours via the detour node x_i of s_i.

To bound the number of edges in our substitute we note that it consists of $O(\epsilon^{-1})$ stars each of which has at most k edges, giving us a total of $O(\epsilon^{-1}k)$ edges. To bound the time required to compute the crossing substitute we note that the most expensive step is the single-source shortest-path computations from the segment nodes. Since there are $O(\epsilon^{-1})$ segment nodes, the entire computation can be carried out in $O(\epsilon^{-1}n)$ time using the shortest-path algorithm of [62].

2.1.2 Face-boundary shortest-path substitutes

Let Q be a set of k *selected* nodes in G that lie on a constant number of faces, and let $\epsilon > 0$ be an error parameter. In this section we show how to construct a face-boundary shortest-path substitute that approximates the all-pairs shortest paths between selected nodes to within a $1+\epsilon$ factor. Here we describe a procedure for sparsification when all the selected nodes lie on the boundary of a single face f. The case of multiple faces is similar. Our sparsification algorithm uses a divide-and-conquer mechanism and makes use of the path-detour technique from subsection 2.1.1.

Path-detour works best with paths that are all roughly of the same length. To accommodate this, we use a *grouping* technique (see, e.g., [63,65]). We consider the selected-node-pair shortest paths in $\log D$ different groups and for each group we build a different substitute. The ith substitute approximates selected-node-pair shortest paths in the range $[d, 2d]$, where $d = 2^i$. These $\log D$ substitutes are unioned to get a substitute that sparsely represents paths of all lengths.

To sparsely represent selected-node-pair shortest paths with lengths in the range $[d, 2d]$ we use the divide-and-conquer procedure described below. A pseudocode version of the sparsification procedure is give in Figure 2.2.

1. **Finding separating paths:** We use a procedure called **separate** that gives a separating set Z consisting of one or two paths of length at most $4d$. These paths divide Q into subsets M and O of size at most $3|Q|/4$ each such that any path of length at most $4d$ between them intersects some path in Z (see Figure 2.3).

2. **Sparsifying intersecting paths:** To sparsely represent shortest paths that cross the separating paths we use path-detour (see Subsection 2.1.1).

3. **Dividing G for the recursion:** For each node x in G we determine whether there is a path of length at most $2d$ between x and some node in M (respectively O) that does not intersect any of the separating paths in Z. If there is such a path then we place x in V_1 (respectively V_2). We construct G_1 and G_2 by taking node-induced subgraphs of V_1 and V_2 respectively.

Note that no node can be in both V_1 and V_2 at the same time. Otherwise, we would get a path of length at most $4d$ between M and O that does not intersect any separating path in Z.

To find the nodes of G that go into V_1 we combine all the nodes of M into a single supernode s and find the nodes of G that are within a distance $2d$ from s in $G - Z$. The set V_2 is determined similarly.

4. **Recursive computation:** We recursively compute sparse substitutes for $[d, 2d]$-shortest paths in G_1 and G_2 with both endpoints in M and O respectively. The two recursive substitutes are unioned along with the crossing substitute found in step 2 to get the face-boundary shortest-path substitute for all the selected nodes.

Before we analyze our procedure for sparsification we describe procedure **separate**. Consider a division of Q into four equal-sized contiguous subsets A_1, A_2, A_3, and A_4 as shown in Figure 2.3. If there is a path of length at most $4d$ from A_1 to A_3 or A_2 to A_4 then we can use it to divide Q into subsets M and O as shown in Figure 2.3. Note that the nodes of M are topologically separated from those of O by the separator. Thus every path between M and O crosses the separating path.

When there is no single path of the desired length that divides Q into roughly equal subsets we can find a two-path separator as follows:

1. We first number the nodes clockwise along the face f starting at the first node of A_1.

2. We then set a_1 to be the lowest-numbered node in A_1 that has a path of length $4d$ or less to some node b_1 in A_2. The path π_1 between a_1 and b_1 is the first path in our separator. If there are no such paths between nodes of A_1 and those of A_2 then a_1 and b_1 are set to be the last node in A_1 and π_1 is set to ϕ.

3. Similarly, we set a_2 to be the lowest-numbered node in A_3 that has a path of length $4d$ or less to some node b_2 in A_4. The path π_2 between them is the second

```
procedure sparsify(G, Q)
[Initializing the global substitute sub]
    sub ← φ;
    [Grouping the paths]
        for each value of i in [0, log D] do
        [Group i sparsifies the paths in the range [2^i, 2^{i+1}]]
            let d ← 2^i;

    [Sparsifying the paths in group i]

        I. [Finding a separator]
            Use separate to find a set Z of ≤ 2 O(d)-length paths such that
            Z divides Q into two sets M and O of roughly equal size
            and cuts all paths of length ≤ 4d between M and O;

        II. [Sparsifying paths that cross the separator]
            Use Path-detour to construct a substitute sub_c for the paths
            that have one endpoint in each of M and O;

        III. [Dividing G for the recursion]
        1. let V_1 = M and V_2 = O;
        2. for each node x in G do
                if there is a path of length ≤ 2d from M to x in G − Z
                then place x in the set V_1
                else place it in V_2;
        3. for i = 1, 2 construct G_i by forming the vertex-induced subgraph
            of G induced by the vertices in V_i;

        IV. [Recursive computation of face-boundary shortest-path substitutes]
            let sub_m = sparsify(G_1, M) and
            sub_o = sparsify(G_2, O);

        V. Union sub_c, sub_m, and sub_o to get sub_i, the substitute for group i;

        VI. [Adding to the global substitute]
            sub ← sub ∪ sub_i;
    end [for]
end [procedure sparsify]
```

Figure 2.2: Procedure **sparsify**

path of our separator. As before, if no such path exists a_2 and b_2 are set to be the last node in A_3 and π_2 is set to ϕ.

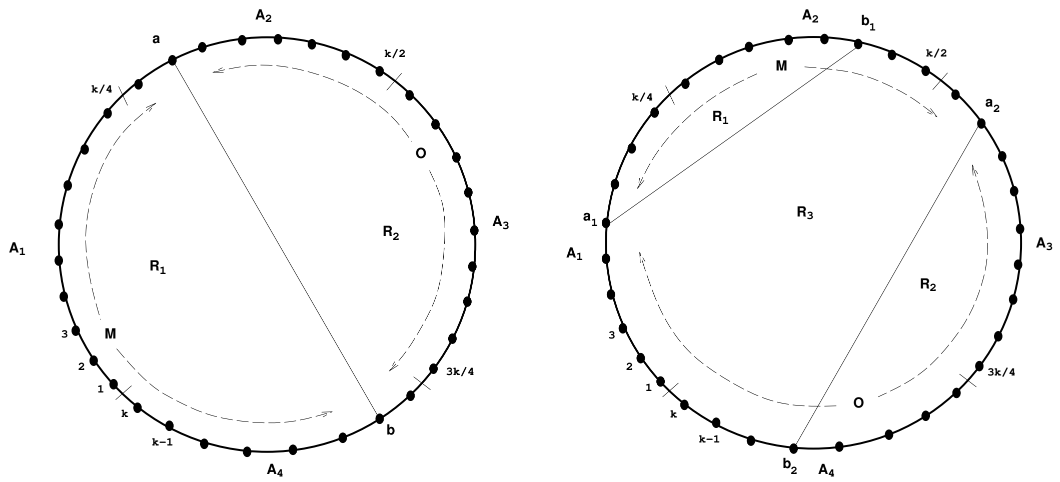

Figure 2.3: Using one or two paths to cut the boundary

4. The nodes in Q are divided into two sets M and O. M contains the nodes of Q from a_1 to a_2 in the clockwise order, including a_1 but excluding a_2, while O contains the rest.

As shown in Figure 2.3, the exterior of face f is topologically separated into three regions R_1, R_2, and R_3. Any path between M and O that has one of its endpoints in R_1 or R_2 is forced to cross one of the separating paths. Also, by the minimality of a_i there is no path from M to O of length at most $4d$ that lies entirely in R_3 and does not intersect either of the separating paths.

To find the lowest-numbered node $a_1 \in A_1$ that has a path of length at most $4d$ to some node in A_2 we first coalesce the nodes of A_2 into a single supernode s. We then find single-source shortest paths from s to all the nodes in A_1. The node a_1 is then set to be the lowest-numbered node that has a path π_1 of length less than or equal to $4d$ from s. The path π_1 is the first path in our separator. Similarly we can find the lowest-numbered node $a_2 \in A_3$, and the corresponding path π_2 from a_2 to some node in A_4..

The time spent by procedure **separate** is dominated by the time required to perform a single-source shortest-path computation from the supernode s. Coalescing the nodes of A_2 into s preserves the planarity of G since all the nodes of A_2 are on a single face. We can therefore use the algorithm of [62] to find shortest paths from s. Thus we can implement procedure **separate** in $O(n)$ time.

To bound the size of the substitute we note that the substitute for shortest paths in any range $[d, 2d]$ is constructed in $O(\log k)$ stages. Furthermore, at each level in the recursion the sum of the sizes of all crossing substitutes is $O(\epsilon^{-1}k)$. Hence the total size of the substitute for group d is $O(\epsilon^{-1}k \log k)$. This combined with the fact there are $O(\log D)$ groups implies that the size of the substitute is $O(\epsilon^{-1}k \log k \log D)$.

To bound the running time of our sparsification procedure we note that the time required for constructing the crossing substitute in step 2 is $O(\epsilon^{-1}n)$. The time needed to find separating paths is $O(n)$ and the time required for dividing G is $O(n)$. This in conjunction with the fact that the recursion depth is $O(\log k)$ and the fact that there are $\log D$ groups gives us the bounds for Theorem 4.

2.2 Face-boundary reachability substitutes: sparse representation for all-pairs reachability

Let G be a planar directed graph and Q be a set of k selected nodes that lie on the boundaries of a constant number of faces. In this section we address the issue of constructing a face-boundary reachability substitute to represent the all-selected-node-pair reachability information in G. Here we show how to construct a substitute when all the selected nodes lie on the boundary of a single face f. The case of multiple faces is similar.

To construct the substitute we follow the same simple divide-and-conquer approach used to find the face-boundary shortest-path substitute in Section 2.1. The strategy is as follows.

- As before, we first divide the selected node set Q into two roughly equal halves M and O (see Figure 2.3).

- We now find a sparse substitute that represents reachability information from the nodes in M to the nodes in O and vice-versa. This step is a variant of the path-detour technique used in Subsection 2.1.1 for finding a crossing substitute.

- Again, we use recursion to find sparse substitutes for all-pairs reachability among nodes in M (respectively nodes in O).
- We then union the three substitutes to get the final face-boundary reachability substitute.

We now give the details of each of these steps.

1. **Finding separating paths:** Similar to the shortest-path setting we use a procedure called **separate2** that gives a set Z of one or two paths in G that divides the selected nodes in Q into two sets M and O. These sets are constructed such that neither of them has more than three-fourth of the nodes in Q and any directed path from M to O (or from O to M) intersects one of the paths in Z (see Figure 2.3). Procedure **separate2** is similar to procedure **separate** and is described later.

2. **Sparsifying intersecting paths:** To sparsely represent reachability information between the nodes of M and O we proceed in the following manner. For each path π in Z we perform the following steps.

    (a) We first number the nodes on π in increasing order starting from the source.

    (b) For each node $u \in Q$ we then find the lowest-numbered node x on π such that there is a directed path from u to x in G. We label node x to be a *marked node* and call it the *entry point* of the node u. To find the entry points of all the nodes in Q we first add a temporary auxiliary node s and construct directed edges from all the nodes on π to s. We then perform an ordered depth-first search from s, exploring the search-tree of a lower numbered node before looking at a higher numbered node, to find all the nodes in Q that can reach s. Note that in the depth-first search we only use incoming edges at each node because we are interested in paths that come into π.

(c) As in the previous step, for each node u in Q we find the highest point y on π such that there is a path from y to u in G. As before, we label node y to be marked and call it the *exit point* of u. This computation can also be done by conducting a depth-first search in a manner analogous to the previous step.

(d) To sparsely represent the paths that cross π we construct a substitute $\mathbf{sub}_\pi$. The substitute $\mathbf{sub}_\pi$ consists of a *spine* π' that is formed by removing all the unmarked nodes of π and adding directed edges between consecutive marked nodes, directed from the lower numbered to the higher numbered node. Each selected node u is attached to the spine by edges to its entry and exit points. We add a directed edge from u to its entry point x, and an edge to u from its exit point y. See Figure 2.4 for an example.

The sparse substitute $\mathbf{sub}_c$, called the *crossing substitute*, formed by the union of the $\mathbf{sub}_\pi$ for all the π in Z represents reachability between nodes in M and O.

3. **Dividing G for the recursion:** For each node x in G we determine whether there is a directed path between two nodes u and v in M (respectively in O) that passes through x and does not intersect any separating path in Z. If there is such a path then we place x in V_1 (respectively V_2). We construct G_1 and G_2 by taking node-induced subgraphs of V_1 and V_2 respectively. No node is in both V_1 and V_2 at the same time because that would result in a path from some node in M to a node in O that does not intersect any separating path, which is a contradiction.

   To find the nodes of G that go into V_1 (respectively V_2) we first combine all the nodes of M (O) into a single auxiliary node s. We then find the nodes of G that can both reach s and are reachable from s in $G - Z$. Here $G - Z$ is G with the nodes on the separating paths removed.

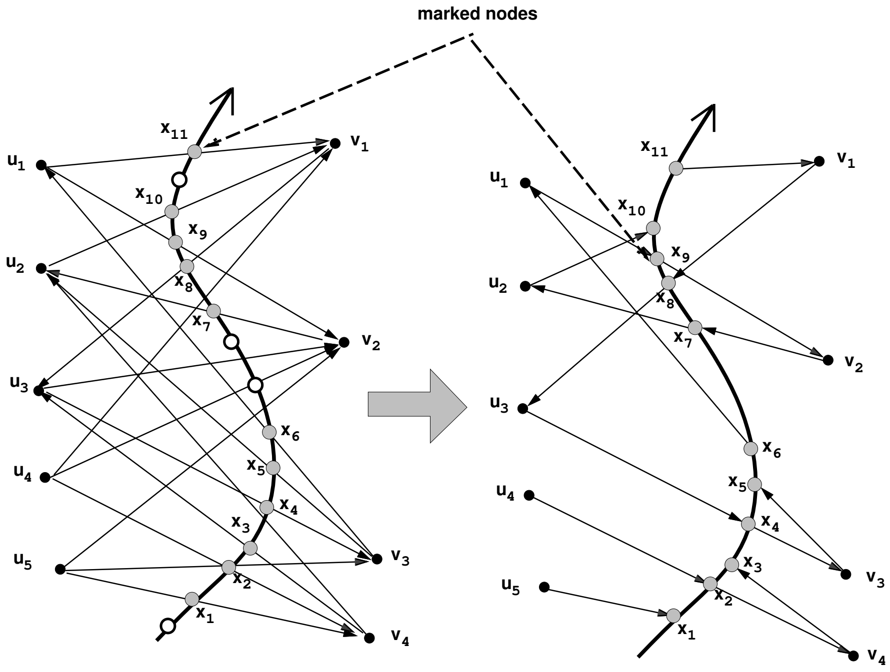

Figure 2.4: Sparsifying the paths that cross a separating path

4. **Recursive computation:** We recursively compute sparse substitutes for all-pairs reachability among the nodes of M in G_1 and the nodes of O in G_2. We union the two recursive substitutes along with the crossing substitute $\mathbf{sub}_c$ to get the face-boundary reachability substitute for all the selected nodes.

A pseudocode version of the algorithm is given in Figure 2.5. Before analyzing our algorithm for sparsification, we describe procedure **separate2**.

As in the case of procedure **separate** consider a division of Q into four equal-sized contiguous subsets A_1, A_2, A_3, and A_4 as shown in Figure 2.3. If there is a path between nodes in A_1 and A_3 or between nodes in A_2 and A_4 then we can use it to divide Q into subsets M and O as shown in Figure 2.3. Note that the nodes of M are topologically separated from those of O by the separator. Thus any path from M to O (or from O to M) crosses the separating path. Furthermore, neither M nor O contain more than three-fourth of the nodes in Q.

When there is no single path that divides Q into roughly equal subsets we can use

```
procedure construct-substitute(G, Q)
    [Finding separating paths]
        Use procedure separate2 to find a set Z of one or two paths
        that divide Q into two (roughly) equal sets M and O
        by cutting all paths between them;

    [Sparsifying intersecting paths]
    1. let the crossing substitute sub_c = φ;
    2. for every path π in Z do
        a. for all the nodes u in Q do
            i. Find the lowest point x on π reachable from u;
            ii. Find the highest point y on π such that u is reachable from y;
            iii. Label the nodes x and y marked;
            iv. Add edges ux and yv to sub_c;
        b. Construct the spine π' by contracting the unmarked nodes of π;
        c. Add the spine π' to sub_c;
        [ end for ]

    [Dividing G for the recursion]
    1. let V_1 = M and V_2 = O;
    2. for each node x in G do
            if there are are nodes u and v in M such that there exists a path
            from u to v in G − Z that passes through x then
            place x in the set V_1 else place it in V_2;
    3. for i = 1, 2 construct G_i by forming the vertex-induced subgraph
        of G induced by the vertices in V_i;

    [Recursive computation of face-boundary reachability substitutes ]
        let sub_1 = construct-substitute(G_1, M) and
            sub_2 = construct-substitute(G_2, O);

    [The final face-boundary reachability substitute]
        let sub = sub_c ∪ sub_1 ∪ sub_2;
end [procedure construct-substitute]
```

Figure 2.5: Constructing the face-boundary reachability substitute

two paths as shown in Figure 2.3. To find these two paths we proceed in a manner similar to that of procedure **separate**.

1. We first number the nodes clockwise along the face f starting at the first node of A_1.

2. We then set a_1 to be the lowest-numbered node in A_1 that has a path to or from some node b_1 in A_2. The path π_1 between a_1 and b_1 is the first path in our separator. If there are no paths between nodes of A_1 and those of A_2 then a_1 and b_1 are set to be the last node in A_1 and π_1 is set to ϕ.

3. Similarly, we set a_2 to be the lowest-numbered node in A_3 that has a path to-or-from some node b_2 in A_4. The path π_2 between them is the second path of our separator. As before, if no such path exists a_2 and b_2 are set to be the last node in A_3 and π_2 is set to ϕ.

4. The nodes in Q are divided into two sets M and O. M contains the nodes of Q from a_1 to a_2 in the clockwise order, including a_1 but excluding a_2, while O contains the rest.

As shown in Figure 2.3 the exterior of face f is topologically separated into three regions R_1, R_2, and R_3 by the two paths π_1 and π_2. Note that either R_1 or R_2 may be empty. Any path between M and O that has one of its end points in R_1 or R_2 is forced to cross one of the separating paths. Also, by the minimality of a_i there is no path from M to O (or from O to M) that lies entirely in R_3 and does not intersect either of the separating paths.

To find the lowest-numbered node $a_1 \in A_1$ that has a path to some node in A_2 we first coalesce the nodes of A_2 into a single supernode s. We then perform a depth-first-search (DFS) from s to find all the nodes in A_1 that can reach s. We also perform a DFS to find all the nodes of A_1 that are reachable from s. The node a_1 is then set to be the lowest-numbered node that has a path π_1 either to-or-from s. The path π_1 is the first path in our separator. Similarly, we can find the lowest-numbered node $a_2 \in A_3$, and the corresponding path π_2 between a_2 and some node in A_4.

The steps outlined above can be carried out in $O(n+k)$ time. We therefore have the following lemma.

Lemma 3 *Procedure* **separate2** *constructs a separating set Z that divides Q into sets M and O such that neither of them has more than three-fourths of the nodes.*

Also, any path from M to O (or from O to M) intersects some path in Z. Moreover, procedure **separate2** *runs in $O(n)$ time.*

To prove the correctness of our sparsification procedure all we need to do is prove that the crossing substitute $\mathbf{sub}_c$ correctly represents the reachability information due to paths that go from the nodes in M to those in O and vice-versa. Consider any path ρ from some node u in M (or O) to a node v in O (or M). We now show that there is a corresponding path ρ' in $\mathbf{sub}_c$ from u to v.

By Lemma 3 we know that ρ crosses some path in Z. Let us suppose it crosses the path $\pi \in Z$ and the intersection point is p. By construction of $\mathbf{sub}_\pi$ we have edges ux and yv in $\mathbf{sub}_\pi$ where x is the entry point of u and y is the exit point of v. By definition x is numbered no higher than p and y is numbered no lower than p. Furthermore, y is reachable from x in $\mathbf{sub}_\pi$, since x occurs before y on the spine π'. Therefore, v is reachable from u in $\mathbf{sub}_\pi$ (and therefore in $\mathbf{sub}_c$) by the alternate path ρ' that uses the edges ux and yv and the subpath from x to y in π'. For example in Figure 2.4 the path from the selected node u_3 to the selected node v_2 is replaced by an alternate path that detours via the marked nodes x_4 and x_9. Note that if v is reachable from u in $\mathbf{sub}_c$ it is also reachable from u in G because every edge in $\mathbf{sub}_c$ corresponds to a path in G.

To bound the size of $\mathbf{sub}_c$ we note that there can be at most $O(k)$ marked nodes since there are at most two paths in Z and each selected node marks no more than two nodes on any path in Z. This also implies that the size of the spine π' (for any π in Z) is at most $2k$. Furthermore, a selected node is attached, via directed edges, to at most four nodes in $\mathbf{sub}_c$. Therefore, the total number of nodes and edges in $\mathbf{sub}_c$ is $O(k)$ (see Figure 2.4). By Lemma 3 the number of selected nodes goes down by a constant factor at every stage in the recursion. Therefore, the size of the face-boundary reachability substitute $\mathbf{sub}$ is $O(k \log k)$.

To bound the running time we note that the construction of $\mathbf{sub}_c$ involves a call to procedure **separate2** and a constant number of depth-first search operations. The total time needed is therefore the sum of the time needed for the recursive computation and the time needed by procedure **separate2**, which is $O(n)$ by Lemma 3. Since the depth of the recursion is proportional to $\log k$ it is easy to see that the total time required by our sparsification procedure is $O(n \log n)$. We thus get the bounds of Theorem 5.

2.3 Fully dynamic data structures for approximate shortest paths and reachability

In this section we design a fully dynamic data structure for maintaining approximate all-pairs shortest paths and the all-pairs reachability in planar graphs. The data structure for the shortest-path problem works for undirected planar graphs with nonnegative edge weights and supports the following operations.

1. **distance**(u, v): Returns the approximate distance between u and v in G.
2. **add**(u, v, w): Results in the addition of the edge uv with weight w.
3. **delete**(u, v): Results in the deletion of the edge uv.

The data structure for reachability supports the following operations.

1. **is-reachable**(u, v): Returns true if v is reachable from u in G.
2. **add**(u, v): Results in the addition of the directed edge uv.
3. **delete**(u, v): Results in the deletion of the directed edge uv.

In this section we will assume that all the edge-additions are planarity-preserving. To see whether edge additions preserve planarity we can run the planarity-testing algorithm from [37] in the background to prevent addition of edges that destroy planarity. Doing this only increases the time-complexity of our update operations by a constant factor. In the descriptions of our algorithms we will not explicitly mention these additional steps.

For both the shortest-path problem and the reachability problem, in the preprocessing stage we find a face-restricted k-cluster partition of G (see Section 1.4). For each piece G_i in the partition we now find a sparse substitute $\hat{G}_i$. For the shortest-path problem $\hat{G}_i$ is the face-boundary shortest-path substitute and for the reachability problem

procedure distance(u, v)
[Forming the auxiliary graph]
$H \leftarrow G_u \cup G_v \cup S$.

[The Query]
1. Run Dijkstra's algorithm in H with u as the source node.
2. Return the distance between u and v found in the previous step.
end [procedure distance]

Figure 2.6: Finding approximate shortest-paths using the skeleton S.

$\hat{G}_i$ is the face-boundary reachability substitute. The substitute graphs $\hat{G}_1, \hat{G}_2, \cdots, \hat{G}_j$ are unioned to form a skeletal graph S.

In our data structures we set k to be $n^{2/3}$ which results in a cluster partition with $O(n^{1/3})$ clusters each of which has no more than $n^{2/3}$ edges. Furthermore, each cluster has $O(n^{1/3})$ boundary nodes that are distributed over at most f faces for some constant f.[1]

Given the face-restricted cluster partition as described above for the shortest-path problem the size of the skeletal graph S is $O(n^{2/3} \log n \log D)$ while its size for the reachability problem is $O(n^{2/3} \log n)$. It also follows from Lemma 2 and Theorem 4 that the total amount of time needed for preprocessing in the case of shortest-paths is $O(n \log n \log D)$. Similarly by Lemma 2 and Theorem 5, the preprocessing time required for the reachability problem is $O(n \log n)$.

In the shortest-path problem the skeletal graph S is a compact representation for approximate all-pairs shortest paths between the boundary nodes and in the reachability problem it is a compact representation for the all-pairs reachability information between boundary nodes. The skeletal graph S is used in the query-stage to determine shortest-paths and reachability information respectively in the two problems.

To perform the query **distance**(u, v) we form an auxiliary graph H by unioning the regions containing u and v along with the skeletal graph S. We then run a sequential shortest-path algorithm on the auxiliary graph H to compute the distance between u and v. See Figure 2.6 for the query-procedure **distance**.

[1] In our data structure we set the value of f to be 4.

procedure is-reachable(u, v)
  [Forming the auxiliary graph]
    $H \leftarrow G_u \cup G_v \cup S.$

  [The Query]
  1. Perform depth-first search in H with u as the source node.
  2. Return "true" if a path to v is found in step **1.**
  end [procedure is-reachable]

Figure 2.7: Determining reachability using the skeleton S.

To perform the query **is-reachable**(u, v) we form an auxiliary graph H by unioning the parent regions G_u and G_v of u and v respectively along with the skeletal graph S. We then perform a depth-first search from u, in H, to see if v is reachable from it in G. See Figure 2.7 for the query-procedure **is-reachable**.

To prove the correctness of our query procedure for the shortest-path problem we need to show that the distance between u and v in H is an accurate estimate of the distance between u and v in G. For each G_i, by construction, the substitute graph $\hat{G}_i$ represents all-boundary-pair shortest paths to within a $1+\epsilon$ factor. Consider a path π of minimum length from u to v in G. Mark all the boundary nodes in π. This marking divides π into a sequence of subpaths such that the first and the last subpaths lie entirely within G_u and G_v respectively and each intermediate subpath is a boundary-to-boundary path that lies entirely within one of the subgraphs G_1 through G_j. Since the boundary-to-boundary shortest paths in G_i are well-estimated by the substitute graph $\hat{G}_i$ it follows that H contains a path from u to v whose length is no more than $1+\epsilon$ times the length of π. A similar argument shows that we can correctly determine the reachability between u and v by doing a depth-first search in H.

To bound the time taken for performing the query we note that H is formed by unioning S and two regions with at most $n^{2/3}$ edges. Therefore, in the shortest-path problem the size of H is $O(n^{2/3} \log n \log D)$, and in the reachability problem the size of H is $O(n^{2/3} \log n)$. Thus, the query procedure **distance** requires $O(n^{2/3} \log^2 n \log D)$ time while the query procedure **is-reachable** requires $O(n^{2/3} \log n)$ time.

We now describe the procedures **add** and **delete** for updating the data structures. The basic idea is to recompute the face-boundary substitues (for shortest paths or reachability) for all the regions that are affected by the update operation. The crux of the proof lies in showing that only a constant number of regions are affected. We discuss the case of an add operation, the delete operation is similar.

Consider adding the edge uv: This results in a change in both G_u and G_v since now both u and v become (if they were not already) boundary nodes. Furthermore, the addition could result in changing the distribution of boundary nodes in G_u, G_v, or both. If in any of them more than f faces now contain boundary nodes, we split that region into smaller regions where boundary nodes are distributed over no more than f faces. However, performing these extra subdivsions only creates a constant number of regions. Therefore, to recompute the face-boundary shortest-path substitutes for all these regions we require $O(n^{2/3} \log n \log D)$ time. Similarly, to recompute the face-boundary reachability substitutes for the affected pieces we only require $O(n^{2/3} \log n)$ time.

A careful look at our updating algorithm shows that repeated requests for adding edges can result in an excess of boundary nodes, since every time an edge uv is added either u or v or both may become boundary nodes. Therefore, we recompute the cluster partition after every $n^{1/3}$ add operations. We also recompute the face-boundary substitutes of the pieces in the partition.

To amortize the rebuild operation over *adds* we note that the cluster partitioning and the rebuilding of all the face-boundary shortest-path substitutes can be done in $O(n \log n \log D)$ time. This coupled with the fact that a rebuild is done once every $n^{1/3}$ *adds* gives us the bounds of Theorem 2 for maintaining approximate shortest paths.

Similarly, we note that the cluster partitioning and the rebuilding of all the face-boundary reachability substitutes can be done in $O(n \log n)$ time. This coupled with the fact that a rebuild is done once every $n^{1/3}$ *adds* gives us the bounds of Theorem 3 for maintaining all-pairs reachability.

Chapter 3

A randomized parallel algorithm for computing shortest paths in directed graphs

In this chapter we investigate the problem of computing single-source shortest paths in parallel. We will restrict our attention to the case when the edges have nonnegative weights. As discussed in Chapter 1 the sequential algorithms for this problem are quite efficient; they require only slightly more than linear time [3,31,32,53]. As discussed in Chapter 1 the known polylog-time parallel algorithms for this problem are inefficient in their processor utilization.

It is a longstanding open problem to find a work-efficient polylog-time algorithm for shortest paths. The gap between what we can and cannot achieve is particularly evident when we consider sparse graphs. For such graphs, the work required by known polylog-time algorithms is essentially the cube of the sequential time required. Put another way, using p processors yields a speed-up of roughly $p^{1/3}$ instead of p.

In this chapter we give work-efficient parallel algorithms for solving the single-source shortest-path problem in sparse digraphs with moderate edge-lengths. Our algorithm gives a polynomial-factor improvement over matrix multiplication in the processor-time product when the input graph is sparse.

Theorem 6 *There is a randomized parallel algorithm that, given a directed graph G with integer lengths and given a source node s, computes shortest-path distances from s to all the other nodes in the graph. With high probability (i.e. probability exceeding $1 - n^{-\Omega(1)}$) the algorithm takes time $O(\sqrt{n}\log^2 n \log L)$ and work $O(m\sqrt{n}\log^2 n \log L)$, where n, m, and L are, respectively, the number of nodes, the number of edges, and the maximum edge-length in G.*[1]

In Chapter 4 we investigate the special case when the underlying graph is planar.

By comparing the work required by our algorithm to the $O(m + n\log n)$ time required by a sequential algorithm for shortest paths, we see that our algorithm incurs an $O(\sqrt{n}\log^2 n \log L)$ factor overhead. Our algorithm is based on a parallel breadth-first search algorithm due to Ullman and Yannakakis [96].

Ullman and Yannakakis show that a breadth-first-search tree can be constructed in $O(\sqrt{n}\,\text{polylog } n)$ time using a linear number of processors. Their algorithm is limited, however, in that it cannot handle lengths on nodes or edges. Thus one of our main contributions is to extend their algorithm so that it can do so. We show that using a simple rounding technique enables their algorithm to approximate shortest paths in the presence of lengths. This technique of finding approximate shortest paths is presented in Section 3.3. It is essentially a method for finding approximate shortest paths that have a limited number of edges. We use this in conjunction with with a scaling technique, for reducing the exact problem to a series of approximate ones to solve the exact problem. The scaling technique presented in Section 3.1 is similar to the one used by Gabow [34] for sequential computation of shortest paths.

In other related work, a parallel algorithm for shortest paths was discovered by Spencer [91]. Actually, his algorithm involves a tradeoff between time and work. Spencer specifies his bounds in terms of a parameter ρ and an upper bound L on edge-lengths . For graphs with nonnegative integral edge lengths at most L the time required

[1] In the rest of the chapter we will use the phrase "*with high probability*" to mean, with probability $1 - n^{-\Omega(1)}$.

is $O((n/\rho)\log n \log(\rho L))$ and the work is $O(n\rho^2 \log\rho \log(\rho L) + (m + n\log L)\text{polylog}(n))$. For comparison, let us ignore logarithmic factors; in order to achieve about $\sqrt{n}$ time, Spencer's algorithm would need to do about n^2 work, whereas our algorithm would need to do about $m\sqrt{n}$ work. Thus our algorithm is considerably more efficient in the case where the graph is sparse (i.e. m not much bigger than n) and less efficient if the graph is dense (i.e. m bigger than $n^{1.5}$). Put another way, if the graph is sparse ($m = O(n)$) and one only has n processors available, our algorithm would use the n processors to achieve a speed-up of about $\sqrt{n}$ while Spencer's algorithm would achieve a speed-up of about $n^{1/4}$.

Subsequent work In recent work Cohen [13,16] has given an algorithm to find $1+\epsilon$-approximate shortest paths which among other things uses the limited search technique presented in Section 3.3. Her algorithm works in polylogarithmic time and is nearly work-optimal. However, her algorithm does not generalize to directed graphs; and there seems to be no way to use it to compute exact shortest paths by repeated approximation even if the underlying graph is undirected.

More recently Shi and Spencer [88] have given a parallel shortest-path algorithm, for undirected graphs, with work-bounds that are similar to ours. Given $\log n \leq t \leq n$ their algorithm runs in $\tilde{O}(t)$ time with $\tilde{O}((n^3/t^2) + m)$ work, or in $\tilde{O}(t)$ time with $\tilde{O}((n^3/t^3) + mn/t)$ work.

To compare the relative efficiency of the two algorithms we set t to be $\sqrt{n}$. If we assume that the edge-lengths are polynomially bounded then for undirected graphs the two algorithms require roughly the same amount of work. However like Cohen's algorithm, their algorithm does not generalize to directed graphs either.

The rest of the chapter is organized as follows. In Section 3.1 we show how to reduce the exact problem to a sequence of approximate problems. The same reduction is also used by the algorithm presented in Chapter 4. In Section 3.2 we discuss some techniques for searching a graph in parallel and describe the Ullman-Yannakakis algorithm for breadth first search. In Section 3.3 we describe procedure FullSearch which is a technique for finding approximate shortest paths in parallel. Finally in Section 3.4 we show how to use procedure FullSearch along with the technique of Ullman and Yannakakis to derive the algorithm of Theorem 6.

3.1 Computing exact shortest paths by repeated approximation

In this section we show how to reduce the problem of computing exact shortest paths in a directed graph G to a series of approximate shortest-path computations.

Gabow [34] has given a scaling algorithm to compute single-source shortest paths in an n-node graph G with maximum edge-length L. His algorithm works by solving a series of $\log L$ instances of the problem (involving the same graph) in which each shortest-path distance is guaranteed to be at most n. For such an instance, any edge whose length exceeds n is superfluous and can be discarded. By using this reduction, we can assume henceforth that the sum of edge-lengths is at most n^3. This reduces the dependence of the algorithm's work-bound on the magnitude of the lengths L.[2]

Next we show how to reduce an exact problem of this form to an approximate problem of the same form. The reduction again uses the central idea of Gabow's scaling algorithm, namely modifying edge-lengths according to node *prices* derived from a previous shortest-path calculation.

We define the $(1-\epsilon)$-approximate shortest-path problem to be computing distance estimates $d(v)$ from a given root with the following properties.

- (A) There is an auxiliary graph that contains G as a subgraph such that the estimates are exact shortest-path distances in the auxiliary graph.

- (B) If the distance in the original graph from one node to another is d, the distance in the auxiliary graph is between $(1-\epsilon)d$ and d.

Thus in the $(1-\epsilon)$-approximate problem we are interested in getting underestimates

[2] We thank David Karger for pointing this out to us.

instead of overestimates. We now prove that the single-source shortest-path problem is efficiently polylog-time-reducible to the $(1-\frac{1}{2})$-approximate shortest-path problem.

Lemma 4 *Single-source directed shortest paths in a graph G with nonnegative integral lengths can be reduced to the $(1-\frac{1}{2})$-approximate shortest-path problem on the same graph. If D is the maximum path length then the exact problem can be solved using $\log D$ calls to an algorithm for the $\frac{1}{2}$-approximate problem in the same graph (with different nonnegative integral lengths).*

Proof: We give a recursive algorithm and show it has recursion depth $\log D$. The algorithm first computes distance estimates $d(v)$ from the source. If these estimates are all zero, then by property (B) of the estimates, the exact distances are all zero. Otherwise the algorithm recursively computes exact shortest-path distances $\widehat{f}(v)$ in a graph with lengths $\ell(uv)$ defined by

$$\widehat{\ell}(uv) := \ell(uv) + \lceil d(u) \rceil - \lceil d(v) \rceil$$

Finally the algorithm determines exact shortest-path distances $f(v)$ in the original graph by setting $f(v) = \widehat{f}(v) + \lceil d(v) \rceil$.

The correctness of this procedure is proved as follows. For any node x and any path P from the source to x,

$$\begin{aligned} \widehat{\ell}(P) &= \sum_{uv \in P} (\lceil d(u) \rceil + \ell(uv) - \lceil d(v) \rceil) \\ &= \lceil d(\text{source}) \rceil + \ell(P) - \lceil d(x) \rceil \end{aligned} \tag{3.1}$$

Since $\lceil d(source) \rceil = 0$, it follows that the shortest path to x indeed has length $f(v)$.

Next we show that the new lengths $\widehat{\ell}$ are nonnegative. For any edge uv of the original graph, the u-to-v distance in the auxiliary graph is clearly at most $\ell(uv)$, so we have $d(v) \leq d(u) + \ell(uv)$. It follows that $\lceil d(v) \rceil \leq \lceil d(u) \rceil + \ell(uv)$, so $\widehat{\ell}(vw) \geq 0$.

To show that the recursion depth is $\log D$, it suffices to show that the maximum shortest-path distance with respect to the lengths $\widehat{\ell}$ is at most $D/2$. For any node x, let P be a shortest path to x according to lengths $\ell(uv)$. By property (B) of the distance estimates, $\lceil d(x) \rceil \geq (1 - \frac{1}{2})\ell(P)$. Hence by (3.1), the distance to x according to lengths $\widehat{\ell}(uv)$ is at most $\ell(P)/2$. □

3.2 Parallel BFS and the Ullman-Yannakakis algorithm

In this section we discuss several techniques for searching a graph in parallel. In particular we describe the Ullman-Yannakakis algorithm for parallel breadth first search and discuss a version of breadth-first search that is suitable for weighted graphs. These ideas will then be used to solve the $(1 - \frac{1}{2})$-approximate shortest-path problem.

The fastest known parallel algorithm for breadth-first search involves repeatedly squaring a matrix, where the element-wise operations are in the min–plus semiring. In fact, this algorithm can compute shortest paths (see [2,25] for more details). The problem with this algorithm is that it requires too many processors; to achieve $O(\log n)$ time requires about n^3 processors. The processor bound has been improved somewhat [41] in the case of breadth-first search.

The most elementary parallel search technique is parallel breadth-first search. In this technique the nodes are visited level by level as the search progresses. Level 0 consists of the starting node s, level 1 consists of the neighbors of s, level 2 consists of the neighbors of the neighbors of s (or, rather, those not belonging to previous levels), and so forth.

To go from, say, level i to level $i+1$, the processors divide up the edges with endpoints among the level i nodes; for each such edge, if the other endpoint has not yet been traversed, it is assigned to level $i+1$. Suppose e_i is the number of such edges; then we can perform this computation in $O(e_i/p)$ time with high probability using the algorithm of [47] for processor allocation. Hence the time required to traverse k levels is $O(e/p + k)$ with high probability.

3.2.1 Limited search

The problem with this approach is that the time required grows linearly with the number of levels traversed. To keep the time small, we must resort to a limited search, in which we limit the number of levels traversed. A *k-limited shortest path* from s to t is a path from s to t that is no longer than any s-to-t path of size at most k. Note that a k-limited shortest path need not itself have size at most k. For any given error parameter $\epsilon > 0$ we define a k-limited *ϵ-approximate* shortest path to be one whose length is at most $1 + \epsilon$ times the length of any path of size at most k. We show how to search such paths in parallel in roughly $O(k\epsilon^{-1})$ iterations.

An estimate d of the distance from s to t is a k-limited ϵ-approximate shortest-path distance if there is an s-to-t path of length at most d, and d is at most $1 + \epsilon$ times the length of any path of size at most k. In the case of all lengths being 1, it is easy to find k-limited shortest paths in k iterations; one just uses k iterations of parallel breadth-first search to find the actual shortest paths to all nodes at distance at most k from the source. We call this *k-limited breadth-first search*; it is essentially the method used in the algorithm of Ullman and Yannakakis.

3.2.2 The Ullman-Yannakakis algorithm

The basic version of Ullman and Yannakakis' parallel algorithm for breadth-first search is based on $\sqrt{n}\log n$-limited search. We review a simplified version. We are given a graph and a source s from which to find a breadth-first search tree. As described in Section 1.5 their algorithm is based on searching from a set of randomly selected nodes.

In this algorithm first, $O(\sqrt{n})$ randomly chosen nodes, along with the source, are designated as *distinguished.* The constant hidden by the big-Oh determines the probability that the algorithm is correct, as we mention below. From each of the distinguished nodes x in parallel, p processors conduct a $\sqrt{n}\log n$-limited search, identifying the minimum path-size from x to each node within path-size $\sqrt{n}$. By choosing $p = m/\sqrt{n}$, we can carry out all of these searches in $O(\sqrt{n}\log n)$ time using $p\sqrt{n} = m$ processors.

Second, an auxiliary graph is formed on the set of distinguished nodes, where the length of an edge from u to v is defined to be the minimum path-size from u to v found during the limited search. All-pairs shortest paths are then computed for the auxiliary graph. This takes total work $O((\sqrt{n})^3 \log n)$, and can easily be done in $O(\sqrt{n} \log n)$ time using m processors. In particular, we have the length of the shortest path in the auxiliary graph from the source s to each of the distinguished nodes.

Third, the minimum path-size from the source s to a node v is computed by taking the minimum, over all distinguished nodes x, of the auxiliary-graph shortest path length from s to x plus the minimum path-size from x to v found in x's limited search.

We can use Theorem 1 to show that, with high probability, this method yields the minimum path-size for all the nodes v. The proof is as follows. Fix in advance one minimum-size path P_v from s to v, for every node v. Now consider the randomly selected distinguished nodes. For any given constant c, by Theorem 1, with probability $1 - \frac{1}{n^c}$ none of the shortest paths P_v has a $\sqrt{n} \log n$-gap with respect to the distinguished nodes. Here, the constant c determines the constant factor in the number of randomly chosen nodes.

Thus the path P_v is broken up into subpaths of size at most $\sqrt{n} \log n$ with the following properties.

1. The last subpath starts at a distinguished node and ends at v.

2. All other subpaths start and end at distinguished nodes.

These subpaths are traversed in the limited search. Hence the minimum path-size from s to the last distinguished node of P_v is correctly computed in the all-pairs computation, and the minimum path-size from s to v is correctly computed in the last step of the algorithm.

Ullman and Yannakakis' approach does not directly work with weighted graphs because there is no apparent way to find even the $\sqrt{n} \log n$-limited shortest paths within the available resource bounds. Our first contribution is a method that approximates these shortest paths.

In Section 3.3 we provide a procedure that, given a source node, finds estimated distances to all nodes from the source such that the estimated distance is at least the shortest-path distance and no more than $1+\epsilon$ times the $\sqrt{n}\log n$-limited shortest-path distance. Given a graph with edge lengths that sum up to D, the procedure works in $O(\sqrt{n}\epsilon^{-1}\log n)$ time doing $O(m\sqrt{n}\log n \log D)$ work.

Note that this procedure is sufficient to make Ullman and Yannakakis' procedure work for finding approximate single-source shortest paths in weighted graphs. We use the limited distances found by procedure FullSearch to create the auxiliary graph. The second and third steps of the Ullman-Yannakakis algorithm do not depend on the lengths being one. By applying essentially the same proof as sketched above for Ullman and Yannakakis' algorithm, it is easy to see that the final estimated distances are at least the actual distances and at most $1+\epsilon$ times actual distances. In Section 3.4 we show how to solve the exact problem using the approximation algorithm sketched here.

3.2.3 Approximating k-limited shortest paths

There is an obvious approach to extending parallel breadth-first search to handle positive integral weights. *Weighted parallel breadth-first search* is just like ordinary parallel breadth-first search, except that for each active edge, if the edge's length is one, its head is assigned to the next level, and if the edge's length is more than one, its length is decreased by one. Thus level i consists of those nodes at distance i from the source.

Weighted parallel breadth-first search has the same disadvantage as ordinary parallel breadth-first search, only more so. The time required grows linearly with the distance traversed. This is especially a problem when the lengths are large. To alleviate this disadvantage, the obvious fix is to uniformly shrink all lengths. This in itself is not enough; the search crucially depends on the lengths being integral. Thus we must round the shrunken lengths up to nearest integers without substantially changing the length of the shortest path.

In Section 3.3, we outline a procedure FullSearch(G, s, k, p, ϵ) for finding approximate k-limited shortest paths from source s in the directed graph G. Here ϵ is a positive number strictly less than one and the estimated path-lengths are at least the actual lengths and no more than $1 + \epsilon$ times the actual lengths. In this case, we say that the estimates are accurate *to within a relative error of* ϵ, or, more briefly, to within ϵ.

Let p be the number of processors available and D be the sum of the edge lengths of G, procedure *FullSearch* finds approximate k-limited shortest paths in time

$$O(\frac{m \log D}{p} + k\epsilon^{-1}),$$

where D is the sum of edge-lengths.

When we use procedure FULLSEARCH for computing approximate k-limited shortest paths, we assign input parameters $k = \sqrt{n} \log n$ and $p = m \log D / \sqrt{n}\epsilon^{-1}$. With these parameters, procedure FULLSEARCH takes time $O(\sqrt{n}\epsilon^{-1} \log n)$. Since Ullman and Yannakakis' algorithm must carry out limited search from each of $O(\sqrt{n})$ nodes simultaneously, the number of processors required for all these calls is $O(m\epsilon \log D)$. Therefore the work done by the algorithm is $O(m\sqrt{n} \log n \log D)$. As discussed above, this approximation algorithm is then used along with the scaling technique of Section 3.1 to derive the final algorithm.

3.3 Procedure FULLSEARCH

In this section we give the procedure FULLSEARCH for finding approximate k-limited shortest paths from a given source node s. We first describe the sub-procedure SEARCH$(G, s, d, k, p, \epsilon)$. It determines the k-limited ϵ-approximate shortest path distances from s of those nodes whose distance is at least d and at most $2d$.

Procedure FULLSEARCH finds k-limited approximate shortest path distances of all nodes simply by calling SEARCH$(G, s, d, k, p, \epsilon)$ in parallel with $d = 1, 2, 4, \ldots$. Since D is the sum of the edge lengths we need to consider at most $\log D$ different values for d in order to determine approximate k-limited shortest paths to all the nodes. This can be accomplished with a $(1 + \log D)$- factor increase in the number of processors. Before calling procedure SEARCH procedure FULLSEARCH finds all the nodes reachable from the source by k-limited paths of length 0. This can be done by performing a k-limited breadth first search from s as in the Ullman-Yannakakis algorithm. For this stage we only use the zero-length edges of G.

Using p processors, procedure $\textsc{Search}(G, s, d, k, p, \epsilon)$ takes time

$$O(\frac{m}{p} + k\epsilon^{-1}) \tag{3.2}$$

where m is the number of edges and n is the number of nodes in G. The first step of the procedure is to define the *scale factor* α by

$$\alpha = \epsilon d/k \tag{3.3}$$

Next, we obtain approximate edge-lengths by rounding each length up to the nearest integer multiple of α. Zero lengths are rounded up to α.

$$\widehat{\ell}(e) = \begin{cases} \alpha\lceil \ell(e)/\alpha\rceil & \text{if } \ell(e) > 0 \\ \alpha & \text{if } \ell(e) = 0 \end{cases}$$

Since the resulting lengths are positive integer multiples of α, we can apply weighted breadth-first search where in each level we traverse a distance α. We search from the source for $\lceil 2(1+\epsilon)k/\epsilon\rceil$ levels and label each node reached with the estimated distance, i.e., α times the number of levels the search required to reach the node. The estimated distance to a node not reached by the search is implicitly taken to be infinite.

The time required to traverse $\lceil 2(1+\epsilon)k/\epsilon\rceil$ levels is $O(m/p + \frac{2k}{\epsilon})$ with high probability using p processors. Next we show that the estimated distances computed are not too bad.

The following proposition follows immediately from the fact that the approximate edge-lengths are at least the actual edge-lengths.

Proposition 1 *For any path P, $\widehat{\ell}(P) \geq \ell(P)$.*

Proposition 1 ensures that we never underestimate distances. We might overestimate distances, but lengths of paths with at most k edges and distance at least d are not overestimated by much.

Lemma 5 *If P is a source-to-v path of size k and length x such that $d \leq x \leq 2d$ then the estimated distance to v is at most $1+\epsilon$ times the length of P.*

Proof: For each edge e of P, $\widehat{\ell}(e) \leq \ell(e) + \alpha$, so $\widehat{\ell}(P) \leq \ell(P) + k\alpha$. By choice of α, we have $k\alpha = \epsilon d$. Since $\ell(P)$ is at least d, we obtain $\widehat{\ell}(P) \leq (1+\epsilon)\ell(P)$. Furthermore, since $\ell(P) \leq 2d$, we have $\widehat{\ell}(P) \leq 2(1+\epsilon)d$, so $\lceil 2(1+\epsilon)k/\epsilon \rceil$ levels are sufficient to traverse P. Hence the estimated distance to v is at most $\widehat{\ell}(P)$. □

Procedure SEARCH$(G, s, d, k, p, \epsilon)$ only accurately estimates the k-limited distances of nodes when those distances are between d and $2d$. To estimate the distances of all nodes, we call SEARCH in parallel with different values of d. Let D be the sum of edge-lengths. Procedure FULLSEARCH(G, s, k, p, ϵ) simply makes $1 + \lceil \log D \rceil$ calls to SEARCH$(G, s, d, k, p, \epsilon)$, each with d assigned a different power of 2 from 2^0 to $2^{\lceil \log D \rceil}$. For each node, we take its estimated distance to be the minimum estimated distance obtained in all these calls.

Lemma 6 *Procedure* FULLSEARCH *computes k-limited ϵ-approximate shortest-path distances in time*

$$O\left(\frac{m \log D}{p} + k\epsilon^{-1}\right) \tag{3.4}$$

Proof: For any node v, the estimated distance to v is at least the actual distance, by Proposition 1. Suppose v is at distance $d(v)$ from the source. Let $d_0 = 2^{\lfloor \log d(v) \rfloor}$, so $d_0 \leq d(v) \leq 2d_0$. By Lemma 5 when SEARCH is called with $d = d_0$ the estimated distance to v is at most $1+\epsilon$ times the length of any path to v with $\leq k$ edges. The time bounds follow from the fact that the $1 + \lceil \log D \rceil$ calls to procedure SEARCH can all be made in parallel. □

3.4 Using procedure FULLSEARCH to solve the $(1-1/2)$-approximate shortest-path problem

We now show how our approximation algorithm can be used to get underestimates that satisfy properties (A) and (B) (see Section 3.1). We closely follow the outline of the Ullman-Yannakakis algorithm. Let G be the input graph, and let n and m denote, respectively, the number of nodes and number of edges in G. The first two steps are as follows. Randomly select $(c+3)\sqrt{n}$ distinguished nodes, where c is a constant. Include the source node among the distinguished nodes. Use FULLSEARCH to find $(1+\epsilon)$-approximate $\sqrt{n}\log n$-limited shortest paths from each distinguished node. The following is a corollary of Theorem 1 and is the key to our algorithm.

Corollary 1 *With probability at least $1-n^{-c}$, for every pair of nodes $u, v \in G$ there is a shortest u-to-v path such that every subpath of size $\sqrt{n}\log n$ contains a distinguished node.*

Next, construct a graph H whose nodes are the distinguished nodes and where there is an edge uv of length $(1-\epsilon)d$ if the estimated u-to-v distance is d. Use matrix powering to compute all-pairs shortest paths in H and let H^* be the complete graph in which the length of the edge uv is the u-to-v distance in H. We next show that distances in the union of H^* with the input graph G satisfies property (B).

Lemma 7 *For every pair of nodes $u, v \in G$, if the u-to-v distance in G is d then the u-to-v distance in $H^* \cup G$ is between $(1-\epsilon)d$ and d.*

Proof: Since every path in G is a path in $H^* \cup G$, the u-to-v distance in $H^* \cup G$ is no more than the corresponding distance in G. Conversely, let P be a shortest u-to-v path in $H^* \cup G$. We show how to obtain a path Q in G such that $(1-\epsilon)\ell(Q) \leq \ell(P)$. The lemma therefore follows.

To construct Q we first construct P' from P by replacing each edge of H^* with the corresponding shortest path in H. Since the length of each edge of H^* equals the length of the corresponding shortest path in H, the length of P' equals the length of P. We obtain Q from P' by replacing each edge uv of H with the shortest u-to-v path P_{uv} of G. The length of each edge uv of H equals $1-\epsilon$ times the estimated u-to-v distance in G, which in turn is at least the actual u-to-v distance. It follows that the length of P is at least $1-\epsilon$ times the length of Q. □

Finally, we obtain single-source distances in $H^* \cup G$. To do this, we rely on the following lemma.

Lemma 8 *With probability at least $1 - n^{-c}$, for any node v, there is a shortest source-to-v path in $H^* \cup G$ that consists of at most $1 + \sqrt{n}\log n$ edges.*

Proof: Assume the high-probability event of Corollary 1 occurs, and let P be a shortest source-to-v path in $H^* \cup G$. Obtain P' from P as follows: for each maximal subpath P_i of P consisting solely of edges of G, replace P_i by the shortest path with the same endpoints whose existence is promised by Corollary 1. The resulting path P' has the following structure. It consists of subpaths in G of size at most $\sqrt{n}\log n$ starting and ending at distinguished nodes, interspersed with edges of H^*, and ending with a subpath in G of size at most $\sqrt{n}\log n$. Furthermore, since P' was obtained by replacing shortest paths in G with shortest paths in G, the length of P' equals the length of P.

Obtain P'' from P' as follows. For each subpath P_i in G of size at most $\sqrt{n}\log n$ starting and ending at distinguished nodes, replace P_i with the edge e_i of H with the same endpoints. Suppose P_i is a u-to-v path, and note that since the estimated u-to-v distance is at most $1 + \epsilon$ times the length of P_i, and the length of e_i is defined to be $1 - \epsilon$ times the estimated distance, the length of e_i is at most the length of P_i. Hence the length of P'' is no more than the length of P'.

The structure of P'' is as follows. It consists of a subpath of edges of $H^* \cup H$, terminating in a subpath of G of size at most $\sqrt{n}\log n$. Obtain P''' from P'' by replacing the subpath of edges of $H^* \cup H$ by the single edge with the same endpoints. It follows that the length of P''' is no more than the length of P''' and P''' consists of at most $1 + \sqrt{n}\log n$ edges. □

Lemma 8 shows that to compute single-source distances in $H^* \cup G$, it is sufficient to consider paths of size at most $k = 1 + \sqrt{n}\log n$. Cohen [14] has observed that the following variant of Bellman-Ford correctly computes shortest paths in this case. Set $d(v) := \infty$ for each node v, except set $d(\text{source}) := 0$. Repeat the following step k times. For each node v other than the source, compute

$$d(v) := \min\{d(u) + \text{length}(uv) \ : \ uv \text{ an incoming edge of } v\}$$

Now we analyze the above algorithm. Carrying out FullSearch for each distinguished node takes $O(\epsilon^{-1}\sqrt{n}\log n)$ time and $O(m\sqrt{n}\log n \log D)$ work. Since $D \leq n^3$, the work bound is $O(m\sqrt{n}\log^2 n)$. Constructing the graph H can be done within the same bounds. Since H has $O(\sqrt{n})$ nodes, computing all-pairs shortest paths in H can be done in $O(\sqrt{n}\log n)$ time and $O(n^{1.5}\log n)$ work. Finally, using the Bellman-Ford variant to compute single-source shortest paths requires $O(\sqrt{n}\log n)$ stages (as noted above). Each stage can be implemented in constant time with high probability by using Megiddo's [75] algorithm for computing minimum. Therefore the shortest paths in $H^* \cup G$ can be found in $O(\sqrt{n}\log n)$ time and $O(m\sqrt{n}\log n)$ work.

We can therefore solve the exact problem by using the scaling technique of Section 3.1 along with the approximateion algorithm outlined above for finding prices. We therefore get the bounds of Theorem 6.

Chapter 4

A linear-processor polylog-time algorithm for shortest paths in planar digraphs

In this chapter we present a nearly work-optimal NC algorithm for finding single-source shortest paths in planar digraphs. Our algorithm makes use of the scaling technique presented in Chapter 3 along with a technique that is essentially a directed version of path-detour from Chapter 2.

Theorem 7 *For n-node planar directed graphs with integral nonnegative edge-lengths $\leq L$, single-source shortest paths can be solved in time $O(\text{polylog } n \log L)$, with probability $1 - n^{-\Omega(1)}$, using n processors.*

The techniques used in our algorithm for planar graphs can also be used to find single-source shortest-paths in general directed graphs. The resulting algorithm although not as efficient as the one presented in Chapter 3 is considerably faster.

Theorem 8 *There is a randomized parallel algorithm that, given an m-edge directed graph G with integer lengths $\leq L$, and given a source node s, computes shortest-path distances from s to all the other nodes in the graph. With probability $1 - m^{\Omega(1)}$ the algorithm runs in $O(\text{polylog } m \log L)$ with m^2 processors.*

Our algorithm for planar digraphs does not depend upon planarity and can be generalized to apply also to any minor-closed family of graphs for which a decomposition tree (based on $O(\sqrt{n}\,\text{polylog } n)$-node separators) as defined in Section 1.4 is provided. For example, given a decomposition tree, we can use our algorithm to find single-source shortest paths in constant-genus graphs.

The best previously known polylog-time algorithm for single-source shortest paths in planar undirected graphs is that of Pan and Reif [82,83], which does $\tilde{O}(n^{1.5})$ work. Cohen [14] has given an algorithm with similar bounds for the directed case. The previous bounds are no better for breadth-first search, the special case where every edge-length is one.

As discussed in Chapter 1 our shortest-path algorithm applied to planar graphs in particular has several further applications. It can be used to obtain an efficient polylog-time algorithm for minimum st cut and maximum st-flow in planar graphs (see Section 4.5 for more details). Previous best algorithms for all these problems relied on the nested dissection algorithm of Pan and Reif [83] and therefore required $\tilde{O}(n^{1.5})$ work.

Our shortest-path algorithm can be generalized to handle negative lengths as long as no negative-length edge appears in a directed cycle. In particular we have the following theorem.

Theorem 9 *There exists a randomized parallel algorithm for finding single-source shortest paths in planar digraphs that do not have any directed cycle with negative-weight edges in them. Given such an n-node planar digraph G with integral edge-lengths between $-L$ and L the algorithm uses n processors and with high probability works in time $O(\text{polylog } n \log L)$.*

If G is an arbitrary n-node m-edge digraph with the same restriction on the distribution of the negative edges then the algorithm uses m^2 processors and with high probability finds single-source shortest paths in $O(\text{polylog } m \log L)$ time.

Using this algorithm, we can solve the longest-path problem in a dag. By using the technique of Hassin and Johnson [49], we can find the maximum flow in an undirected planar network where s and t are not necessarily on the same face, both in polylog time and using a linear number of processors. See section 4.5 for more details.

The top-level description of our shortest-path algorithm is as follows. We use the scaling reduction of Chapter 3. To solve the approximate problem, as in Chapter 3, we show how to construct a graph in which shortest-path distances approximate shortest-path distances in the original graph. The auxiliary graph created in Chapter 3 was guaranteed to have shortest paths with at most $\sqrt{n}\log n$ edges. However, this is not sufficient to implement a polylog-time solution. We therefore construct an auxiliary graph in which there exist shortest paths with polylogarithmic number of edges. Again, we use a parallel version of the Bellman-Ford algorithm to quickly find shortest paths in this auxiliary graph.

To construct a graph that approximates shortest paths in the original graph we use similar but denser graphs on the separators. This idea was used by Lingas [71] in the context of breadth-first search and later by Cohen [14] for shortest-paths. However, in [14] the dense graphs on the separators were simply complete graphs where there was an direct edge uv representing the shortest path from u to v. In our algorithm, we too use a graph in which each edge represents a path, but the shortest path may consist of more than one edge. We call this graph a *summary* of the original graph. Summaries are formally defined in Subsection 4.1.2. Our main subprocedures, outlined in Section 4.3, operate on summaries. Since a summary is typically dense, naive approaches to manipulating a summary would be too expensive. The key idea of our approach, incorporated into Lemma 16, uses the fact (see Section 4.2) that in a dense summary of a sparse graph, many pairs of edges represent intersecting paths of the original graph. This fact provides the basis for efficiently searching a summary. Section 4.2 describes how these intersecting paths can be found and Lemma 9 outlines a procedure to find paths all of which intersect a given path. This procedure is similar to the path-detour technique of Chapter 2.

The rest of the chapter is organized as follows. In Section 4.1 we discuss some of the basic ingredients used in the algorithm including a parallel version of the path-detour technique suitable for directed graphs (see Lemmas 9, 10, and 11). The path-detour technique is the basis for our parallel algorithm. In Section 4.2 we describe how to find intersecting paths to facilitate the parallel search procedure outlined in Lemmas 9, 10, and 11. In Section 4.3 we describe the approximation algorithm for constructing the auxiliary graph that is the key to obtaining the bounds of Theorem 7, and in Section 4.4 we show how to extend the algorithm to get the bounds of Theorem 8. Finally in Section 4.5 we address the problem of handling negative-weight edges. We also discuss the application of this extension to computing longest paths and maximum st-flows.

4.1 The ingredients

In this section we describe some of the basic ingredients used by our parallel algorithm. In particular we detail a parallel version of the path-detour technique that is suitable for directed graphs.

4.1.1 Limited Bellman-Ford search from a source-path

Given a graph G with a source, and given a positive integer r, we consider the problem of computing, for each node v, the shortest source-to-v path consisting of at most r edges. As discussed in Chapter 3 this problem can be solved in $O(r \log |G|)$ time and $O(r|G|)$ work by running the Bellman-Ford algorithm for r stages. We now show how to adapt this idea to give a shortest-path procedure that finds approximate r-edge distances in a graph G to a path $P = \eta_1 \ldots \eta_s$. This portion is similar to the idea of finding shortest paths from the detour node of P in the undirected setting (see Section 1.6).

Lemma 9 *There is a procedure that takes as inputs a graph G, a path P in G, a positive integer r, a distance bound d^*, and an error parameter ϵ. The procedure computes the following for each node v in G and each distance $d \leq d^*$ that is a multiple of ϵd^*:*

(A) Either a node η in P such that

1. *There is a v-to-η path of length at most $d + \epsilon d^*$, and*
2. *For every node η' occurring before η in P, every v-to-η' path consisting of at most r edges has length more than d.*

(B) or ∞, if there is no v-to-P path of length at most d and size at most r.

The procedure takes time $O(r \log m)$ and work $O(m\epsilon^{-1}r^2)$, where m is the number of edges in G.

If $d^* = 0$ then for each node v the procedure finds the lowest node η such that there is a v-to-η path. It is easy to see that this satisfies conditions (1) and (2). To find η for all the nodes v we can perform a depth first search (looking at paths of length $\leq r$) from the tail of P.

Otherwise, first the procedure discards edges of length more than d^*, as such edges obviously do not participate in paths of length at most d^*. Next it uses rounding to replace the edge-lengths with small integers. Let $\ell(vw)$ denote the length of edge vw, and define $\widehat{\ell}(vw) := \lceil \ell(vw)/\alpha \rceil$, where $\alpha = \epsilon d^*/(r+1)$. Let $i^* = (r+1)\epsilon^{-1}$, and note that $\widehat{\ell}(vw) \leq i^*$.

Let the source-path be $P = \eta_1\eta_2 \ldots \eta_s$. We give a procedure using lengths $\widehat{\ell}(\cdot)$ that computes a table T_v of size $i^* + 1$ for every node v, satisfying the following *correctness condition*: for each nonnegative $i \leq i^*$, $T_v[i]$ is the minimum index j such that there is a v-to-η_j path of length at most i and size at most r. If there is no such index, $T_v[i] = \infty$.

Before giving this procedure, we show how it can be used to prove Lemma 9. For any distance $d \leq d^*$ that is a multiple of ϵd^*, let $i = \lceil \frac{d}{\alpha} + r \rceil$. For a node v, if $T_v[i]$ is ∞ then there is no v-to-P path of length at most d and size at most r. If $T_v[i] = j$ then there is a v-to-η_j path that has $\widehat{\ell}$-length at most $\lceil \frac{d}{\alpha} + r \rceil$. Therefore the actual length of the path to η_j is no more than $\alpha \lceil \frac{d}{\alpha} + r \rceil \leq d + \epsilon d^*$. Hence η_j satisfies (1) of Lemma 9. Moreover, for any η' occurring before η_j in P, every v-to-η' path Q of size at most r satisfies

$$i < \sum_{vw \in Q} \lceil \ell(vw)/\alpha \rceil \leq \left(\sum_{vw \in Q} \ell(vw)/\alpha \right) + r$$

Substituting for i in the inequality above we see that

$$\lceil \frac{d}{\alpha} + r \rceil < \frac{\ell(Q)}{\alpha} + r$$

Therefore $d < \ell(Q)$. Hence η_j satisfies condition (2) of Lemma 9.

Now we give the procedure. To initialize the tables, for every i we set $T_v[i] := \infty$ for all nodes v not in the path P, and set $T_{\eta_j}[i] := j$ for nodes η_j in the path.

The basic step is relaxing all edges by simultaneously updating each table T_v as follows.

$$T'_v[i] := \min\{T_v[i], \min\{T_w[i - \ell(vw)] \ : \ \ell(vw) \leq i\}\}. \tag{4.1}$$

It is a simple induction on r to show that, after r iterations of applying (4.1) to all the nodes of G simultaneously, the correctness condition holds. Each of these r iterations requires time $O(\log m)$ and work $O(i^* m)$, where m is the number of edges of G. Substituting for i^* yields the bounds of Lemma 9. A similar procedure can be used to compute r-limited shortest-paths from P to the nodes of G.

4.1.2 Summaries

A *summary* H of G is a graph consisting of nodes of G (*ordinary nodes*) and *segment nodes*. Each segment node is a *copy* of some node of G. We define all such copies to be distinct.[1] Each edge uv of H corresponds to a path in G between the nodes corresponding to u and v. Note that distinct edges need not map to disjoint paths. The mapping from edges of H to paths of G is represented by a *derivation dag*, described later.

The segment nodes of H are partitioned into disjoint paths, called *segments*. The only edges in H between segment nodes are the edges belonging to the segments. These edges are called *segment edges* to distinguish them from *ordinary edges*, which connect ordinary nodes to segment nodes or to other ordinary nodes. The *segment graph* of H

[1] Hence when two summaries are unioned, segment nodes are never coalesced, as happens to ordinary nodes appearing in both summaries.

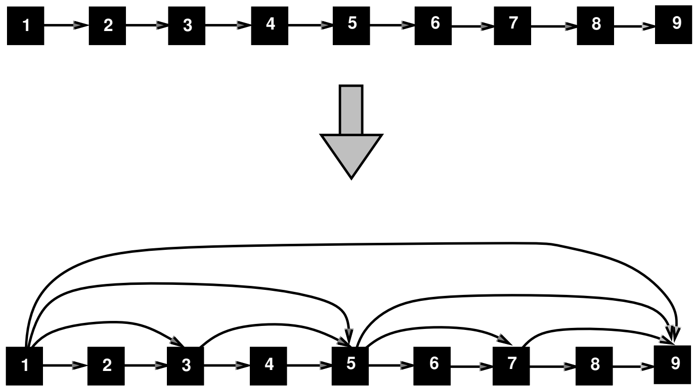

Figure 4.1: Augmenting a segment with jump edges.

is the multi-graph obtained by coalescing each segment into a single node. A *supernode* of H is either an ordinary node or a segment. Thus the nodeset of the segment graph of H consists of the supernodes of H. Note that the segment graph may have multiple edges between the same endpoint. Note also that it has no edges between segments. We define the *edge-multiplicity* of the segment graph to be equal to the maximum number of edges between any two nodes of the segment graph.

A *hop* in H is a path from one ordinary node to another all of whose internal nodes are segment nodes belonging to a single segment. Since there are no edges between segments, any path in H between ordinary nodes can be decomposed into hops. A k-hop path is one consisting of at most k hops.

4.1.3 Applying limited Bellman-Ford to a summary

In computing limited shortest paths in a summary, we cannot afford to spend lots of time traversing the many edges in a segment. For purposes of computing shortest paths, therefore, we shall assume the segments have been *augmented* with *jump edges* as shown in Figure 4.1. Each jump edge is assigned a length equal to the sum of the lengths jumped, so distances are not affected. These jump edges have the property that any subpath of the segment can be replaced by a path of at most $2 \log s$ jump edges, where s is the size of the segment. Moreover, given any subset X of the nodes of the segment, there is a node-induced subgraph consisting of $|X| \log s$ nodes and $2|X| \log s$ edges that has the same property with respect to X: any path in the original segment starting and ending at nodes of X can be replaced by a path in the subgraph consisting of at most $2\log s$ edges. This is called the subgraph *relevant* to X. We assume in the following lemmas that the segments of the input summary H have been thus augmented. We also assume that sufficient preprocessing has taken place that, given a subset X of the nodes of a segment, it is easy to obtain the relevant subgraph.

Lemma 10 *Given an augmented summary H and a source node v of H, k-hop-limited shortest paths to all ordinary nodes of H can be found in time $O(k\log^2 n_0)$ and work $O(km\log^3 n_0)$, where n_0 is the number of nodes in H and m is the number of ordinary edges.*

Proof: In order to achieve a work bound that depends only on the number m of ordinary edges, rather than on the total number of edges in the summary, we must restrict our attention to only some of the segment nodes and edges. Call a segment node *active* if it has incident ordinary edges. For each segment, we consider only the subgraph relevant to the segment's active nodes. hence the total number of nodes and edges under consideration is $O(m\log n_0)$. By applying limited Bellman-Ford search from the source node v with $r = 2k\log n_0$, we obtain the desired shortest paths. $\square$

4.1.4 Parallel Path-detour: Fast parallel approximation of k-hop paths that cross P

Next we make use of the shortest-path approximation algorithm of Lemma 9 for searching from a path P. Our goal is to construct a representation of nearly shortest k-hop paths that go through the path P. Lemma 11 outlines the parallel version of our path-detour technique.

Lemma 11 *There is a procedure that, given an augmented summary H, a distance bound d^*, an accuracy parameter ϵ, a path P in H of length at most $\epsilon d^*/4$, and a positive integer k, constructs a P-*substitute *for H, consisting of a* copy *P' of P and a set of* pseudoedges *between ordinary nodes and nodes of P', with the following properties:*

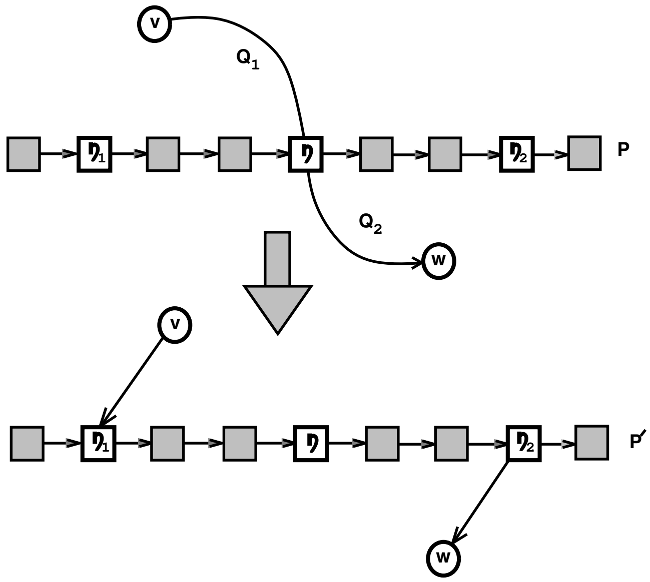

Figure 4.2: The P-substitute satisfies correctness properties (III) and (IV).

I. Each ordinary node has $O(\epsilon^{-1})$ pseudoedges to and from P'.

II. Each edge in P' has cost $1-\epsilon$ times the corresponding edge in P.

III. Each pseudoedge corresponds to a path in H and has length $1-\epsilon$ times the length of the path.

IV. For ordinary nodes v and w, if there is a v-to-w path of length $d \in [d^*/2, d^*]$ consisting of at most k hops and containing a node of P, then there is a v-to-w path of length at most d consisting of a pseudoedge, followed by a subpath of P', followed by a pseudoedge.

The procedure constructs the P-substitute in time $O(k\log^2 n_0)$ and work $O(m\epsilon^{-1}k^2\log^3 n_0)$, where n_0 is the number of nodes in H.

Proof: As before, we consider only relevant subgraphs of active nodes within each segment, so the number of edges under consideration is $O(m\log n_0)$. The procedure to construct the P-substitute is as follows. Apply the procedure of Lemma 9 to H' with $r = (2+2\log n_0)k$, distance bound d^*, and error parameter $\bar{\epsilon} = \epsilon/16$. For each ordinary node v we construct $\bar{\epsilon}^{-1}$ outgoing edges, one for each distance $d \leq d^*$ that is a multiple of $\bar{\epsilon}d^*$. The edge corresponding to distance d is $v\eta$, where η is the copy in P' of the node of P satisfying (1) and (2) of Lemma 9, namely:

(1) there is a v-to-η path of length at most $d + \bar{\epsilon}d^*$, and

(2) for every node η' occurring before η in P, every v-to-η' path consisting of at most r edges has length more than d.

The length assigned to the pseudoedge is $1-\epsilon$ times the length of the path it represents, namely the path of (1). For those d that satisfy condition (B) of Lemma 9 there is no corresponding pseudoedge.

We have described the construction of outgoing pseudoedges. Incoming pseudoedges are constructed analogously. The time and work for this procedure are dominated by the time and work required by the two calls to the procedure of Lemma 9. The bounds of Lemma 11 therefore follow.

Now we show the construction satisfies the correctness properties. Properties (I) and (II) are immediate. Let Q be a v-to-w path of length between $d^*/2$ and d^* consisting of at most k hops and containing a node η of P. Write $Q = Q_1\eta Q_2$, and let $d_1 = (\lceil \ell(Q_1)/\bar{\epsilon}d^* \rceil)\bar{\epsilon}d^*$, and $d_2 = (\lceil \ell(Q_2)/\bar{\epsilon}d^* \rceil)\bar{\epsilon}d^*$. Let $v\eta_1$ be the outgoing pseudoedge in the substitute that corresponds to distance d_1, and let $\eta_2 w$ be the incoming pseudoedge that corresponds to distance d_2. By construction $\ell(Q_1) \leq d_1 \leq \ell(Q_1) + \bar{\epsilon}d^*$ and $\ell(Q_2) \leq d_2 \leq \ell(Q_2) + 2\bar{\epsilon}d^*$. Therefore by condition (2) of Lemma 9 (as shown in Figure 4.2) η occurs no earlier than η_1 and no later than η_2 on P. Thus η_1 occurs no later than η_2 on P. Also by condition (1) of Lemma 9, the length of the v-to-η_1 path is at most $d_1 + \bar{\epsilon}d^*$ and the length of η_2-to-w path is no more than $d_2 + \bar{\epsilon}d^*$.

Let P'' be the subpath of P' from η_1 to η_2. The path consisting of the pseudoedge $v\eta_1$ followed by P'' followed by $\eta_2 w$ has length at most

$$(1-\epsilon)[d_1 + d_2 + 2\bar{\epsilon}d^* + \ell(P)] \leq (1-\epsilon)[\ell(Q_1) + \ell(Q_2) + 4\bar{\epsilon}d^* + \ell(P)]$$

Substituting for $\bar{\epsilon}$ and $\ell(P)$ we see that the length of the new path is at most $(1-\epsilon)[\ell(Q) + \epsilon d^*/2]$. Since $\ell(Q) \geq d^*/2$, the length of the new path $(1-\epsilon)[\ell(Q) + \epsilon d^*/2]$ is less than or equal to $\ell(Q)$. The lemma therefore follows. $\square$

4.1.5 The derivation dag

In manipulating a summary H of a graph G, it is useful to maintain the mapping from edges of H to paths in G. A *derivation dag* for a graph G and summary H is a directed acyclic graph (dag) whose vertices represent paths in G. The vertices with no predecessors represent edges of G. Every other vertex v has an ordered pair of predecessors; the path represented by v is the path obtained by composition from the paths represented by v's predecessors. We require that for each edge of H the corresponding path of G is represented by some vertex in the derivation dag. For brevity, we abuse the terminology and say that the edge of H is represented by the vertex. We also require that the the derivation dag be shallow and not be much bigger than the graph G. In particular, the derivation dag should have depth polylogarithmic in $|G|$ and size $\tilde{O}(|G|)$. Here *depth* is the maximum number of edges in a directed path in the dag.

In connection with a derivation dag, we will use the term *constructing* a path P to mean adding to the dag a vertex representing P (including adding arcs and additional vertices as needed.) We shall have need on occasion to construct a subpath P' of a path P already represented. This is facilitated by the fact that one can always find a logarithmic number of paths represented in the dag whose composition is P'.

We list some of the operations that can be performed with the help of the derivation dag. Each of these operations can be performed by doing a constant number of upward and downward sweeps of the dag. Hence for each operation, the work is proportional to the size of the dag, and the parallel time is proportional the its depth. By our requirements about the size and depth of the derivation dag, the time is therefore polylogarithmic in $|G|$, and the work is $\tilde{O}(|G|)$.

Derivation dag operations:

1. Given a set of represented paths (i.e. given the corresponding vertices of the derivation dag), find the set of underlying nodes S in G belonging to these paths.

2. Given a set S of nodes of G, find the set of represented paths containing nodes

in S.

3. Given a length parameter r and a set X of represented paths, partition the paths in X into node-disjoint segments of length at most r. The partitioning has the following minimality property: for two adjacent segments of the same path of X, if the two segments were combined into one, the resulting segment would be longer than r.

4. Given a set of represented paths X and a set of nodes S of G, for each path π in X determine one of the nodes in S it contains (if any), and find and construct the two subpaths of π, **(1)** up to and **(2)** after the node.

5. Given a set X of represented paths; for every other represented path π determine whether it intersects some path in X or not. If π intersects a path in X find one such intersecting path α, and find the lengths of the two subpaths π_1, and π_2 (of π) formed by the intersection of π and α.

Throughout this chapter we will assume that all the relevant paths are represented in the derivation dag. During the course of the algorithm every path that is discovered/constructed (for example by the parallel searching procedures of Lemmas 9, 10, and 11), will be represented in the derivation dag by constructing the appropriate edges and nodes. We will not explicitly mention these steps.

4.2 Near-covers

The edges of a summary H of G need not map to disjoint paths in G. We say two edges of H *intersect* if the corresponding paths in G share a node. In this section we describe a procedure to choose a few edges in the summary H such that a large number of other edges of H intersect the chosen edges. This in conjunction with the limited Bellman-Ford search technique outlined in Lemmas 9, 10, and 11 is used to find approximate shortest paths in Section 4.3.

We say G is *sparse* if every subgraph of G has at most c times as many edges as nodes, where c is a constant. For example, constant-degree graphs and planar graphs are sparse. If G is sparse but H is dense, it intuitively follows that many of the edges of H intersect. We use this idea in the lemmas below, which are the basis for our shortest-path approximation algorithm.

For the remainder of this chapter unless otherwise stated G will denote a sparse graph from a minor-closed family of graphs. Given such a G the following lemma can be proved using an argument due to Leighton [68] for lower-bounding the crossing number of a graph.

Lemma 12 *Let H be a summary of G such that the segment graph of H has m edges and n nodes and maximum edge-multiplicity 1. If the number of ordinary edges m in H is $\Omega(n)$ then $\Omega(m^3/n^2)$ pairs of ordinary edges of H intersect.*

A *near-cover* of H is a set of ordinary edges A such that a constant fraction of the ordinary edges of H intersect some edge in A. As a corollary of Lemma 12 we can get the following bounds on the size of a near-cover for the summary H:

Corollary 2 *Let G be a sparse graph from a minor-closed family of graphs and let H be a summary of a G such that the segment graph of H has m edges and n nodes and maximum edge-multiplicity r. If $m = \Omega(rn)$ then there exists a near-cover of H of size $O(r^2n^2/m)$.*

Proof: We can write $H = H_1 \cup H_2 \cup \ldots \cup H_r$, where for each i the segment graph of H_i has at least m/r ordinary edges and at most one edge between each pair of nodes. It suffices to find a near-cover of each H_i of size $O(rn^2/m)$. Let $m_i \geq m/r$ be the number of ordinary edges in H_i. It follows from Lemma 12 that $\Omega(m^3/(r^3n^2))$ pairs of ordinary edge in H_i intersect. Therefore, the number of edges that an average edge of H_i intersects is $\Omega(m^2/(r^2n^2))$. This means we can pick a near-cover A_i of $O(rn^2/m)$ ordinary edges such that a constant fraction of the ordinary edges of H_i intersect one or more edges in A_i. Unioning the near-cover for each H_i, we get the near-cover for H. $\square$

4.2.1 Finding a near-cover in parallel

It is possible to construct a near-cover satisfying Lemma 2 by using a greedy procedure that repeatedly picks and deletes the edge of H that accounts for the maximum intersection. However, we are interested in finding a near-cover in parallel. We therefore use the following randomized algorithm.

Given a summary H such that its segment graph has m edges and n nodes we first choose a subset A of the ordinary edges of H by choosing each edge with probability $2\delta n^2/m^2$, where δ is a constant to be chosen later. We then determine $|A|$ and the number of the ordinary edges of H that intersect edges in A. If $|A| > 3\delta n^2/m$ or if fewer than a fraction γ of the ordinary edges of H intersect A, we try again, choosing another A. Otherwise, we return A as the near-cover. If the edges of H are represented as paths in a derivation dag each iteration of this algorithm can be efficiently implemented by using a constant number of derivation dag operations derivation-dag operations.

By definition the algorithm returns a good near-cover. To prove that it terminates quickly we reformulate the problem in a slightly different fashion. Let H_I*, called the intersection graph of* H, be constructed as follows:

1. For every ordinary edge e in H, H_I contains a corresponding node v_e.
2. There is an edge in H_I between two nodes v_e and v_f if the corresponding edges e and f of H intersect.

We say that a node v_e *dominates* another node v_f in H_I if there is an edge between v_e and v_f. A set R of nodes dominates a node v_e if v_e is adjacent to at least one node in R.

Given this reformulation we can restate Lemma 12 as:

Lemma 13 *If H is a summary of G such that its segment graph n-nodes and m-edges then the intersection graph H_I of H has $\Omega(m^3/n^2)$ edges.*

Under this reformulation a near-cover is just a subset of nodes R in H_I that dominate a constant fraction of the nodes of H_I. Also note that under this reformulation Lemma 12 translates to: The number of edges in H_I is at least m^3/n^2.

The algorithm is the same as before: We select a random subset R of the nodes by choosing each node with probability $2\delta n^2/m^2$ and check to see if it is a near-cover. If it is then we stop and return R. Otherwise we try again. Note that we don't actually construct H_I. This reformulation is only for the purposes of the proof.

To establish a time-bound on the running time of our algorithm we show that a randomly chosen R satisfies the following properties.

1. The size of R is small, and
2. R dominates a constant fraction of the nodes in H_I.

Therefore if we repeat the random selection process $O(\log n)$ times then with high probability we will find a set R that is small and that dominates the required fraction of the nodes.

First we note that the expected size of R is $|V(H_I)|2\delta\frac{n^2}{m^2} = 2\delta\frac{n^2}{m}$. Therefore, by Markov's inequality $|R|$ is at most $3\delta\frac{n^2}{m}$ with probability at least 1/3.

Next we address the number of nodes dominated by R. We require a technical lemma.

Lemma 14 *Given any directed graph $\vec{H}_I$, there is a set S of nodes of H_I and a set Ar of arcs outgoing from S such that*

1. $|S| \leq |V(\vec{H}_I)|/2$, *and*
2. *the maximum outdegree in $\vec{H}_I - Ar$ is at most one more than twice the average outdegree in the same graph.*

Proof: The proof relies on the following algorithm. Initialize S and A to the empty set, and repeat the following two steps until condition 2 holds. Select a node v with maximum outdegree in $\vec{H}_I - Ar$, and add v to S (v may already belong to S), then select an outgoing arc of v that is not in Ar, and add it to Ar.

The algorithm is guaranteed to terminate because each iteration reduces the number of arcs in $\vec{H}_I - A$, and condition 2 surely holds in a graph with no arcs. Since the algorithm always reduces the outdegree of a node with maximum outdegree, every node in S has either maximum outdegree or one less than maximum, throughout the algorithm.

Suppose that $|S|$ exceeds $|V(H_I)|/2$ at some iteration in the algorithm. Just before that iteration, we had $|S| = |V(H_I)|/2$ and the maximum degree of $\vec{H}_I - A$ was more than twice the average degree, plus one. Hence the total outdegree of nodes of S exceeds $2|S|$(average degree), which is $|V(H_I)|$(average degree), a contradiction. $\square$

We use Lemma 14 to show that there are positive constants a and b such that with probability at least a, the set R dominates at least a fraction b of the nodes of H_I.

Orient the edges of H_I arbitrarily to get $\vec{H}_I$. Apply Lemma 14 to obtain $S \subset V(\vec{H}_I)$ and $Ar \subset A(\vec{H}_I)$ satisfying the conditions of the lemma.

Let Δ be the average outdegree of $\vec{H}_I - Ar$. Let $\{X_v : v \in V(\vec{H}_I)\}$ be independent 0-1 random variables, each with probability δ/Δ of being 1. Define random variables Y_v by

$$Y_v = \begin{cases} 1 & \text{if there is an outgoing arc } v \to w \text{ in } \vec{H}_I - Ar \text{ for which } X_w = 1 \\ 0 & \text{otherwise} \end{cases}$$

Note that

$$Y_v \geq \sum_{v \to w} X_w - \sum_{v \to w, v \to z, w \neq z} X_w X_z$$

Hence the expected value of $\sum_v Y_v$ is at least

$$\sum_v \sum_{v \to w} E[X_w] - \sum_v \sum_{v \to w, v \to z} X_w X_z$$

The first term is $\sum_w (\delta/\Delta)\text{indegree}(w)$, where indegree$(w)$ denotes the indegree of w in $\vec{H}_I - Ar$. This sum equals

$$\begin{aligned} & \frac{\delta}{\Delta}(\text{sum of indegrees}) \\ = \quad & \frac{\delta}{\Delta}(\text{sum of outdegrees}) \\ = \quad & \delta|V(\vec{H}_I)|(\text{average outdegree}/\Delta) \\ = \quad & \delta|V(\vec{H}_I)| \end{aligned}$$

As for the second term, the contribution of a node v is at most

$$\binom{outdegree(w)}{2}\left(\frac{\delta}{\Delta}\right)^2$$

which is at most $3\delta^2$ since outdegree$(w) \leq 2\Delta + 1$ by condition 2 of Lemma 14. Thus the second term is $\sum_v 3\delta^2$, which is $3\delta^2|V(H_I)|$. We have shown that the expected value of $\sum_v Y_v$ is at least $(\delta - 3\delta^2)|V(H_I)|$.

We estimate the average outdegree Δ of $\vec{H}_I - Ar$ as follows. Since $|V(H_I)| = m$ and $|S| \leq |V(H_I)|/2$, by condition 1 of Lemma 14 the number of remaining nodes $|V(H_I) - S|$ is at least $m/2$. By applying Lemma 13 to the graph $H_I - S$, we obtain $|E(H_I - S)| \geq \alpha m^3/n^2$ (for some constant α). Since every edge of $H_I - S$ appears as an arc in $\vec{H}_I - Ar$ and since average outdegree is at least half average total degree, it follows that $\vec{H}_I - Ar$ has average outdegree $\Delta \geq \alpha m^2/2n^2$.

Finally we return to the set R randomly chosen by our algorithm. Since each node is included in R with probability $2\delta n^2/m^2$, a node v is at least as likely to be included as X_v is likely to be 1. It follows that the expected number of nodes dominated by R is at least the expected value of $\sum_v Y_v$, which was shown to be $(\delta - 3\delta^2)|V(H_I)|$. Let $\delta = 1/6$, so this value is $|V(H_I)|/12$. Let $Z = |V(H_I)| -$ (number of nodes dominated by R), so the expected value of Z is at most $|V(H_I)|(1 - \frac{1}{12})$. By Markov's inequality, Z is at most its expected value times $1 + \frac{1}{12}$ with probability $1/13$. When this occurs, Z is at most $|V(H_I)|(1 - \frac{1}{144})$, so the number of nodes dominated by R is at least $|V(H_I)|/144$.

Thus with probability $1/39$ we will have a small set R (of size $\leq \frac{n^2}{2m}$) that dominates $|V(H_I)|/144$ nodes of H_I. This in conjunction with Corollary 2 gives us the following theorem.

Theorem 10 *Let G be a sparse graph from a minor-closed family of graphs and let H be a summary of a G such that the segment graph of H has m edges and n nodes and maximum edge-multiplicity r. If m is $\Omega(rn)$ then there exists a near-cover A of size $O(r^2n^2/m)$. Furthermore, it can be found in expected time $O(\text{polylog } |G|)$ and work $O(r(|G| + |H| + n^2) \text{ polylog } |G|)$.*

4.3 The shortest-path approximation algorithm

In this section we describe our randomized algorithm for single-source shortest-path approximation. Throughout this section we use G_0 to denote the (sparse) input graph, and N to denote $|V(G_0)|$. We also assume (as justified in Section 3.1) that the sum of the edge lengths of G_0 is at most N^3. We use G to denote a subgraph of G_0. Our goal is an algorithm that requires $O(\text{polylog } N)$ time and uses N processors. It is sufficient to give an algorithm requiring $O(\text{polylog } N)$ time and $O(N \text{ polylog } N)$ work, because then the processor bound can be reduced to N at the expense of increasing the running time by an $O(\text{polylog } N)$ factor.

We want the correctness and running time of our algorithm to hold with high probability, i.e., with probability $1-N^{-c}$, where c is any given constant. For this reason the time bounds for our procedures depend on $\log N$. For example, the procedure of Theorem 10 for finding a near-cover in a graph G takes expected time $O(\text{polylog } |G|)$. By having this procedure start over whenever its running time significantly exceeds the expected time, we get a procedure that only starts over $O(\log N)$ times with high probability, for a total running time of $O(\text{polylog } N)$.

For an accuracy parameter ϵ, the algorithm computes $(1-\epsilon)^{O(\log^2 N)}$-approximate shortest-path distances. We therefore assign $\epsilon := \Theta(1/\log^2 N)$ so as to get $\frac{1}{2}$-approximate distances. Our procedures also refer to a parameter k, a limit on number of hops as in Lemma 10 and 11. In order to achieve correctness with high probability, we assign $k := \hat{c} \log N$ where $\hat{c}$ is a constant to be determined.

We use H to denote a summary of G, and we use n and m to denote the number of nodes and edges in the segment graph of H. For every summary H under consideration, each segment has $O(\epsilon^{-1} n)$ nodes and at most $O(\epsilon^{-1})$ edges to and from each ordinary node (cf. Lemma 11). Hence the maximum edge-multiplicity of the segment graph of H is $O(\epsilon^{-1})$. We use $V_o(H)$ and $V_s(H)$ to denote, respectively, the set of ordinary nodes of H and the set of supernodes of H (nodes of the segment graph of H).

For a set S of nodes, S*-distances* are the shortest-path distances (in a given graph) between pairs of nodes in S. An $(1-\epsilon)$-*accurate underestimate* of a distance d is an estimate between $(1-\epsilon)d$ and d; the definition of an $(1+\epsilon)$-accurate overestimate is analogous.

Lemma 15 *Consider two graphs, A and B, where $V(A) \cap V(B) = V'$. If V'-distances in B are $(1-\epsilon_2)$-accurate underestimates of V'-distances in A, and, for some $S \subset V(A)$, S-distances in A are $(1-\epsilon_1)$-accurate underestimates of some other set of distances δ then*

1. *S-distances in $A \cup B$ are $(1-\epsilon_1)(1-\epsilon_2)$-accurate underestimates of δ, and*
2. *V'-distances in B equal V'-distances in $A \cup B$.*

Proof: Follows immediately from the definitions. □

4.3.1 Reducing the size and diameter of a summary

This subsection contains the core of our algorithm for shortest-path approximation. In particular, we describe two procedures, Compact and SmallPath. The first takes as parameters a summary H and a node-subset S, and produces a summary whose size is proportional to S that approximates distances in H. The second takes a summary H and produces another summary of size proportional to that of H that approximates distances in H and that obeys the following crucial property: for any two ordinary nodes u and v, if there is a u-to-v path in the output summary, there is a shortest such path that consists of only $O(\log^2 N)$ hops.

Both these procedures are based on a combination of the random-sampling technique of Ullman and Yannakakis [96] (see Section 1.5) and our procedure Abbreviate for finding k-limited shortest paths. Intuitively, the goal of Abbreviate is to search k-hop paths between n nodes in a summary H with m ordinary edges. Naively this would take work proportional to nm, where m is typically around n^2. Our method performs the search in around n^2 work, using the fact that the summary has a near-cover of size roughly n^2/m. Namely, it finds a near-cover and replaces its edges with paths; it then does searches both to and from all these paths using the procedure of

Lemma 11, effectively searching all k-hop paths that intersect the edges of the near-cover. The work for the search is roughly m per path, for a total of roughly n^2. In order to search the paths that do not intersect the edges of the near-cover, it deletes the edges intersecting the near-cover and recurses.

Lemma 16 *There is a procedure that given a distance bound d^* and a summary H such that no ordinary edge in H has length more than d^* produces a summary $\widehat{H}$ with the following properties:*

A1: $V(H) \cap V(\widehat{H})$ is the set of ordinary nodes of H.

A2: $\widehat{H}$ has at most $\max\{0, n/4\log N - cn^2/\epsilon^3 m\}$ segments, where c is a constant.

A3: For $u, v \in V_0(H)$, if the u-to-v distance in H is d such that $d^*/2 \leq d \leq d^*$, the u-to-v distance in $\widehat{H}$ is at least $(1-\epsilon)d$.

A4: For $u, v \in V_o(H)$, if there is a k-hop u-to-v path of length d in H such that $d^*/2 \leq d \leq d^*$, then the one-hop u-to-v distance in $\widehat{H}$ is at most d.

The procedure runs in time $O(k\,\mathrm{polylog}\,N)$ and requires work $O((|G| + k^2\epsilon^{-4}n^2)\,\mathrm{polylog}\,N)$ with high probability.

Proof: The procedure is recursive. The base case is when the number of ordinary edges is $8cn\epsilon^{-3}\log N$. In this case, the procedure uses the procedure of Lemma 10 to find k-hop-limited shortest paths from every ordinary node to every other ordinary node. The graph $\widehat{H}$ consists of the ordinary nodes of H, together with an edge vw for every pair of ordinary nodes such that a limited shortest path was found from v to w; the length of the edge is taken to be the length of the path. In this case, properties (A1) through (A4) clearly hold. From lemma 10 it can be seen that the time required for this computation is $O(k\log^2 n_0)$ and the work required is $O(k^2n^2\epsilon^{-3}\log^3 n_0 \log N)$.

The recursive case is as follows. First it uses the procedure of Theorem 10 to find a near-cover A of size $cn^2/4\epsilon^2 m$ for H (this is where c is determined), i.e. a set of ordinary edges of H that intersect at least half the ordinary edges. Let H' be the summary obtained from H by deleting all the intersected edges. Let m' be the number of ordinary edges of H'. Since A is a near-cover, $m' \leq m/2$. The procedure calls itself recursively on H', returning a summary $\widehat{H}'$. By the inductive hypothesis,

A1′: $V(H) \cap V(\widehat{H}') = V_0(H)$.

A2′: $\widehat{H}'$ has at most $\max\{0, n/4 \log N - cn^2/\epsilon^3 m'\}$ segments.

We will show how the procedure obtains $\widehat{H}$ from $\widehat{H}'$ by adding $cn^2/\epsilon^3 m$ segments and putting in edges between these segments and the ordinary nodes. Thus property (A1) will hold. The number of segments in $\widehat{H}$ will be $\max\{0, n/4 \log N - cn^2/\epsilon^3 m'\} + cn^2/\epsilon^3 m$. , which is at most $\max\{cn^2/\epsilon^3 m, n/4 \log N - cn^2/\epsilon^3 m\}$ since $m' \leq m/2$. Moreover, since $m > 8cn\epsilon^{-3} \log N$ (else the base case applies), the second term of the $\max\{\}$ dominates the first. Thus (A2) holds.

As for (A3) and (A4), the u-to-v paths in H not represented in H' (and hence not represented in $\widehat{H}'$) are those containing edges intersecting A. The added segments will help represent these paths. Recall that an ordinary edge e is said to intersect an edge $a \in A$ if the path in G that e represents shares a node with the path that a represents. Using the derivation dag, the procedure determines one such node v_e for each intersected edge e. For each such node, the procedure determines one edge $a \in A$ whose path contains the node. We say that node is *essential* to a.

Next the procedure uses the derivation dag to process the paths represented by the edges in A. It decomposes these paths into node-disjoint subpaths of length at most $\epsilon d^*/4$. Since each edge has length at most d^*, each of the $cn^2/4\epsilon^2 m$ paths gives rise to at most $4/\epsilon$ subpaths, for a total of $cn^2/\epsilon^3 m$ subpaths. The procedure applies a splicing operation to each such subpath, splicing out all inessential nodes. When edges are merged due to splicing, their lengths are summed, so the distances between essential nodes are maintained. The spliced subpaths will be segments in $\widehat{H}$.

In order to determine which ordinary edges to include in $\widehat{H}$, the procedure first constructs an auxiliary graph H_A from H, and then applies the procedure of Lemma 11 to it. The auxiliary graph is obtained from H by adding the spliced subpaths and some edges between them and the ordinary nodes. These are determined as follows.

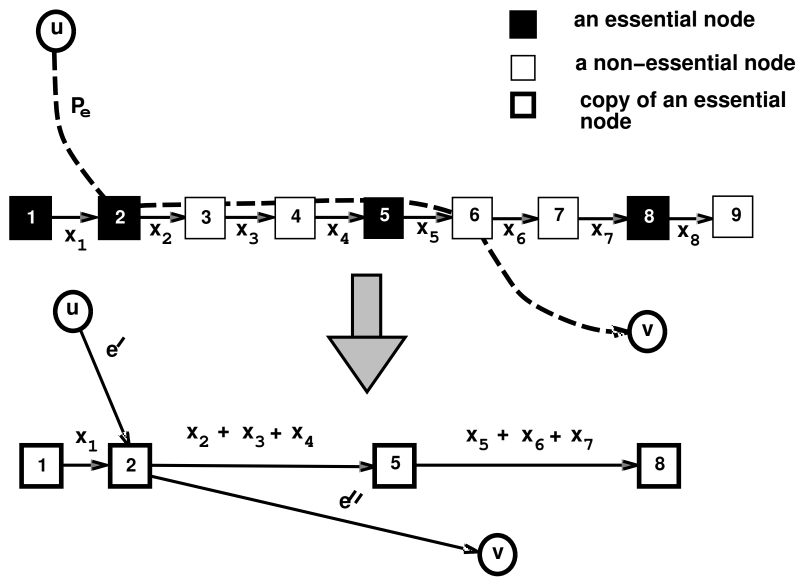

Figure 4.3: Constructing H_A from H.

The procedure processes the paths P_e represented by the intersected edges e. For each such path P_e, the procedure constructs the two subpaths P_e' and P_e'' obtained by splitting P_e at the node v_e. Since v_e is essential to one of the paths represented by the edges in A, a copy of it appears in one of the spliced subpaths. The procedure adds to H_A the edges e' and e'' corresponding to P_e' and P_e'', attaching these edges to the copy of v_e, as shown in Figure 4.3.

Next the procedure calls the procedure of Lemma 11 to find P-substitutes for each of these subpaths P. These P-substitutes are added to $\widehat{H}'$ as segments to obtain $\widehat{H}$. As before, we splice out segment nodes that have no ordinary edges attached to them. Thus the number of segments added is $cn^2/\epsilon^3 m$ as required. The amount of work required to construct one P-substitute is $O(m\epsilon^{-1}k^2 \log^3 n_0)$, so the amount of work required to construct $cn^2/\epsilon^3 m$ substitutes is $O(ck^2n^2\epsilon^{-4} \log^3 n_0)$. Since the depth of recursion is logarithmic, the lemma follows. $\square$

Corollary 3 *There is a procedure* ABBREVIATE *that from a summary H produces a summary $\widehat{H}$ with the following properties:*

B1: $V(H) \cap V(\widehat{H}) = V'$.

B2: $\widehat{H}$ *has at most* $n/4$ *segments.*

B3: For $u, v \in V'$*, if the u-to-v distance in* H *is* d*, the u-to-v distance in* $\widehat{H}$ *is at least* $(1-\epsilon)d$*.*

B4: For $u, v \in V'$*, if the k-hop u-to-v distance in* H *is* d*, then the one-hop u-to-v distance in* $\widehat{H}$ *is at most* d*.*

The procedure runs in time $O(k \operatorname{polylog} N)$ *and requires work* $O((|G|+k^2\epsilon^{-4}n^2) \operatorname{polylog} N)$ *with high probability.*

Proof: The procedure ABBREVIATE is as follows. Let H_i be the subgraph of H consisting of edges of length at most $N^3/2^i$. There are at most $3 \log N$ nonempty subgraphs. The procedure simply applies the procedure of Lemma 16 to each H_i with the distance bound $d^* = N^3/2^i$, and unions the results $\widehat{H}_i$ to form the summary $\widehat{H}$. The only nodes the $\widehat{H}_i$'s have in common are the ordinary nodes of H. Properties (B1) through (B3) are immediate from properties (A1) through (A3) of Lemma 16.

As for property (B4), let P be a k-hop u-to-v path in H, such that $N^3/2^{i+1} \leq l(P) \leq N^3/2^i$. Surely, P exists in H_i. Furthermore, $d^*/2 \leq l(P) \leq d^*$ (where $d^* = N^3/2^i$). Hence by property (A4) the one-hop distance in $\widehat{H}_i$ (and hence in $\widehat{H}$) is at most the length of P. This proves property (B4). The work bound follows from the fact that there are at most $2 \log N$ calls to the procedure of Lemma 16. $\square$

We now restate the random sampling technique of Ullman and Yannakakis (see Theorem 1) in terms of summaries. Let H be a summary of G. For each pair of ordinary nodes u, v, let us designate a shortest u-to-v path P_{uv}. For a summary H of G, suppose we randomly select a subset consisting of half the ordinary nodes of H. Define a *gap* to be a path with no internal nodes being selected. Say a gap is *large* if it consists of at least k hops, where k is the global we defined to be $\hat{c} \log N$ for a constant $\hat{c}$. The following proposition determines the value of $\hat{c}$.

Proposition 2 *With high probability, no designated path* P_{uv} *contains a large gap.*

Proof: For each P_{uv} and each ordinary node w in P_{uv}, consider the $\hat{c}\log N$-hop subpath of P_{uv} starting at w. Say this subpath *fails* if it is a gap. As in Theorem 1 one can show that the probability of a subpath failing is $\exp(-\Omega(\hat{c}\log N))$. Since there are at most N^3 of these subpaths, the probability that any fail is at most $N^3\exp(-\Omega(\hat{c}\log N))$. By choice of the constant $\hat{c}$, therefore, one can ensure that with high probability no subpath fails. $\square$

Lemma 17 *There is a procedure* COMPACT *that, given a summary H and a subset S of the ordinary nodes of H, produces a summary H_S that with high probability has the following properties:*

C1: S-distances in H_S are $(1-\epsilon)^{\log_{8/7} n}$-accurate underestimates of S-distances in H, where n is the number of supernodes in H.

C2: H_S has at most $8|S|$ supernodes.

The procedure runs in time $O(\text{polylog } N)$ and work $O((|G| + \epsilon^{-4}n^2)\text{polylog } N)$ with high probability.

Proof: If $|S| \geq n/8$, then H itself obeys (C1) and (C2), so the procedure returns H. Otherwise, the procedure selects a random subset consisting of half the ordinary nodes of H. Also consider the nodes of S as selected. The procedure applies ABBREVIATE to H, obtaining $\widehat{H}$. The procedure constructs H' from $\widehat{H}$ by deleting all the ordinary nodes that were not selected. Finally the procedure obtains H_S by recursing on H'.

The proof of correctness is by induction on n. By (B2) of Corollary 3, H' has $n/4$ segments. H' has at most $|S| + n/2$ ordinary nodes, and $|S| < n/8$, so H' has a total of at most $\frac{7}{8}n$ supernodes. By the inductive hypothesis, H_S has at most $8|S|$ supernodes so property (C2) holds.

By the inductive hypothesis, S-distances in H_S are $(1-\epsilon)^{(\log_{8/7} n)-1}$-accurate underestimates of S-distances in H'. We claim that with high probability S-distances in H' are $(1-\epsilon)$-accurate underestimates of S-distances in H. It follows that property (C1) holds.

It remains to prove the claim. It follows from (B3) that S-distances in $\widehat{H}$ and hence in H' are no less than $1-\epsilon$ times the corresponding distances in H. We must prove that they are no greater than the corresponding distances. By Proposition 2, for every $u, v \in S$ there is a shortest u-to-v path P_{uv} that decomposes into subpaths, where each subpath consists of at most k hops and starts and ends at selected nodes. By property (B4), for each such subpath there is a one-hop path in $\widehat{H}$ with the same endpoints and having no greater length. Since H' contains all segments in $\widehat{H}$ and all selected nodes, each such one-hop path is contained in H'. This proves the claim. $\square$

Lemma 18 *There is a procedure* SMALLPATH *that from a summary H produces a summary H^* that with high probability has the following properties (n is the number of supernodes in H):*

D1: $V_o(H)$-distances in H^* are $(1-\epsilon)^{\log_{4/3} n}$-accurate underestimates of $V_o(H)$-distances in H.

D2: For any ordinary nodes u, v, a shortest u-to-v path exists in H^* that consists of at most $O(\log N \log n)$ hops

D3: H^* has at most $2n$ supernodes.

The procedure runs in time $O(\text{polylog } N)$ and work $O(|G| + \epsilon^{-4}n^2)\,\text{polylog } N)$ with high probability.

Proof: As in Lemma 17, the procedure selects a random subset of half the ordinary nodes of H. Then the procedure applies ABBREVIATE to H with $k = c \log N$, obtaining $\widehat{H}$. The procedure obtains a subgraph H' of $\widehat{H}$ by deleting all unselected ordinary nodes. The procedure calls itself recursively on H', obtaining H'^*, and returns the union $H^* = H \cup H'^*$.

By (B2), H' has $n/4$ segments. It has at most $n/2$ ordinary nodes, so it has at most $(3/4)n$ supernodes in total. By (D3) of the inductive hypothesis applied to H', $|V_s(H'^*)| \leq 2(3/4)n$. The only supernodes in H^* that are not in H'^* are the unselected ordinary nodes, and there are at most $n/2$ such nodes. Thus the total number of supernodes in H^* is at most $2(3/4)n + n/2 = 2n$. Thus property (D3) holds.

Since H^* contains H as a subgraph, clearly $V_o(H)$-distances in H^* are no greater than the corresponding distances in H. Conversely, (B3) implies that $V_o(H)$-distances in $\widehat{H}$ are at least $1-\epsilon$ times the corresponding distances in H, and so certainly distances in H' between ordinary nodes are at least $1-\epsilon$ times the corresponding distances in H, since H' is a subgraph of $\widehat{H}$. By (D1) of the inductive hypothesis, distances in H'^* between ordinary nodes are at least $(1-\epsilon)^{(\log_{4/3} n)-1}$ times the corresponding distances in H', and hence at least $(1-\epsilon)^{\log_{4/3} n}$ times the corresponding distances in H, proving (D1).

It remains to prove (D2). We use an argument like that used in the proof of Lemma 17.

Claim 1 *For any path Q in $H \cup H'^*$ between ordinary nodes, there exists an equally short path Q' with the same endpoints such that every subpath of Q' consisting of at least $c \log N$ hops contains a node in H'^*.*

Proof: Write $Q = Q_1 v_1 Q_2 v_2 \cdots v_{f-1} Q_f$, where the v_i's are the selected nodes. Then each Q_i not contained in H'^* belongs wholly to H. Replace each such Q_i with the designated shortest path P_{uv} with the same endpoints. The claim follows by Proposition 2.

Property (D2) follows by induction from the following claim.

Claim 2 *For any path Q in $H \cup H'^*$ between ordinary nodes, there exists an equally short path Q'' that either consists of at most $c \log N$ hops or else consists of at most $c \log N$ hops followed by a path wholly in H'^* followed by at most $c \log N$ hops.*

Proof: Let Q' be the path of Claim 1. If Q' consists of at most $c \log N$ hops, we are done. Otherwise, it contains a first selected node u and a last selected node v; moreover, the prefix of Q' ending at u has at most $c \log N$ hops and similarly for the suffix of Q' starting at v. By Lemma 15, the u-to-v subpath of Q' can be replaced by a path lying wholly in H'^*. $\square$

4.3.2 Constructing a graph with small diameter

Our goal in this subsection is to solve the $\frac{1}{2}$-approximate shortest-path problem: given a graph with a source, compute distance underestimates from the source to each of the nodes. These estimates must be actual shortest-path distances in some graph. It follows from Lemma 4 of Chapter 3 that our algorithm to this problem yields an algorithm of comparable performance for the exact shortest-path problem, thus giving us the bounds of Theorem 7.

Given a graph G_0, our goal is to construct a graph $\overline{G}_0$ that is not too much bigger than G_0, such that $V(G_0)$-distances in $\overline{G}_0$ are $(1-\epsilon)^{O(\log^2 N)}$-underestimates of $V(G_0)$-distances in G_0 itself, and such that for any nodes $v, w \in V(G_0)$, if a shortest v-to-w path exists in $\overline{G}_0$, one exists that has only $O(\text{polylog } N)$ hops, and hence, using the augmentation described in Subsection 4.1.3, only $O(\text{polylog } N)$ edges. Single-source shortest paths can then be solved in $\overline{G}_0$ using limited Bellman-Ford. The time required is $O(\text{polylog } N)$ because only $O(\text{polylog } N)$ stages must take place. The work required is $O(|\overline{G}_0| \text{ polylog } N)$.

As stated earlier we assume that G_0 is a sparse graph from a minor-excluded family of graphs. In addition if we suppose that G_0 and its subgraphs have size-$\tilde{O}(\sqrt{n})$ separators, and we have been provided with a decomposition tree for G_0 (or can compute one quickly in parallel), then we show how to construct such a $\overline{G}_0$ of size $\tilde{O}(|G_0|)$. Essentially we apply Lemma 18 to each splitting set of each vertex in the decomposition tree. It is easy to show that because shortest paths between nodes in the same splitting set are small, shortest paths between arbitrary nodes are only slightly larger. Of course, the construction for nodes of the separator must reflect paths between nodes not in the separator. We therefore use bottom-up processing of the decomposition tree to carry out these constructions.

Namely we work up the decomposition tree T of G, constructing a summary for each vertex v of T. The summary for one vertex is obtained by applying procedures COMPACT and SMALLPATH to the union of the summaries for the children of that vertex. We combine all these summaries, constructing a "substitute graph" $G'(v)$ in which distances are underestimates of distances in the pertinent graph $G(v)$ of vertex v and in which shortest paths exist that have only polylog size. For a vertex v of T, let $h(v)$ denote the height of v.

Lemma 19 *For a vertex v of the decomposition tree, we can compute a graph $\overline{G}(v)$ and a summary $H^*(v)$ with the following properties.*

- *E1: $V(G(v))$-distances in $\overline{G}(v)$ are $(1-\epsilon)^{2h(v)\log N}$-accurate underestimates of $V(G(v))$-distances in $G(v)$.*
- *E2: $S(v)$-distances in $H^*(v)$ equal $S(v)$-distances in $\overline{G}(v)$.*
- *E3: For any $a, b \in S(v)$, a shortest a-b path in $H^*(v)$ exists that has $O(\log^2 N)$ hops.*
- *E4: The number of supernodes in $|H^*(v)|$ is $O(|S(v)|)$.*
- *E5: For any two children v_i and v_j of v in the decomposition tree, any path in $\overline{G}(v)$ between $\overline{G}(v_1)$ and $\overline{G}(v_2)$ must contain a node of $S(v)$.*
- *E6: For any $a \in S(v)$ and $b \in V(G(v))$, if there is an a-b path in $\overline{G}(v)$, there is a shortest path of size $O(h(v)\log^3 N)$.*

The time required is $O(\text{polylog } N)$, and the work is $O(\epsilon^{-4}|G(v)| \text{ polylog } N)$.

Proof: The proof is by induction on the height of v. For brevity, we use G to denote $G(v)$ and S to denote $S(v)$. To compute $H^*(v)$, suppose its children are v_1, v_2, v_3. Let $G' = \overline{G}(v_1) \cup \overline{G}(v_2) \cup \overline{G}(v_3)$, and $H' = H^*(v_1) \cup H^*(v_2) \cup H^*(v_3)$. Then it follows from the inductive hypothesis that $V(G)$-distances in G' are $(1-\epsilon)^{2(h(v)-1)\log N}$-accurate underestimates of distances in G, and $V(H')$-distances in H' equal $V(H')$-distances in G'. Apply Compact to H' and $S(v)$, and then apply SmallPath to the result, obtaining $H^*(v)$. Properties E3 and E4 follow immediately from properties C2, D2, and D3. Moreover, by properties C1 and D1, S-distances in $H^*(v)$ are $(1-\epsilon)^{2\log N}$-accurate underestimates of S-distances in H'. It follows that S-distances in $H^*(v)$ are $(1-\epsilon)^{2\log N}$-accurate underestimates of S-distances in G'. Let $\overline{G}(v) = G' \cup H^*(v)$. Then Lemma 15 implies that properties E1 and E2 are satisfied. The bounds on running time and work follow from the bounds in Lemmas 17 and 18. Property E5 is satisfied because $\overline{G}(v_i)\ cap\overline{G}(v_j) \subseteq S(v)$, and in obtaining $\overline{G}(v)$ we add no edges between $\overline{G}(v_i)$ and $\overline{G}(v_j)$.

Finally we prove E6. For $a \in S(v)$ and $b \in G(v)$, let P be a shortest a-b path in $\overline{G}(v)$. Let c be the last node of P in $S(v)$. By property E3 the a-c subpath of P has at most $O(\log^2 N)$ hops and therefore at most $O(\log^3 N)$ edges. Let c' be the node following c in P. By property E5 the c'-b subpath of P is entirely in some $\overline{G}(v_i)$. Since c' is not in $S(v)$ and is adjacent to $c \in S(v)$, c must belong to the separator for $G(v)$. Hence c belongs to $S(v_i)$. By the inductive hypothesis the c-b subpath of P can be assumed to have size $O(h(v_i)\log^3 N)$. Since $h(v_i) = h(v) - 1$, the entire path has size $O(h(v)\log^3 N)$. This proves property E6. $\square$

Recall the splitting property we described when we introduced decomposition trees. Using a simple top-down induction using property E5 of Lemma 19, one can show that the decomposition tree has the splitting property with respect to the graphs $\overline{G}(\cdot)$.

Let G_0 be the original graph. It follows from property E1 of Lemma 19 that $V(G_0)$-distances in the graph $\overline{G}_0 = \overline{G}(\text{root})$ are $(1-\epsilon)^{O(\log^2 N)}$-accurate underestimates of $V(G_0)$-distances in G_0. We show that there are small shortest paths in $\overline{G}_0$ between nodes of $V(G_0)$.

Lemma 20 *For any nodes a, b of graph G_0, a shortest path in $\overline{G}_0$ exists that has $O(\log^4 N)$ edges.*

Proof: Let v_a be the lowest vertex in the decomposition tree such that $G(v_a)$ contains a, and similarly for v_b. Let v be the lowest common ancestor of v_a and v_b in the decomposition tree. Let P be a shortest a-b path in $\overline{G}_0$. The splitting property for the graphs $\overline{G}(\cdot)$ implies that P goes through a node c of $S(v)$. By applying property E6 to the a-c subpath and the c-b subpath, we obtain the lemma. $\square$

To find single-source shortest paths in G_0 we use the scaling algorithm of Section 3.1. For computing the prices required for the scaling algorithm we construct the auxiliary graph $\overline{G}_0$ as explained in Lemma 19. The shortest-paths in $\overline{G}_0$ are good underestimates of the shortest paths in G_0 and can be used as prices for the scaling algorithm. By Lemma 20 we also know that shortest paths in $\overline{G}_0$ have only $O(\log^4 N)$ edges. We therefore use the limited Bellman-Ford searching technique of Chapter 3 to compute $(1 - \frac{1}{2})$-approximate shortest paths. Using these as prices we get the bounds of Theorem 7.

4.4 Extending the algorithm to general graphs

To extend our parallel algorithm to general graphs we use a different version of Lemma 12 that applies to graphs that are not necessarily from minor-excluded families. Given a graph G_0 with m_0 edges we have the following lemma.

Lemma 21 *Let H be a summary of G_0 such that the segment graph of H has n nodes, m edges, and maximum edge-multiplicity 1. If the number of ordinary edges m is at least $8(m_0 + n)$ then $\Omega(\frac{m^3}{(m_0+n)^2})$ pairs of ordinary edges of H intersect.*

Note the difference between the statements of Lemmas 12 and 21. In Lemma 12 the number of intersections depends only on m and n while here it is also dependent on the number of edges m_0 in the underlying graph G_0. Thus Lemma 21 is weaker than Lemma 12. Using this crossing theorem we can find a near-cover of size $O((m_0+n)^2/m)$ in the same manner as before.

The proof of Lemma 21 is a little different from the original proof due to Leighton [68] used in Lemma 12. We therefore give the details of the proof in this section. In subsection 4.4.1 we present the proof of Lemma 21 and in Subsection 4.4.2 we show how to extend our parallel algorithm to general graphs.

4.4.1 Counting intersections in summaries of general graphs

Our proof of Lemma 21 uses the same inductive strategy as the proof by Leighton [68] for Lemma 12. However, the details of the various steps are somewhat different. Let $g(n, m)$ denote the number of crossings in in H. Recall that two edges of H are said to cross if their corresponding paths in G_0 share a common node. We define a crossing between edges ab and cd to be a *proper crossing* if the edges ab and cd are both ordinary edges and they do not share a common end point. In our proof we will show that the number of proper crossings in H obeys the stated bounds. In order to get an initial lower bound on the number of crossings we use the following proposition.

Proposition 3 *If H is a summary of G_0 with no proper crossings then the number of ordinary edges m in H is at most $5m_0$.*

Proof: Let $\hat{H}$ be the segment graph of H. Recall that each edge of $\hat{H}$ corresponds to an ordinary edge of H. Given an edge $e \in \hat{H}$ let π_e denote the corresponding path in G_0. First we construct a new graph I by removing from $\hat{H}$ all edges e such that the path π_e consists of only one edge. This results in a loss of at most m_0 edges. Let the number of edges in I be equal to m'.

We first randomly assign one of two colors (*blue* or *red*) to each of the ordinary nodes in I and construct a new graph I' by retaining only those edges of I that go from a red node to a blue node. The expected number of edges in I' is $m'/4$. By the probabilistic method of Erdös [90] we therefore know that there is a way to color the nodes of I such that at least $m'/4$ of the edges in I go from a red nodes to blue nodes. Consider one such coloring, and the corresponding subgraph I' of I containing only red-to-blue edges.

In what follows we show how to assign, for edge $e \in I'$ a unique edge $\hat{e} \in G_0$ from π_e, such that for any two edges e and f in I' the corresponding assigned edges $\hat{e}$ and $\hat{f}$ are not the same. Since there are only m_0 edges in G_0, by our assignment the number of edges in I' is at most m_0. Therefore, by construction of I' the number of ordinary edges in H is at most $5m_0$.

Given an edge $e \in I'$ we use one of the following three rules to find $\hat{e}$.

1. If there exists an edge $x \in \pi_e$ such that $x \notin \pi_f$ for any other $f \in I'$ then set $\hat{e}$ to be the earliest such edge in π_e.

2. Otherwise, if there is another edge $f \in I'$ such that such that f and e share the first endpoint and π_e and π_f share an edge then assign $\hat{e}$ to be the last edge on

π_e.

3. If neither of these conditions is met then set $\hat{e}$ to be the first edge on π_e.

We claim that for any two edges j and k in I' the corresponding assigned edges $\hat{j}$ and $\hat{k}$ are not equal. If $\hat{j}$ is neither the first nor the last edge on π_j then by definition it cannot be equal to $\hat{k}$. Otherwise suppose $\hat{j}$ and $\hat{k}$ are the same. We then have two cases.

Case I: $\hat{j}$ is the last edge in π_j:

Let a and b be respectively the tail and the head of the edge j. By the construction of I', node a is colored red and node b is colored blue. Also, by our assumption there are no proper crossings in I'. Therefore j and k share an endpoint. Furthermore, since $\hat{k} = \hat{j}$, the assigned edge $\hat{k}$ is either the first edge or the last edge in π_k. If it is the last edge then the common endpoint between j and k is b. If it is the first edge in k_{pi} then k goes from a' to b' (for some a' and b' in I') such that a' and b' are not equal to a or b. This follows from the following facts.

1. Both π_j and π_k are multi-edge paths; therefore $b' \neq b$ and $a \neq a'$.

2. The nodes a and a' are colored red while b and b' are colored blue. Therefore, $a \neq b'$ and $a' \neq b$.

This implies that j and k form a proper crossing which is a contradiction. Thus the j and k only share the second endpoint. Therefore k goes from some red colored node c that is different from a to the blue colored node b.

By choice of $\hat{j}$ we also know that there is an edge $f \in I'$ such that f and j share the first endpoint a and π_f and π_j share an edge in G_0. Pick an f that maximizes the distance of the shared edge α from the endpoint a. We now demonstrate the existence of an edge $g \in I'$ such that f and g form a proper crossing thus resulting in a contradiction.

If $\alpha = \hat{j}$ then we set g to be equal to k. Otherwise let β be the next edge on π_j. By choice of f, there exists another edge $g \in I'$ such that β is contained in π_g.

Furthermore, g and j share the second endpoint b. In other words g is an edge between some red node $h \neq a$ and the blue endpoint b of j. The claim that g and j share an endpoint follows from the assumption that I' has no proper crossings. The claim that the shared endpoint is b follows from the maximality in the choice of f. We now claim that g and f exhibit a proper crossing. We have two cases

- ($\alpha = \hat{j}$:) In this case $g = k$. Also, by the arguments above k and f do not share an endpoint. We therefore get a proper crossing.

- ($\alpha \neq \hat{e}$:) In this case f contains α, while g contains the next edge β of π_j. Therefore both π_g and π_f contain the common endpoint of α and β. Furthermore, from the arguments above we know that f and g do not share any endpoints. We therefore get a proper crossing.

This is a contradiction, therefore $\hat{j}$ is not equal to $\hat{k}$.

Case II: $\hat{j}$ is the first edge in π_j:

Suppose $\hat{k}$ is the same as $\hat{j}$. Then by choice of $\hat{k}$ it is either the first or the last edge in π_k. If it is the first edge on π_k then j and k share the first endpoint. This is a contradiction because by choice of $\hat{j}$ there is no edge f starting at the first endpoint of j such that π_f and π_j share an edge.

Suppose $\hat{j}$ is the last edge in π_k then $k = cd$ for some red node c and blue node d. We now claim that such that $c \neq a$ and $d \neq b$. This is because $\hat{j}$ is the first edge in π_j and neither π_j nor π_k is a single-edge path. By construction c and a are red, and b and d are blue. Therefore, $a \neq d$ and $b \neq c$. Thus j and k form a proper crossing which is a contradiction. Therefore $\hat{j} \neq \hat{k}$. The proposition thus follows. □

We are now ready to prove Lemma 21.

[Proof of Lemma 21]: We show that $g(n, m) \geq \Omega(\frac{m^3}{(m_0+n)^2})$ provided $m \geq 8(m_0 + n)$. In the remainder of the proof let $\alpha = m_0 + n$. Our first claim is based on Proposition 3.

$$g(n, m) \geq m - 7\alpha. \tag{4.2}$$

To prove our claim we use a simple induction proof. From Proposition 3 we know that if $m \geq 7m_0$ then at least one edge makes a crossing. After removing this edge we have at least $m - 7\alpha - 1$ crossings by induction. The total number of crossings is therefore $\geq m - 7\alpha$. Our claim therefore follows.

Our second claim is the following.

$$g(n,m) \geq \frac{3}{512}\binom{\alpha}{4}(\frac{m}{\binom{\alpha}{2}})^3, \text{whenever} m \geq 8\alpha \tag{4.3}$$

We consider two cases.

Case I: $8\alpha \leq m \leq 9\alpha$**:**

In this range we have

$$\begin{aligned}
\frac{3}{512}\binom{\alpha}{4}(\frac{m}{\binom{\alpha}{2}})^3 &= \frac{3}{512}\frac{\alpha(\alpha-1)(\alpha-2)(\alpha-3)}{1\cdot 2\cdot 3\cdot 4}\frac{m^3\cdot 2^3}{\alpha(\alpha-1)\alpha(\alpha-1)\alpha(\alpha-1)} \\
&< \frac{m^3}{512\alpha^2} \\
&\leq m - 7\alpha.
\end{aligned}$$

To show that $\frac{m^3}{512\alpha^2} \leq m - 7\alpha$, write $m = 8\alpha + k$, with $0 \leq k \leq \alpha$. Then $m - 7\alpha = \alpha + k$. We now show that $(8\alpha + k)^3 \leq 512\alpha^2(\alpha + k)$.

$$\begin{aligned}
(8\alpha + k)^3 &= 512\alpha^3 + k^3 + 192\alpha^2 k + 24\alpha k^2 \\
&\leq 512\alpha^3 + 217\alpha^2 k \\
&\leq 512\alpha^2(\alpha + k)
\end{aligned}$$

This establishes our induction basis.

Case II: $m > 9\alpha$**:**

Let us suppose that H has $g(H) = g(n,m)$ proper crossings. For each crossing between edges ab and cd, count the $n-4$ nodes $w \notin \{a,b,c,d\}$. the node w is counted $g(H(w))$ times, where $H(w)$ is the subgraph of H induced by the removal of w, and $g(H(w))$ is the number of proper crossings in $H(w)$. Let $E(w)$ denote the edge set of $H(w)$. Removal of w results in a loss of at most $n \leq (m_0 + n) = \alpha$ edges from H. Therefore, $|E(w)| \geq m - n > 8\alpha$, so we can use induction. For $H(w)$ let $\hat{\alpha} = m_0 + (n-1) = \alpha - 1$. Then by induction

$$g(H(w)) \geq \frac{3}{512}\binom{\hat{\alpha}}{4}\left(\frac{m}{\binom{\hat{\alpha}}{2}}\right)^3 = \frac{3}{512}\binom{\alpha-1}{4}\left(\frac{m}{\binom{\alpha-1}{2}}\right)^3$$

Therefore

$$\sum_{w \in H} g(H(w)) = \frac{3}{512}\binom{\alpha-1}{4}\frac{\sum_{w\in H}|A(w)|^3}{\binom{\alpha-1}{2}^3}.$$

Now,

$$\sum_{w \in H} |A(w)| = (n-2)m,$$

So

$$\sum_{w \in H} |A(w)|^3 \geq n\left(\frac{(n-2)m}{n}\right)^3.$$

Therefore,

$$\begin{aligned}
g(H) &\geq \frac{1}{(n-4)}\frac{3}{512}\binom{\alpha-1}{4} n\left(\frac{(n-2)m}{n}\right)^3 \frac{1}{\binom{\alpha-1}{2}^3} \\
&= \frac{3}{512}\binom{\alpha}{4}\frac{m^3}{\binom{\alpha}{2}^3}\frac{\alpha-4}{n-4}\frac{\alpha^3}{n^2\alpha}\frac{(n-2)^3}{(\alpha-2)^3} \\
&\geq \frac{3}{512}\binom{\alpha}{4}\frac{m^3}{\binom{\alpha}{2}^3}.
\end{aligned}$$

Lemma 21 therefore follows. □

4.4.2 Solving the shortest-path problem on general graphs

To solve the exact shortest-path problem on an m-edge graph G from source s we need to solve the $1-\frac{1}{2}$-approximate shortest-path problem. As in the nonnegative setting, we solve the problem by producing another graph G^* that approximates all-pairs shortest paths in G. To produce G^* we use procedure SMALLPATH on G. A modification of Lemma 18 shows that this requires $O(\text{polylog } m)$ time and $O((|G_0|+\epsilon^{-4}m^2) \text{ polylog } m)$ work. This results in a summary G^* with the following properties:

D1: $V(G)$-distances in G^* are $(1-\epsilon)^{\log_{4/3} m}$-accurate underestimates of $V(G)$-distances in G.

D2: For any ordinary nodes u, v, a shortest u-to-v path exists in G^* that consists of at most $O(\log^2 m)$ hops

D3: G^* has at most $2m$ supernodes.

Furthermore, since the edge-multiplicity (see Subsection 4.1.2) of the segment graph of G^* is polylogarithmic the number of edges in G^* is $O(m^2)$. Also if we augment the segments as described in Subsection 4.1.2, for any two nodes u, v of G, the shortest u-to-v path in G^* contains at most $O(\log^3 m)$ edges. Thus we can find single-source shortest paths in G^* in $O(\log^4 m)$ time and $O(m^2 \operatorname{polylog} m)$ work. Using this in combination with the scaling technique of Lemma 4 from Section 3.1 we get the bounds of Theorem 8.

4.5 Extending the parallel algorithm to handle negative edges

In this section we show how to extend our algorithm to a scenario when some of the edges are allowed to have negative weights. For this chapter we will only concentrate on the case when no negative-length edge appears in a directed cycle in the graph G.

Our parallel algorithm for the nonnegative case works by finding price functions that are underestimates of the actual shortest paths. These are then used to find reduced costs. If there are negative edges in the graph then underestimating the shortest paths may create negative cycles, thus making the problem infeasible, even if there were no negative cycles in the original graph.

However, if we restrict ourselves to graphs of the kind described above then we cannot inadvertently create negative cycles because the negative edges are not part of any directed cycle.

Our new algorithm also uses Gabow's scaling idea [34] to solve the problem by successive approximation (see Section 3.1). As discussed in [34] in order for this method to work the prices $d(\cdot)$ must obey the following property from I.

(I) For every edge $uv \in G$ the value $d(u) + \ell(uv)$ must be at least equal to $d(v)$.

Using these prices we set the reduced cost of the edge uv to be $d(u) + \ell(uv) - d(v)$. Note that the reduced cost of the edge uv is nonnegative even if the original cost $\ell(uv)$

was negative. Also, as proved in [34], the shortest paths from any source remain the same even with respect to the new lengths. Therefore, in order to solve the single source shortest path problem on a graph with negative edge weights we only need to find a set of prices $d(v)$ that satisfy condition (I). Using these prices we construct a graph where all the edge-weights are nonnegative. After that we can use our old algorithm that applies to graphs with nonnegative edge-weights.

To find the prices at the nodes of G we first make the weights of all nonnegative edges 0. Now our graph contains only nonpositive edge weights. Any price function that satisfies condition (I) for this graph also satisfies it for the original graph. Also note that zeroing $+ve$-weight edges does not create any negative-weight cycles since none of the negative edges occur in any directed cycles.

In the remainder of the section we show how to adapt our algorithm from the nonnegative setting to work for a graph with nonpositive edge weights. In order to make the algorithm work on such a graph, we need to make the limited Bellman-Ford searching technique of Section 4.1 work when the edges have nonpositive weights. In Subsection 4.5.1 we show how to change the procedures of Lemmas 9 and 11 to handle nonpositive edge-weights; and in Subsection 4.5.2 we show how to modify the remaining procedures to work with nonpositive edge weights. In the following let d^* be a distance parameter and let $\hat{d}^*$ be equal to $-d^*$.

4.5.1 Limited Bellman-Ford in a graph with nonpositive edge-weights

Given an n-node m-edge graph G in which all the edges have nonpositive edge-weights in order to find approximate r-limited shortest paths from a path $P = \eta_1 \ldots \eta_s$ we use the following modified version of Lemma 9.

Lemma 22 *There is a procedure that takes as inputs a graph G, a path P in G, a positive integer r, the distance bound $\hat{d}^* \leq 0$, and an error parameter ϵ. The procedure computes the following for each node v in G and each distance $0 \geq d \geq \hat{d}^*$ that is a multiple of $\epsilon\hat{d}^*$.*

(A) Either a node η in P such that

1. There is a v-to-η path of length at most $d + \epsilon d^*$, and

2. For every node η' such that there is a v-to-η' path Q of length $\geq \hat{d}^*$; either η' occurs after η on P or $\ell(Q) \geq d$.

(B) or ∞, if there is no v-to-P path with length between d and $\hat{d}^*$, and size at most r.

The procedure takes time $O(r \log m)$ and work $O(m\epsilon^{-1}r^2)$, where m is the number of edges in G.

Proof: If $\hat{d}^* = 0$ then for each node v the procedure finds the lowest node η such that there is a v-to-η path. It is easy to see that this satisfies conditions (1) and (2). To find η for all the nodes v we can perform a depth-first search (looking at paths of length $\leq r$) from the tail of P.

Otherwise, first the procedure discards edges of length $\leq \hat{d}^*$, as such edges obviously do not participate in paths of length $\geq \hat{d}^*$. Next it uses rounding to replace the edge-lengths with small integers. Let $\ell(vw)$ denote the length of edge vw, and define $\widehat{\ell}(vw) := \lfloor \ell(vw)/\alpha \rfloor$, where $\alpha = \epsilon d^*/(r+1)$. Let $i^* = -(r+1)\epsilon^{-1}$, and note that $\widehat{\ell}(vw) \geq i^*$.

Let the source-path be $P = \eta_1\eta_2 \dots \eta_s$. We can use the same procedure as in Lemma 9 that uses the lengths $\widehat{\ell}(\cdot)$. As described in Lemma 9 that procedure computes a table T_v of size $i^* + 1$ for every node v, satisfying the following *correctness condition*: for each negative $i \geq i^*$, $T_v[i]$ is the minimum index j such that there is a v-to-η_j path of length between i and i^* and size at most r. If there is no such index, $T_v[i] = \infty$.

We now show how it can be used to prove Lemma 22. For any distance $d \geq \hat{d}^*$ that is a multiple of $-\epsilon d^*$, let $i = \lceil d/\alpha \rceil$. For a node v, if $T_v[i]$ is ∞ then there is no v-to-P path of length between d and $\hat{d}^*$, and size at most r. If $T_v[i] = j$ then there is a v-to-η_j path that has $\widehat{\ell}$-length at most $\lceil d/\alpha \rceil$. Therefore the actual length of the path to η_j is no more than $\alpha\lceil d/\alpha \rceil + r\alpha \leq d + \epsilon d^*$. Hence η_j satisfies condition (1) of Lemma 22.

Consider a v-to-η' path Q of size r with length between $\hat{d}^*$ and d. If η' occurs before η then by construction

$$i < \sum_{vw \in Q} \widehat{\ell}(vw) = \sum_{vw \in Q} \lfloor \ell(vw)/\alpha \rfloor \leq \sum_{vw \in Q} (\ell(vw)/\alpha)$$

Substituting for i in the previous inequality we see that $d < \ell(Q)$. Hence η_j satisfies condition (2) of Lemma 22. $\square$

As in the nonnegative setting we can now use this procedure to construct a representation for k-hop paths that go through a given path P. To do that we use the following procedure which is a modification of the procedure in Lemma 11.

Lemma 23 *There is a procedure that, given an augmented summary H, the distance bound $\hat{d}^*$, an accuracy parameter ϵ, a path P in H, and a positive integer k, constructs a P-*substitute *for H, consisting of a* copy *P' of P and a set of* pseudoedges *between ordinary nodes and nodes of P', with the following properties.*

I. Each ordinary node has $O(\epsilon^{-1})$ pseudoedges to and from P'.

II. Each edge in P' has the same cost as the corresponding edge in P.

III. Each pseudoedge corresponds to a path in H and has length $1+\epsilon$ times the length of the path.

IV. For ordinary nodes v and w, if there is a v-to-w path of length $d \in [\hat{d}^*, \hat{d}^*/2]$ consisting of at most k hops and containing a node of P, then there is a v-to-w path of length at most d consisting of a pseudoedge, followed by a subpath of P', followed by a pseudoedge.

The procedure constructs the P-substitute in time $O(k \log^2 n_0)$ and work $O(m\epsilon^{-1}k^2 \log^3 n_0)$, where n_0 is the number of nodes in H.

Proof: As in the nonnegative setting the complexity of the procedure will depend upon the number of ordinary edges in H. By always considering relevant subgraphs as defined in Section 4.1 we can be sure that the number of edges under consideration is $O(m \log n_0)$. The procedure to construct the P-substitute is as follows. Apply the procedure of Lemma 22 to H' with $r = (2 + 2\log n_0)k$, distance bound $\hat{d}^*$, and error parameter $\bar{\epsilon} = \epsilon/8$. For each ordinary node v we construct $\bar{\epsilon}^{-1}$ outgoing edges, one for each distance $d \geq \hat{d}^*$ that is a multiple of $\bar{\epsilon}d^*$. The edge corresponding to distance d

is $v\eta$, where η is the copy in P' of the node of P satisfying (1) and (2) of Lemma 22, namely:

(1) there is a v-to-η path of length at most $d + \bar{\epsilon}d^*$, and

(2) For any v-to-η' path Q of length $\geq \hat{d}^*$ either η' occurs after η or $\ell(Q) > d$.

The length assigned to the pseudoedge is $1 + \epsilon$ times the length of the path it represents, namely the path of (1). For those d for which (Lemma 22,B) holds, there is no corresponding pseudoedge.

We have described the construction of outgoing pseudoedges. Incoming pseudoedges are constructed analogously. The time and work for this procedure are dominated by the time and work required by the two calls to the procedure of Lemma 22. The bounds of Lemma 23 therefore follow.

Now we show the construction satisfies the correctness properties. Properties (I) and (II) are immediate. Let Q be a v-to-w path of length between $\hat{d}^*$ and $\hat{d}^*/2$ consisting of at most k hops and containing a node η of P. Write $Q = Q_1\eta Q_2$, and let $d_1 = (\lceil \ell(Q_1)/\bar{\epsilon}d^* \rceil)\bar{\epsilon}d^*$, and $d_2 = (\lceil \ell(Q_2)/\bar{\epsilon}d^* \rceil)\bar{\epsilon}d^*$. Let $v\eta_1$ be the outgoing pseudoedge in the substitute that corresponds to distance d_1, and let $\eta_2 w$ be the incoming pseudoedge that corresponds to distance d_2.

Since $\ell(Q) \geq -d^*$ both d_1 and d_2 are $\geq \hat{d}^*$. This follows from the fact that all the weights in the underlying graph G are nonpositive. Also by definition $\ell(Q_1) \leq d_1 \leq \ell(Q_1) + \bar{\epsilon}d^*$ and $\ell(Q_2) \leq d_2 \leq +\ell(Q_2) + \bar{\epsilon}d^*$. Therefore, by condition (2) of Lemma 22 (as shown in Figure 4.2) η occurs no earlier than η_1 and no later than η_2 on P. Thus η_1 occurs no later than η_2 on P. Also by condition (1) of Lemma 22 length of the v-to-η_1 path is at most $d_1 + \bar{\epsilon}d^*$ and the length of η_2-to-w path is no more than $d_2 + \bar{\epsilon}d^*$.

Let P'' be the subpath of P' from η_1 to η_2. The path consisting of the pseudoedge $v\eta_1$ followed by P'' followed by $\eta_2 w$ has length at most

$$(1 + \epsilon)[d_1 + d_2 + 2\bar{\epsilon}d^* + \ell(P)] \leq (1 + \epsilon)[\ell(Q_1) + \ell(Q_2) + 4\bar{\epsilon}d^* + \ell(P)]$$

Since $\ell(P) \leq 0$ the length of the new path is at most $(1+\epsilon)[\ell(Q)+4\bar{\epsilon}d^*]$. By assumption $\ell(Q) \leq \hat{d}^*/2$. Therefore, substituting the value of $\bar{\epsilon}$ we see that the length of the new path is at most $\ell(Q)$. □

We can now use this new limited Bellman-Ford algorithm in much the same way as before to find approximate single-source shortest paths when the underlying graph has all nonpositive edge-weights. In Subsection 4.5.2 we explain these steps.

4.5.2 Finding approximate shortest paths with nonpositive edge weights

We follow the same sequence of subroutines as in the algorithm for the nonnegative case. The subroutines themselves are also essentially the same with minor modifications. We now state the same sequence of subroutines, as in Section 4.3, with the necessary modifications. The proofs are identical to those in Section 4.3 and are omitted.

Lemma 24 *Consider two graphs, A and B with nonpositive edge weights, where $V(A) \cap V(B) = V'$. Suppose that V'-distances in B are $(1+\epsilon_2)$-accurate underestimates of V'-distances in A, and, for some $S \subset V(A)$, S-distances in A are $(1+\epsilon_1)$-accurate underestimates of some other set of nonpositive distances δ. Then*

1. *S-distances in $A \cup B$ are $(1+\epsilon_1)(1+\epsilon_2)$-accurate underestimates of δ, and*

2. *V'-distances in B equal V'-distances in $A \cup B$.*

Proof: Follows immediately from the definitions. □

We now describe the procedures leading up to the two new versions of Compact and SmallPath called Compact2 and SmallPath2 respectively. We will state most of the lemmas without proof, pointing out only the differences from the previous setting. As in the nonnegative setting G_0 will denote the underlying sparse graph, and G will denote the subgraph of G_0 under consideration. N and n will denote the number of nodes in G_0 and G respectively. We also assume for the sake of convenience that the edge weights are polynomially bounded in N.

In all of these procedures the underestimates will not be accurate underestimates unlike in the nonnegative case. For our purposes it is enough that they are underestimates; and that the price function we finally derive satisfy the shortest-path inequality (see inequality (I) at the beginning of the section).

Lemma 25 *There is a procedure that, given the distance bound $\hat{d}^*$ and a summary H such that no ordinary edge in H has length less than $\hat{d}^*$, produces a summary $\widehat{H}$ with the following properties.*

A1: $V(H) \cap V(\widehat{H})$ *is the set of ordinary nodes of* H.

A2: $\widehat{H}$ *has at most* $\max\{0, n/4 \log N - cn^2/\epsilon^2 m\}$ *segments, where c is a constant.*

A3: For $u, v \in V_o(H)$, if there is a k-hop u-to-v path of length d in H such that $\hat{d}^* \leq d \leq \hat{d}^*/2$, then the one-hop u-to-v distance in $\widehat{H}$ is at most d.

The procedure runs in time $O(k \operatorname{polylog} N)$ *and requires work* $O((|G| + k^2\epsilon^{-3}n^2) \operatorname{polylog} N)$ *with high probability.*

The proof is the same as that for Lemma 16 except that in this case the near-cover paths need not be split into smaller segments because the edge weights are nonpositive. The running time therefore depends on ϵ^{-3} instead of ϵ^{-4}.

Corollary 4 *There is a procedure* ABBREVIATE2 *that from a summary H produces a summary $\widehat{H}$ with the following properties.*

B1: $V(H) \cap V(\widehat{H}) = V'$.

B2: $\widehat{H}$ *has at most $n/4$ segments.*

B3: For $u, v \in V'$, if the k-hop u-to-v distance in H is d, then the one-hop u-to-v distance in $\widehat{H}$ is at most d.

The procedure runs in time $O(k \operatorname{polylog} N)$ *and requires work* $O((|G|+k^2\epsilon^{-3}n^2) \operatorname{polylog} N)$ *with high probability.*

Again the proof is same as before. For each i the procedure applies the algorithm of Lemma 25 to the subgraph H_i of H that contains only edges with weight $\geq -D/2^i$. The algorithm is applied with the distance bound $d^* = D/2^i$ (and $\hat{d}^* = -d^*$). Thus the substitute found in the ith stage approximates paths in the range $[-D/2^i, -D/2^{i+1}]$.

Lemma 26 *There is a procedure* COMPACT2 *that, given a summary H and a subset S of the ordinary nodes of H, produces a summary H_S that with high probability has the following properties.*

C1: S-distances in H_S are underestimates of S-distances in H.

C2: H_S has at most $8|S|$ supernodes.

The procedure runs in time $O(\text{polylog } N)$ and work $O((|G| + \epsilon^{-3}n^2)\,\text{polylog } N)$ with high probability.

Again the proof is the same as in Lemma 17 except that we now use procedure ABBREVIATE2 instead of procedure ABBREVIATE.

Lemma 27 *There is a procedure* SMALLPATH2 *that, given a summary H with n super nodes, produces a summary H^* that with high probability has the following properties.*

D1: $V_o(H)$-distances in H^* are underestimates of $V_o(H)$-distances in H.

D2: For any ordinary nodes u, v, a shortest u-to-v path exists in H^* that consists of at most $O(\log N \log n)$ hops.

D3: H^* has at most $2n$ supernodes.

The procedure runs in time $O(\text{polylog } N)$ and work $O((|G| + \epsilon^{-3}n^2)\,\text{polylog } N)$ with high probability.

4.5.3 Constructing a graph with small diameter when the underlying edges have nonpositive weight

Following the same strategy as before, we use procedures ABBREVIATE2 and SMALLPATH2 to construct a graph $\overline{G}_0$ where shortest paths have few edges and that underestimates the shortest paths in G_0. As before we make use of the decomposition tree.

Note that the shortest-paths in $\overline{G}_0$ are not necessarily accurate underestimates of the shortest paths in G_0. However, as in the nonnegative setting these estimates obey the shortest-path inequality (inequality (I)) in G_0 and therefore give us a valid price function, which is what we want. We therefore have the following lemma.

Lemma 28 *For a vertex v of the decomposition tree, we can compute a graph $\overline{G}(v)$ and a summary $H^*(v)$ with the following properties.*

- *E1: $V(G(v))$-distances in $\overline{G}(v)$ are underestimates of $V(G(v))$-distances in $G(v)$.*
- *E2: $S(v)$-distances in $H^*(v)$ equal $S(v)$-distances in $\overline{G}(v)$.*
- *E3: For any $a, b \in S(v)$, a shortest a-b path in $H^*(v)$ exists that has $O(\log^2 N)$ hops.*
- *E4: The number of supernodes in $|H^*(v)|$ is $O(|S(v)|)$.*
- *E5: For any two children v_i and v_j of v in the decomposition tree, any path in $\overline{G}(v)$ between $\overline{G}(v_1)$ and $\overline{G}(v_2)$ must contain a node of $S(v)$.*
- *E6: For any $a \in S(v)$ and $b \in V(G(v))$, if there is an a-b path in $\overline{G}(v)$, there is a shortest path of size $O(h(v) \log^3 N)$.*

The time required is $O(\text{polylog } N)$, and the work is $O(\epsilon^{-3}|G(v)| \text{ polylog } N)$.

The graph $\overline{G}_0$ gives us good underestimates for the shortest paths in G_0. As in the earlier case we can find these shortest paths by using the limited Bellman-Ford search of Chapter 3. We thus get the prices necessary to turn the problem from one involving both nonnegative and negative weights to one involving only nonnegative weights. We can now use the old algorithm to find single-source shortest paths. In a similar fashion we can also extend our algorithm for general graphs to the case when negative edges are present. We therefore get the bounds of Theorem 9. We have the following immediate corollary for computing longest paths in dag.

Corollary 5 *There exists a procedure that given an m-edge dag with nonnegative edge weights, finds the longest paths from a source to all the other sinks. The procedure*

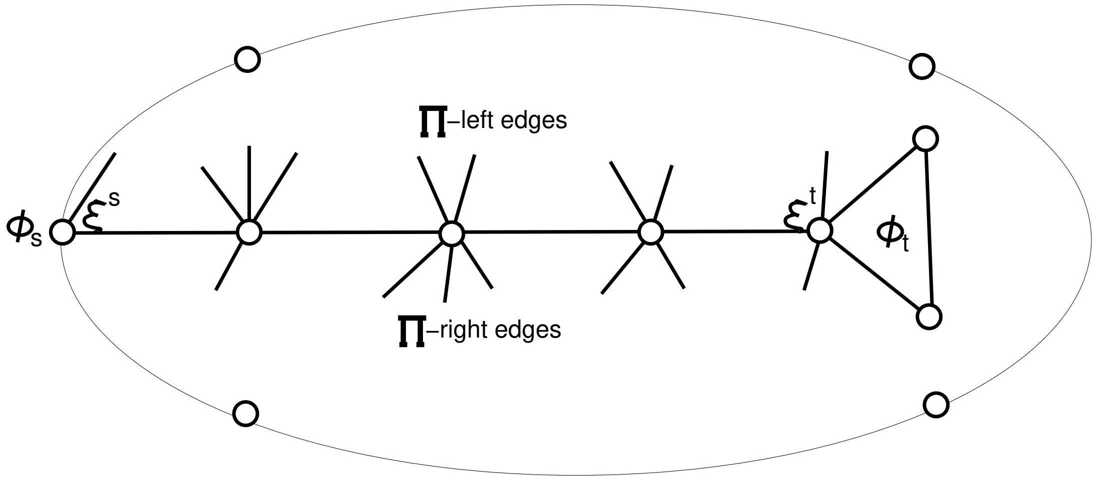

Figure 4.4: A partial rendition of the dual network (from [49]). The Π-left edges are above the ξ^s-to-ξ^t path and the Π-right edges are below it.

runs in time $O(\text{polylog } m \log L)$ *(with probability* $1 - m^{-\Omega(1)}$*) and uses* m^2 *processors. For* n*-node planar dags there exists a procedure that runs in time* $O(\text{polylog } n \log L)$ *(with probability* $1 - n^{-\Omega(1)}$*) and uses* n *processors. Here* L *denotes the maximum edge weight.*

Proof: By negating all the edges we can change the problem to one of finding shortest paths in a dag with nonpositive edge weights. Using the algorithm above we can therefore find these paths. □

4.5.4 Computing a maximum flow in an undirected planar graph

Given an undirected planar network with nonnegative edge capacities we can find a maximum st-flow from a source s to a sink t by using the shortest path algorithm described above. In this subsection we detail this procedure. Hassin and Johnson [49] have given an algorithm for computing the maximum flow in a planar undirected network by computing shortest paths in the dual network. We will use their results to get our bounds. We give a brief sketch of their algorithm here. For details see [49]. Their max-flow algorithm uses the following steps:

1. Let ξ^s be a vertex on the dual face (in the dual network) corresponding to the source node s. Similarly let ξ^t be a node on the dual face corresponding to the sink t.

2. Perform a single-source shortest-path computation from ξ^s in the dual network.

3. Let Π be the shortest path from ξ^s to ξ^t. Further, let $\xi_1, \xi_2, \cdots, \xi_k$ be the nodes on Π, where $\xi_1 = \xi^s$ and $\xi_k = \xi^t$.

4. Define any edge e incident to some node ξ_i to be a Π-*left edge* if it is to the left of Π in the embedding (while traveling from ξ^s to ξ^t). Otherwise define it to be a Π-*right edge* if it is to the right (see Figure 4.4).

5. For every node ξ_i on Π find a *minimum ξ_i-cycle* in the dual network. A minimum ξ_i-cycle is a shortest cycle that uses exactly one Π-left edge and exactly one Π-right edge. Each of these cycles forms a cut in the primal graph. Let $v_{\max}$ be the value of the minimum among all these cycles. The value $v_{\max}$ is the value of the minimum cut as well as the value of the maximum flow in the primal graph.

   These cycles can be found in polylog time with linear processors using our shortest path algorithm for nonnegative edge graphs along with a divide-and-conquer min-cut algorithm due to Reif [87].

6. Transform the dual network by making two copies of the path Π and adding directed edges from the right copy ξ_i'' to the left copy ξ_i' of every node ξ_i. The weight of the edge $\xi_i''\xi_i'$ for all ξ_i is set to $-v_{\max}$. These are the only directed edges in the graph. Let G_d denote this transformed dual network.

7. Let ξ_r be a node on Π such that the value of the minimum ξ_r cycle is $v_{\max}$. Find single-source shortest paths from ξ_r' to all the other nodes in G_d.

8. Let $d(\cdot)$ be the shortest path distances from ξ_r. Obtain a maximum flow in the primal graph by setting the flow in the edge uv to be equal to $d(v^t) - d(u^t)$ where $u^t v^t$ is the dual edge corresponding to the primal edge uv.

Lemma 29 (Hassin and Johnson) *The flow computed by the above algorithm is legal and is a maximum st-flow.*

The only computation involving negative edges is the shortest-path computation from ξ_r' in the transformed dual network. The new directed edges all have weight $-v_{\max}$. Hassin and Johnson show that in the transformed dual network G_d there are no negative cycles. However, G_d, as defined, does not satisfy our requirement that no negative edge appear in any directed cycle. We now show that it can be further transformed to a network where such a condition holds.

The ξ-minimum cycles found by the algorithm of Hassin and Johnson are all nested and they divide the graph a sequence of concentric bands. Let $C_1, C_2, \cdots, C_k$ be the minimum ξ-cycles of $\xi_1, \xi_2, \cdots, \xi_k$, and let $N_0, N_1, N_2, \cdots, N_k$ be the bands of the dual network between these cycles. The band N_i is the subnetwork bounded by and including C_i. It does not include C_{i+1}. The band N_0 includes everything that is outside C_0 while the band N_k includes C_k and everything that is inside it. See Figure 4.5 for an example.

Hassin Johnson [49] show that the shortest paths in the transformed dual network have a special property.

Definition 7 A path P from ξ_r' to a node v in N_i is called a *normal path* if it can be divided into subpaths $P_r, P_{r+1}, \cdots, P_q, P_{q-1}, P_{q-2}, \cdots, P_{2q-i}$ such that the subpath P_j is in band N_j, and the subpaths $P_{q-1}, P_{q-2}, \cdots, P_{2q-i}$ do not use any negative edges. Thus in a normal path the negative edges are used only when the path traverses from N_j to N_{j+1} (see Figure 4.5).

Lemma 30 (Hassin and Johnson) *For any node v in the band N_i there exists a ξ_r'-to-v shortest-path that is normal.*

By Lemma 30 negative edges are used only when a shortest path traverses from band N_j to band N_{j+1}. Also, every shortest path first traverses into concentric bands once and then comes out. It does not go back and forth. We will therefore find the shortest paths in two stages. The first stage finds the portion of the shortest paths that traverse into the concentric bands and the second stage is used to find the portion

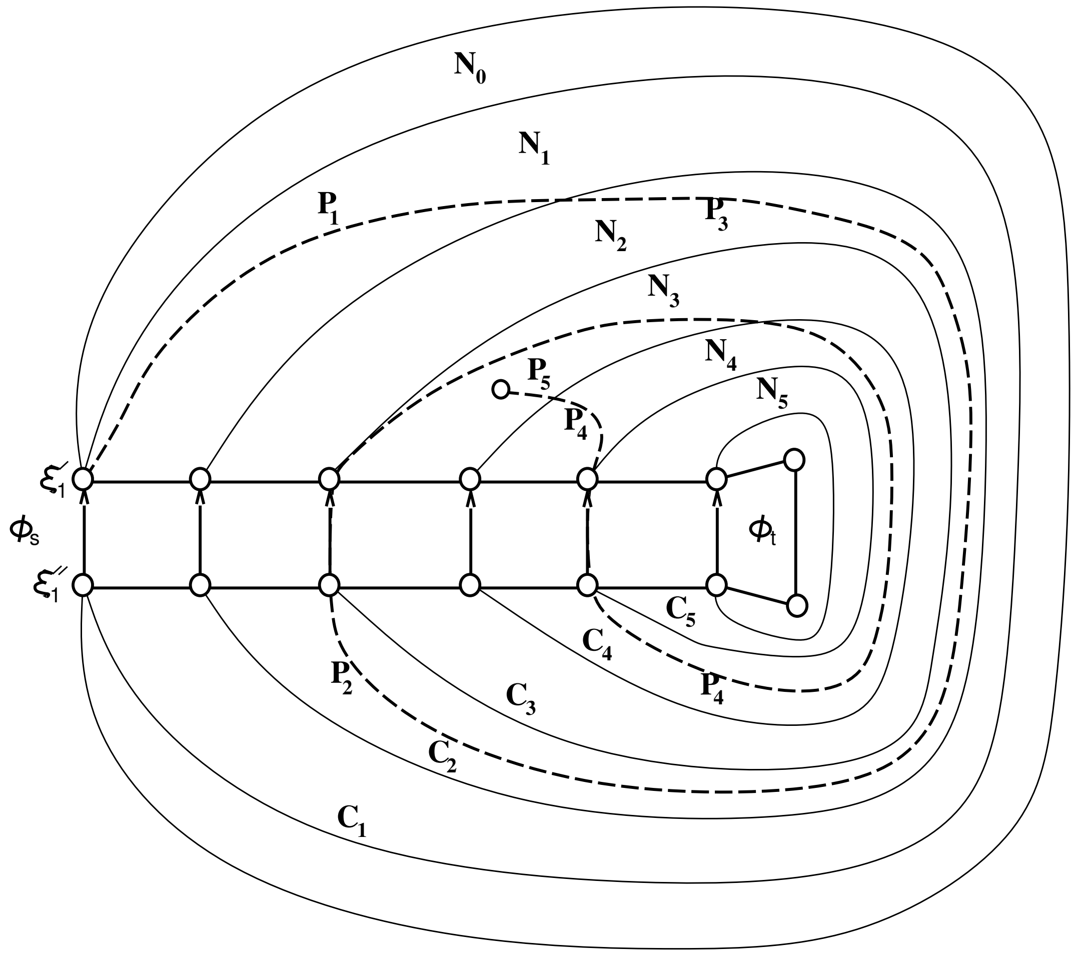

Figure 4.5: The bands N_0 through N_k induced by the minimum cycles C_1 through C_k (from [49]). The figure also shows a normal shortest path from $\xi_1^{'}$ to v.

of the paths that come back out through the bands. To find the portion of the paths that traverses inwards we modify the dual network in the following manner.

1. In the dual network G_d, we duplicate the nodes on the cycles C_1 through C_k. In other words, for each node x on a cycle C_j we create two copies x_1 and x_2.

2. For every node x on C_j we assign the edges of N_{j-1} to the first copy x_1 and the nodes of N_j to the second copy x_2.

3. We construct a directed edge of weight zero from the first copy x_1 to the second copy x_2.

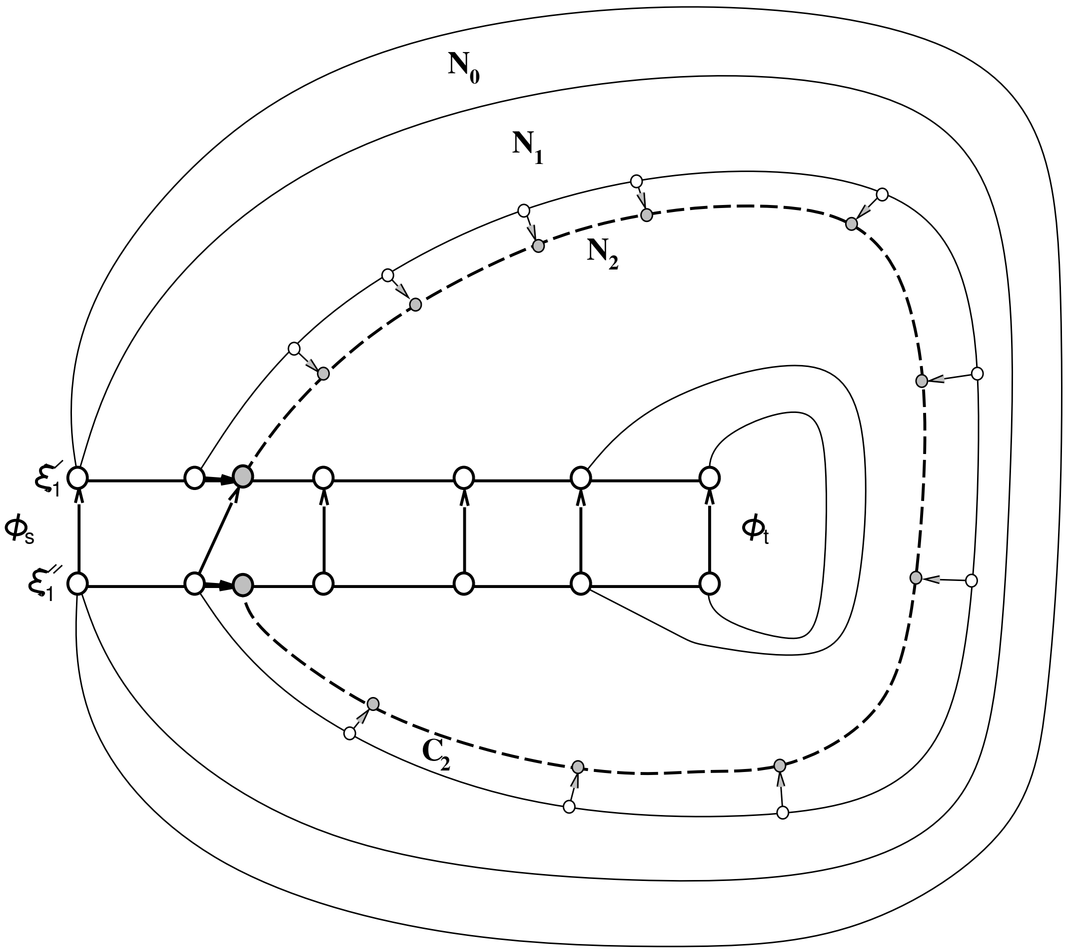

Figure 4.6: Modifying the dual network in order to remove negative edges from directed cycles. The figure shows the duplication of the cycle C_2. The dashed edges and the shaded nodes indicate the second copy.

4. The negative weight edge along C_j from ξ_j' to ξ_j'' is replaced by an edge with the same weight from the first copy of ξ_j' to the second copy of ξ_j''.

Figure 4.6 shows an example of this construction.

Lemma 31 *In the modified network G_d the negative weight edges do not appear in any directed cycle. Furthermore, all normal single-source shortest-paths from ξ_r' that travel only into the concentric bands (without coming back out) are preserved by the transformation.*

Proof: The first claim follows from the fact that all edges that connect nodes in two different bands go from a lower level band N_j to the higher level band N_{j+1}. Thus any cycle is confined to a single band. Since the negative-weight edges go from a lower level band N_j to a higher level band N_{j+1} they cannot be part of any cycle.

The second claim follows by construction since all paths that travel inwards are preserved by the modification. □

By running our shortest-path algorithm on this modified dual network we can find all normal shortest-paths that go only in the forward direction. In order to find the remaining portion of the shortest paths. We use the dual network with the negative edges removed. After removing the negative edges we add a new source node s and add directed edges from s to all the edges in G_d with weights equal to the shortest paths found in the first stages. We then find shortest paths from s to all the other nodes to get our final answer.

Since s has no incoming edges, the new edges added are not part of any cycle. Therefore the new graph also has the property that no negative edges are involved in any cycles. We can therefore use our algorithm to find shortest paths for the second stage as well. We therefore have the following theorem.

Theorem 11 *There exists a procedure that given an undirected n-node planar network with nonnegative edge-capacities $\leq L$, finds the maximum st flow from the source s to the sink t. The procedure runs in time $O(\text{polylog } m \log L)$ (with probability $1 - m^{\Omega(1)}$) and uses n processors.*

Chapter 5

An efficient parallel algorithm for shortest paths in planar layered digraphs

The parallel algorithm of Chapter 4 applies to a large class of graphs, and though efficient in an asymptotic sense it has a polylogarithmic running time with a large exponent. In this chapter we focus on a special class of planar graphs called planar layered digraphs and show that for this class of graphs we can develop a parallel algorithm that is much more efficient than the general algorithm presented in Chapter 4.

Efficient parallel algorithms for computing shortest paths have been devised for two special classes of planar digraphs: series-parallel digraphs and grid digraphs. A *series-parallel digraph* [97] is an acyclic digraph with exactly one source and exactly one sink that is recursively constructed by series and parallel compositions. In a series-parallel digraph the weight of a shortest path between the source and the sink is obtained by evaluating an arithmetic expression with operators + (associated with series compositions) and min (associated with parallel compositions), which can be done optimally in $O(\log n)$ time with $n/\log n$ processors [17].

A grid digraph has the vertices arranged in a rectangular grid and the edges directed

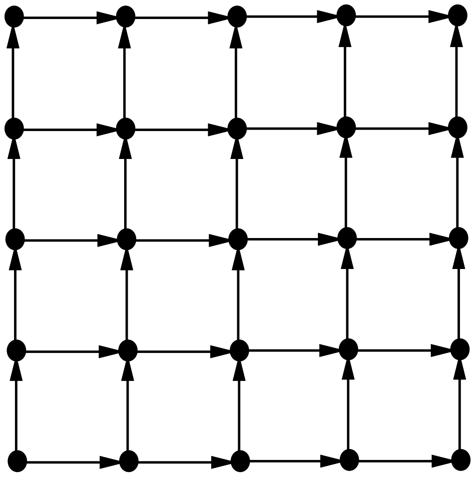

Figure 5.1: An example of a grid digraph.

from left to right and from bottom to top (an example is shown in Figure 5.1). Apostolico et al. [5] gave an algorithm to compute shortest paths in a grid in $O(\log^2 n)$ time with $O(n)$ processors. The shortest path problem on grid digraphs has applications in text processing, biological research, tomography and medical diagnosis.

In this chapter we consider a class of digraphs that extends grid digraphs, namely, *planar layered digraphs*. In a planar layered digraph the vertices are arranged along parallel lines, called layers, and edges connect vertices of consecutive layers and do not intersect. We present an efficient parallel algorithm for the shortest path problem in such digraphs that runs in $O(\log^3 n)$ time with n processors. The algorithm uses a divide and conquer approach and is based on the novel idea of a *one-way separator*, which has the property that any directed path can cross it only once. Our algorithm runs in the CREW PRAM model wherein different processors are allowed to concurrently read from the same location but are not allowed to concurrently write to the same location.

A CRCW version of this algorithm runs in time $O(\log^2 n \log\log n)$ and uses $n/\log\log n$ processors. We also show how to improve our algorithm by using some techniques of [1] and [6] to improve the time bound to $O(\log^2 n)$ in the CREW

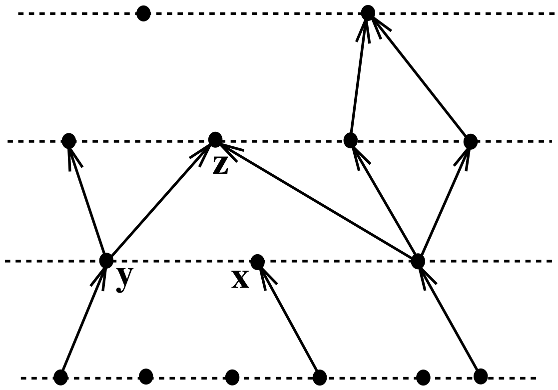

Figure 5.2: A planar layered digraph.

model and $O(\log n \log\log n)$ in the CRCW model.[1] The processor bounds still remain as $n/\log n$ for the CREW model and $n/\log\log n$ for the CRCW model.

The rest of this chapter is organized as follows. In Section 5.1 we formally define planar layered digraphs and introduce the notion of one-way separators. In Section 5.2 we prove the existence of a one-way separator for planar layered digraphs and show how to use such a separator to design an efficient divide-and-conquer shortest path algorithm. In Section 5.3 we discuss how our results relate to the previous one for grid digraphs. Finally, in Section 5.4 we present our conclusions and comment on the open problem of efficiently finding shortest paths in planar *st*-graphs.

5.1 Preliminaries

Let $G = (V, E)$ be a directed acyclic graph with n vertices and m edges, we say that G is a layered digraph if V is partitioned into p subsets, called *layers*, $l_1, \ldots, l_k$ such that all edges of G are between consecutive layers. Given p parallel lines consecutively numbered in the plane, a *p-line embedding* of G is a drawing such that

- All vertices of layer l_i are drawn on line i;
- Edges are drawn as straight lines

[1]We would like to thank Alok Aggarwal for pointing this out to us.

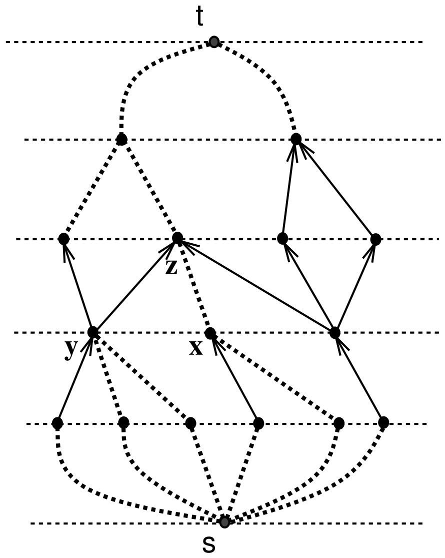

Figure 5.3: Adding extra edges to create a planar layered st-graph.

A *planar p-line embedding* is a p-line embedding without crossings. G is a planar layered digraph if it admits a planar p-line embedding. Figure 5.2 gives an example of a layered graph with a planar 5-line embedding. Layered graphs have been studied under the name of *proper hierarchies* in [101]. Di Battista and Nardelli [20] give efficient algorithms to test if a layered digraph with only one source is planar. Kosaraju [19] has developed an efficient parallel algorithm to evaluate monotone planar layered circuits. Recently Ramchandran and Yang [86] have developed a linear-processor polylog-time algorithm for evaluating general monotone planar circuits.

We can transform any planar layered digraph G into a planar layered digraph with exactly one source and one sink in the following manner: We first introduce two new nodes s and t and put them in two new layers, one at the beginning and one at the end. Then, for each node x in layer i that is a sink we find the closest node y to its left that is attached to a node in layer $i+1$. Let z be the right-most node in layer $i+1$ that has an incoming edge from y. If there is no such node y then we set z to be the left-most node in layer $i+1$. To make x a non-sink node we introduce an infinite weight edge from x to z. For example, Figure 5.3 shows how to transform the planar

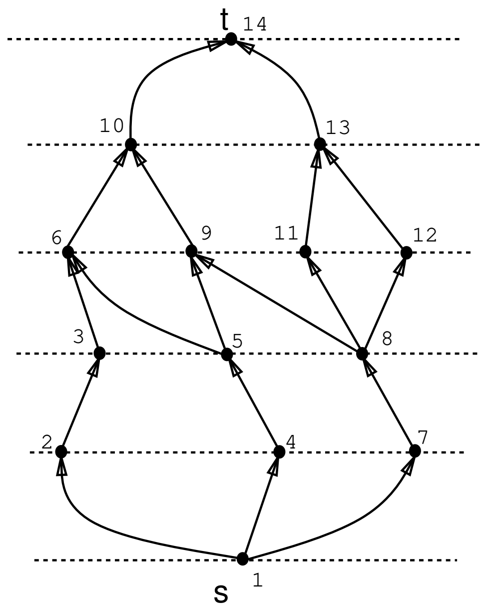

Figure 5.4: Left ordering of a planar st-graph.

layered digraph in Figure 5.2 into a single-source single-sink planar layered digraph. It is not hard to see that these infinite-weight edges do not alter any shortest paths or destroy planarity. This computation can be accomplished in $O(\log n)$ time with n processors using standard techniques (see for example [60]). A planar layered digraph with exactly one source s (on the first layer) and one sink t (on the last layer) is called a planar layered st-graph. In the rest of the paper we solve the problem of finding the shortest path between the source and the sink of a planar layered st-graph. To find the shortest path between any two vertices u and v in G we perform a preprocessing step to remove all the vertices which are not on any path between u and v. A brief outline of the algorithm follows.

1. Let T_u be the set of vertices that are on some path from u to t in G.

2. Let S_v be the set of vertices that are on some path from s to v in G.

3. Discard all the vertices in G that are not in T_u or S_v.

Theorem 12 *The algorithm outlined above can be implemented in $O(\log n)$ time with $n/\log n$ processors in an EREW PRAM.*

Proof: All the steps can be done in $O(\log n)$ time using techniques in [94]. □

A planar layered st-graph is a special case of a planar st-graph which is defined as a planar acyclic digraph with exactly one source, s, and exactly one sink, t, embedded in the plane so that s and t are on the boundary of the external face. These graphs were first introduced in the planarity testing algorithm of Lempel *et al.* [70].

We now define the concept of a *left* ordering of the vertices in a planar st-graph, which will prove useful in our algorithm. This ordering was introduced by Tamassia and Preparata [92]. We do this by making use of the dual graph of G (labeled G^*) defined as follows.

1. Every internal face f in G corresponds to a vertex in G^*.
2. The dual edge e^* of an edge e is directed from the face to the left of e to the face to the right of e.
3. The external face of G is dualized to two vertices of G^*, denoted s^* and t^*, which are incident with the duals of the edges on the leftmost and rightmost paths from s to t, respectively.

Definition 8 For every $x \in V$ we denote by $\mathit{left}(x)$ and $\mathit{right}(x)$ the two faces that separate the incoming and outgoing edges of a vertex $x \neq s, t$. For $x = s$ or $x = t$, we conventionally define $\mathit{left}(x) = s^*$ and $\mathit{right}(x) = t^*$.

Definition 9 We say that x is *below* y, denoted $x \uparrow y$, if there is a path in G from x to y. Also, we say that x is *to the left of* y, denoted $x \rightarrow y$, if there is a path in G^* from $\mathit{right}(x)$ to $\mathit{left}(y)$.

Definition 10 The left ordering (denoted by $<_l$) is defined on the basis of the relations "left" and "below". A vertex x occurs before another vertex y in the ordering if either $x \rightarrow y$ or $x \uparrow Y$.

A planar st-graph with its vertices numbered according to the *left ordering* is shown in Figure 5.4. Tamassia and Vitter [94] give optimal EREW algorithms to construct the left ordering of a planar st-graph.

5.2 Shortest paths in a planar layered st-graph

Let $G = (V, E)$ be a planar layered st-graph with source s and sink t. Given G with its associated embedding in layers, in this section, we show how to determine the shortest path from s to t in G. The crux of the algorithm lies in finding one-way separators (defined in Section 1.4 of Chapter 1). We show how to find one-way separators X_1, X_2, and X_3 such that using three one-way divisions we can partition G into at most four pieces A, B, C, and D such that none of pieces has more than two-thirds of the nodes in G. We then show how this division can be used to formulate a recursive solution to the single-source shortest-path problem. To show the existence of such separators we need the following lemma, which follows directly from the arguments in [73]:

Lemma 32 *Let G be any n-vertex planar layered st-graph containing layers 1 through p. Let S_i denote the set of vertices in the ith layer, and let n_i denote the size of the set S_i. Also let $p+1$ be an additional layer containing no vertices. Given any layer j there exists a layer $i \leq j$ such that $n_i + 2(j-i) \leq 2\sqrt{t_j}$, where t_j denotes the number of vertices in layers 1 through j. Similarly there exists a layer $k \geq j$ such that $n_k + 2(k-j-1) \leq 2\sqrt{n-t_j}$. In particular, we let i (respectively k) be the layer which minimizes the function $n_i + 2(j-i)$ (respectively $n_k + 2(k-j-1)$).*

Theorem 13 *Given an n-vertex planar layered st-graph G containing layers 1 through p, there exist one way separators X_1, X_2, and X_3 such that the separator X formed by unioning X_1, X_2, and X_3 is an $O(\sqrt{n})$ separator that 2/3-splits G into at most four pieces.*

Proof: Given G, let j be the smallest layer such that the layers 1 through j have at least $n/2$ vertices. Let i and k be layers as in Lemma 32. We let X_1 be the set of nodes S_i in layer i, and X_2 be the set of nodes S_k in layer k.

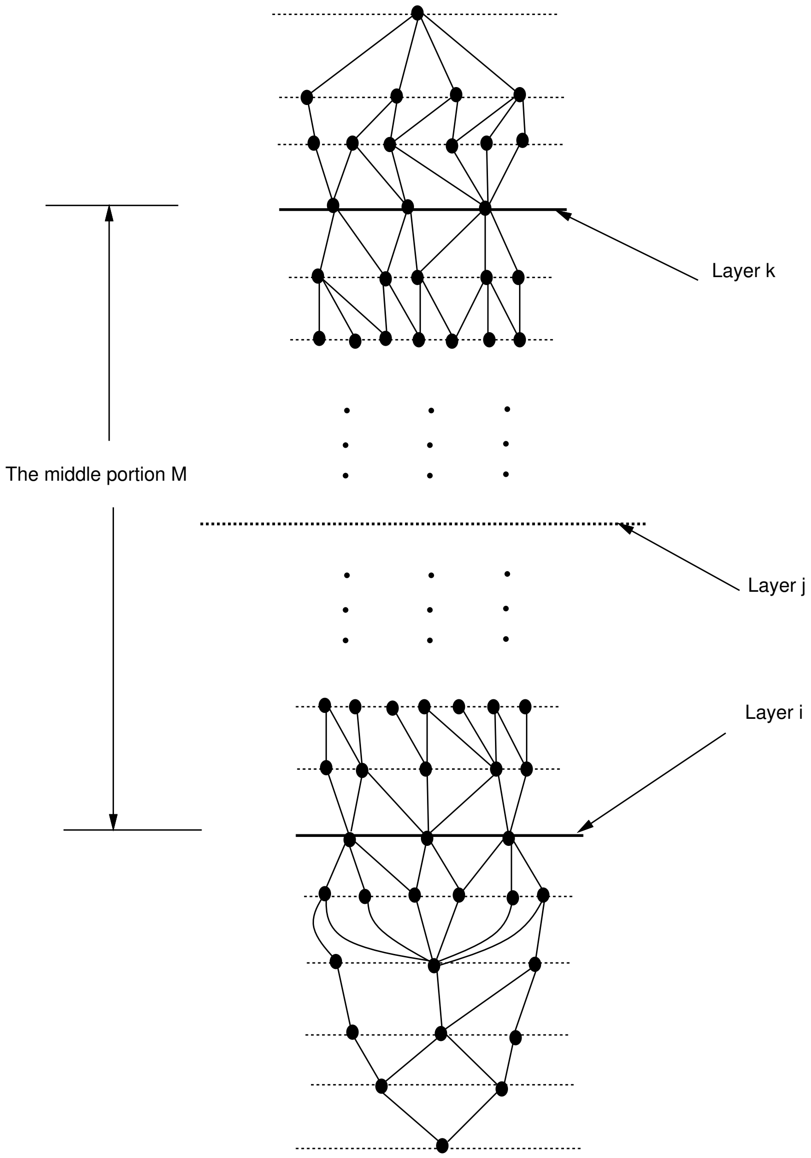

Figure 5.5: Slicing the graph into three parts.

Let A be the part of G above k, let B the part below i, and let M be the remaining part in the middle (as shown in Figure 5.5). By construction A and B each have less than $2n/3$ vertices. If the number of vertices in M is less than $2n/3$, we let M be the third portion C and set the fourth portion D to be equal to ϕ. Otherwise we form a new planar layered st-graph G' from the middle part M and two new vertices s' and t', adding edges from all the vertices in layer k to t' and from s' to all the vertices in layer

i (see Figure 5.6). Let m be the middle vertex in the left ordering L_1 of G', and let q be a path constructed by taking the leftmost outgoing edge from m until reaching t', and the rightmost incoming edge from m until reaching s' (see Figure 5.6). The path q splits G' into two equal-sized parts C and D. We let X_3 be the set of nodes on q.

We claim that X_1, X_2, and X_3 are one-way separators and that the separator $X_1 \cup X_2 \cup X_3$ is a $O(\sqrt{n})$ separator that 2/3-splits G. By construction none of the pieces have more than $2n/3$ vertices. Therefore, G is 2/3-split by X. Also, $|q| \leq k-j+1$ therefore

$$\begin{aligned}
|X| &\leq |X_1| + |X_2| + |X_3| \\
&= n_i + n_k + |q| \\
&\leq n_i + n_k + 2(j-i) + 2(k-j-1) \\
&\leq 2(\sqrt{t_j} + \sqrt{n-t_j}) \\
&\leq 2(\sqrt{n/2} + \sqrt{n/2}) \\
&= 2\sqrt{2n}.
\end{aligned}$$

To prove that X_1, X_2, and X_3 form one-way separators we note the following. By the definition of planar layered digraphs no path in G intersects with vertices in S_i or S_k more than once (see Figure 5.5) therefore X_1, and X_2 are one-way separators. To look at the interactions of paths in G with the path q, we look at the two subpaths q_1 and q_2 of q, where q_1 is the portion below the middle vertex and q_2 is the portion above. Let C be the subgraph to the left of q and D the subgraph to the right of q. By construction we can see that no path can cross from D to C since no path can enter q_1 from the right, and no path can leave q_2 from the left (see Figures 5.6 and 5.7). Therefore, X_3 is also a one-way separator. □

We now show how to obtain the separator described in Theorem 13 for a planar layered st-graph G quickly in parallel. A brief sketch of the algorithm is outlined below.

1. For every vertex n determine the level in which it is located. This can be easily determined from the given layered embedding.

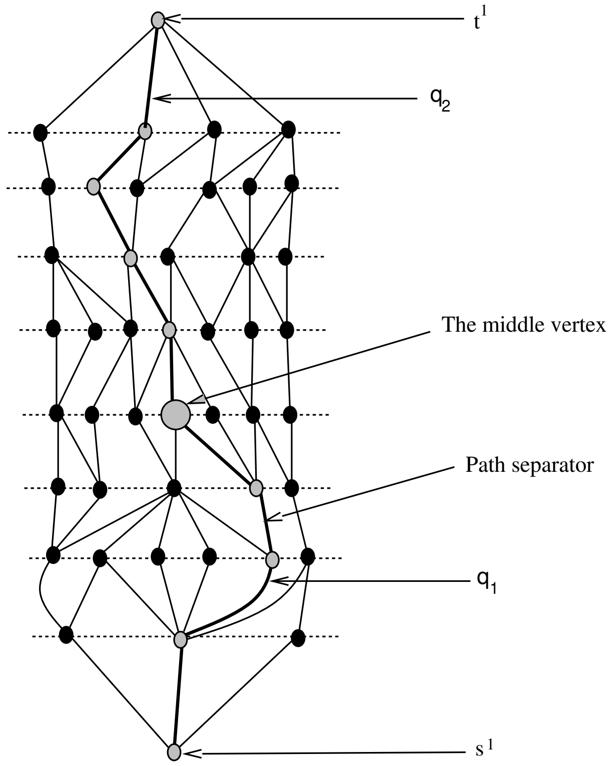

Figure 5.6: Finding a path separator in the middle graph.

2. Determine the number of vertices in level i for all i.

3. Determine the levels i, j, and k.

4. Construct the middle graph M and determine its weight. If the weight is less than $2n/3$ we are done.

5. Construct the left ordering of M, and find the middle vertex x.

6. Construct the "left-most outgoing and right-most incoming" separator starting (as shown in Theorem 13) from x.

Theorem 14 *Given an n-vertex planar layered st-graph G the algorithm outlined above constructs the one-way separator of Theorem 13 and can be implemented in time $O(\log n)$ using $n/\log n$ processors on an EREW PRAM.*

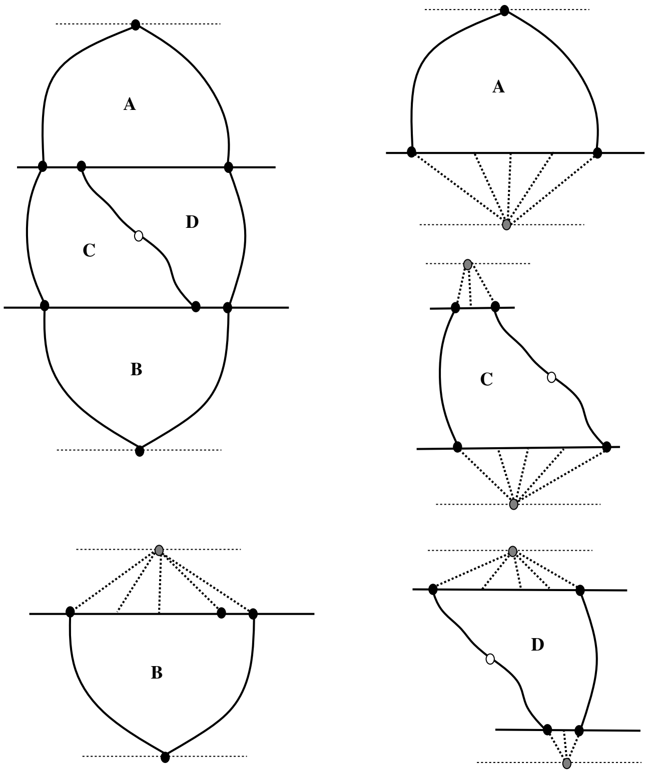

Figure 5.7: A schematic view of the four pieces obtained by using the separator.

Proof: Steps 1 through 4 can be done using standard techniques (see [60]). Steps 5 and 6 can be completed in $O(\log n)$ time using techniques given in [94]. □

Theorem 14 gives us a natural divide-and-conquer strategy for the shortest path problem. The only problem is that even though the separator from Theorem 14 divides the vertices in G into small pieces, it does not guarantee that the number of all-pairs subproblems in each piece is small. This is because, the boundary nodes in G may all end up in one piece. To overcome this problem we use a technique by Frederickson [29]. The idea is as follows.

Mark any vertex used as a separating vertex at some stage, a *boundary vertex.* Use the separator algorithm from Theorem 14 once, to divide the boundary nodes (into small pieces), at every stage. Our separator algorithm can be easily modified to divide a collection of marked nodes. This change in our division strategy guarantees that at any stage each piece has a small number of boundary nodes. In particular, we have the following lemma, which follows from the arguments in [29].

Lemma 33 *Let G be an n-vertex planar layered st-graph, and let $0 \leq \epsilon \leq 1$. Using the separator X defined in Theorem 13 we can divide the graph into $\Theta(n/r)$ regions, such that each region has at most r vertices, and there are only $O(n/\sqrt{r})$ boundary vertices in all. Furthermore, none of the regions contain more than $O(\sqrt{r})$ boundary nodes.*

Our strategy is now clear; we use X to divide G into four pieces A, B, C, and D. Solve the four all-pairs subproblems (among the boundary vertices) in the pieces A, B, C, and D and efficiently patch the solutions along the separator X. To solve the all-pairs problems in each of the pieces we construct planar layered st-graphs from the pieces A, B, C, and D by introducing new source sink vertices and adding edges from the top and the bottom layers, as shown in Figure 5.7. It is easy to see that we introduce only $o(n)$ new vertices (in all the stages) in this fashion. It is easy to see that, all the work is done in the patch-up stage. Therefore to get an efficient shortest-path algorithm we need to get an efficient algorithm for the patch-up stage.

The separator X can split the parent graph in up to four pieces. To make our patch-up algorithm simpler, we view, any split that divides the graph into more than two pieces as a sequence of two splits each dividing a parent graph P into two pieces P_1 and P_2.

In the remaining part we show how to patch up the results of any two pieces P_1 and P_2 to obtain the shortest path information of the parent graph P at all levels of recursion. Let n_p denote the number of nodes in P. To patch up the pieces P_1 and P_2 we need to update the shortest path information along the common boundary. Let us suppose without loss of generality that the source set S is in P_1 and the sink set T is in P_2, and the common boundary between them is X. From Lemma 33 we know that the number of nodes in $S \cup X \cup T$ is $O(\sqrt{n_p})$. In the reminder of the paper we show that the patch-up operation can be done efficiently using $n_p/logn$ processors.

We know from Theorem 13 that X is either a layer of vertices or a special "right-most-edge-in, left-most-edge-out" path constructed from a middle vertex m. We have already solved the subproblems associated with pieces P_1 and P_2, therefore, we know

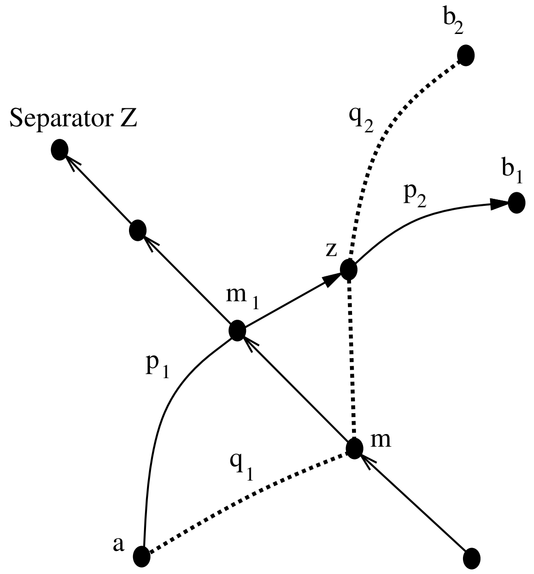

Figure 5.8: Shortest paths from a are non crossing.

the all-pairs shortest paths from vertices in S to those in X, and from vertices in X to those in T. We now have to construct the all-pairs shortest paths from vertices in S to those in T. On the face of it this seems an easy enough job, since for each pair of vertices (a, b) for which $a \in S, b \in T$, all we have to do is check for every vertex in X whether it is in the shortest path from a to b or not. This can be done in constant time with $|S|$ processors in a CRCW PRAM. However, a little thought reveals that such a naive approach would require too many processors. We first use an idea from [5] to show that the all-pairs shortest paths from vertices in S to vertices in T can be computed in $\log^2 n$ time with $n_p/\log n$ processors. We make use of the following definition and lemma.

Definition 11 Let a and b be two vertices separated by a one-way separator $X \subset V$ in the graph G. We denote by $l(a, b)$ the lowest point in the left ordering of the vertices in X in the graph G such that a shortest path from a to b in G passes through it.

Lemma 34 *Let a be a vertex in S and let b_1 and b_2 be vertices in T. Let $m_1 \in X = l(a, b_1)$ in the parent graph P. We claim that if b_2 occurs after b_1 in the left ordering of the parent graph P then $l(a, b_2)$ does not occur before m_1 in the left ordering.*

Proof: We give the proof for the case when X is a path; the proof when X is a layer is similar. The proof is by contradiction. Let $l(a, b_2) = m$ occur before m_1 in the left ordering of P (see Figure 5.8). By the construction of the separator we know that both b_1 and b_2 are on the boundaries of the parent graph P (either on the right boundary or on the layer just below the sink as in Figure 5.7); therefore, the paths from a to b_1 and b_2 must cross each other at some point. Let the point of crossing be z. Consider the two pieces of each path; let the subpath from a to z be p_1, and the one from z to b_1 be p_2. Similarly let the two parts of the path from a to b_2 be q_1 and q_2. Without loss of generality assume that the weight of p_1 is less than that of q_1. We now have a contradiction because the path composed of p_1 and q_2 has weight less than the one made up of q_1 and q_2. □

A similar lemma can be proved about paths from two source vertices to a single sink vertex. We can now use Lemma 34 to do the patching up across the boundary X with n_p processors. For every vertex $a \in S$ we use $\sqrt{n_p}$ processors to determine simultaneously all the shortest paths p_b^a from a to each $b \in T$.

Let us denote the problem of patching across the separator, for a given source vertex a, as $P(a, X, T)$ where we are given the shortest paths from a to every vertex in X and between all pairs of vertices (x, b) where x is in X and b is in T, and we need to determine the shortest paths from a to every vertex $b \in T$. The algorithm to solve $P(a, X, T)$ is as follows:

1. Let b_m be the middle vertex in the left ordering of T.

2. Determine the vertex x_m such that $x_m = l(a, b_m)$.

3. If $n_x = 1$ use $\sqrt{n_p}/\log n$ processors to determine for all $b \in T$ the shortest path from a to b. Otherwise go to Step 4.

4. Divide T into two sets T_1 and T_2 such that $T_1 = \{b \in T \mid b < b_m\}$ and $T_2 = T - T_1$. Similarly divide X into two corresponding sets X_1 and X_2 such that $X_1 = \{x \in X \mid x < x_m\}$ and $X_2 = X - X_1$.

5. Recursively solve the two subproblems $P(a, X_1, T_1)$ and $P(a, X_2, T_2)$.

6. For all vertices $b \in T_1$ compare the shortest path from a to b through X_1, with the one through x_m, and pick the smaller one.

Theorem 15 *Starting with an n-vertex planar layered st-graph, it is possible at any stage of recursion to obtain the shortest path results of the parent graph P from those of its children P_1 and P_2 by running $|S|$ copies of the algorithm outlined above. Moreover this can be done in time $O(\log^2 n)$ in a CREW PRAM, using $n_p/\log n$ processors.*

Proof: Steps 1 and 2 are obviously correct. To prove the correctness of Step 4 we make use of Lemma 34. We know from Lemma 34 that for any $b \in T_2$ the shortest path from a to b does not use any vertex in X_1, and for all $b \in T_1$ the shortest path form a to b either passes through X_1 or through x_m. We can therefore recursively divide the problem into two subproblems $P(a, X_1, T_1)$ and $P(a, X_2, T_2)$, and do the correction in Step 6 to determine the shortest paths to vertices in T_1. To determine the processor and time complexity we note that the left ordering of the any planar *st*-graph with n vertices can be determined in $O(\log n)$ time with $n/\log n$ processors [94]. Thus, we can determine the orderings of S and T in $O(\log n)$ time with $\sqrt{n_p}/\log n$ processors. The time complexity of the various steps are as follows.

1. Step 1, 2, 3, and 4 can be performed in $O(\log n)$ time with $\sqrt{n_p}/\log n$ processors in an EREW PRAM, since $|S \cup X \cup T| = O(\sqrt{n_p})$.

2. The depth of recursion is $O(\log n_p)$, since the size of the sink set T goes down by a factor of 2 every time step 4 is executed. Therefore, the total time taken for the patch up is $O(\log n_p \log n) = O(\log^2 n)$.

3. Step 6 can be done in $O(\log n)$ time with $\sqrt{n_p}/\log n$ processors in a CREW PRAM.

Since running a single copy of the algorithm requires $\sqrt{n_p}/\log n$ processors, $|S|$ copies can be run in $O(\log^2 n)$ time with $n_p/\log n$ processors. $\square$

Theorem 16 *Given a n-vertex planar layered st-graph G with nonnegative edge weights, we can determine the shortest path from s to t in time $O(\log^3 n)$ with $n/\log n$ processors in a CREW PRAM.*

Proof: The proof immediately follows from Lemma 33 and Theorems 14 and 15. □

5.3 The Tube minima problem and a more efficient solution

Consider at any stage of the computation the array A, where $A[i,k,j]$ is the value of the shortest path from vertex $i \in S$ to vertex $j \in T$ that is constrained to pass through the vertex $k \in X$. Let $\Theta_A(i,j)$ be the smallest index k that minimizes $A[i,k,j]$. We say that A is totally monotone if Θ_A satisfies the following properties:

$$\Theta_A(i,j) \leq \Theta_A(i+1,j);$$

$$\Theta_A(i,j) \leq \Theta_A(i,j+1).$$

Furthermore every submatrix A' also satisfies these properties. It can be shown that at every stage of the computation the matrix A constructed as above is totally monotone. It is not hard to see that to patch up two subproblems P_1 and P_2 we only need to determine the value of $\Theta_A(i,j)$ for all i,j. The problem of determining the value of Θ was formalized by Aggarwal and Park as the *tube minima* problem and has many applications in computational geometry [1]. They give the following bounds for calculating the tube minima of an $\sqrt{n} \times \sqrt{n} \times \sqrt{n}$ array:

Theorem 17 *The tube minima of an $\sqrt{n} \times \sqrt{n} \times \sqrt{n}$ array can be found in time $O(\log n)$ and work $O(n)$ in a CREW PRAM.*

Similarly for the CRCW model of computation Atallah [6] has given an algorithm with following bounds:

Theorem 18 *The tube minima of an $\sqrt{n} \times \sqrt{n} \times \sqrt{n}$ array can be found in time $O(\log\log n)$ and work $O(n)$ in a CRCW PRAM.*

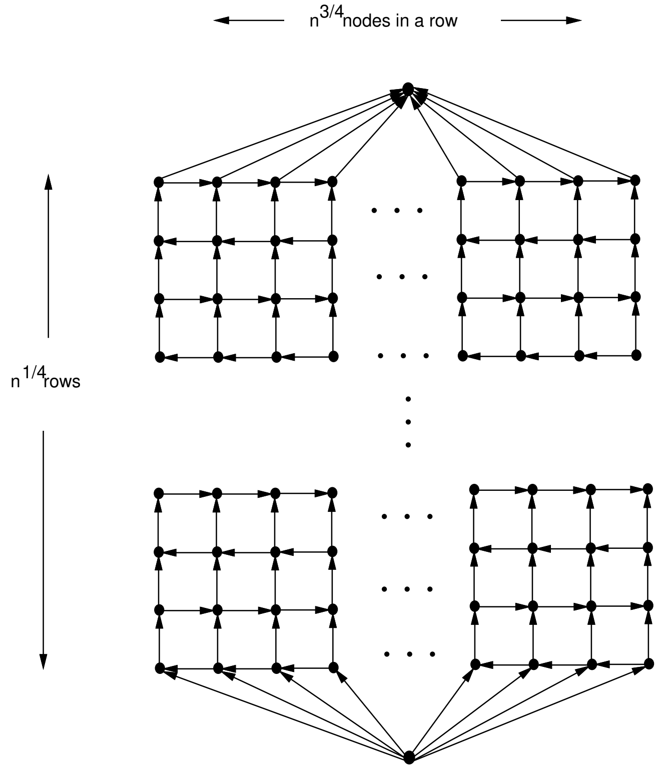

Figure 5.9: A bad planar *st*-graph.

Using these algorithms in the patchup stage we can improve our bounds to the following:

Theorem 19 *Given a n-vertex planar layered st-graph G with nonnegative edge weights, we can determine the shortest path from s to t in time* $O(\log^2 n)$ *with* $n/\log n$ *processors in a CREW PRAM, and in time* $O(\log n \log\log n)$ *with* $n/\log\log n$ *processors in a CRCW PRAM.*

5.4 Remarks

We have given a linear processor CREW algorithm for determining the shortest paths in a planar layered digraph which runs in time $O(\log^2 n)$. Our motivation for considering layered graphs was to see if they can be used to solve the shortest path problem for planar *st*-graphs however since they extend the class of grid digraphs they may have other applications. As far as planar st-graphs are concerned, it seems unlikely that a divide and conquer solution based on one-way separators alone will give a linear processor solution. As evidence we present the graph in Figure 5.9 that does not have a one-way separator of size $O(\sqrt{n})$. It would be interesting to see if the techniques of this chapter can be combined with those of Chapter 4 to design a linear-processor parallel shortest-path algorithm for planar st-graphs that has a small polylogarithmic running time.

Chapter 6

Shortest paths in the presence of negative edge weights

In this chapter we investigate the problem of computing shortest paths when the edge-weights are allowed to have negative values. We give efficient sequential, parallel, and dynamic algorithms for computing single-source shortest paths in planar digraphs. We also give improved algorithms for several related problems like maximum flow, perfect matching etc.

In the sequential setting we obtain an algorithm that takes time $O(n^{4/3}\log(nL))$, where the lengths are integers greater than $-L$. For general graphs, the best bound known is $O(n^{1/2}m\log L)$ time, due to Goldberg [44] which yields $O(n^{3/2}\log L)$ time on sparse (e.g. planar) graphs. For undirected planar graphs, Pan and Reif [82,83] showed how to achieve $O(n^{3/2})$ time using separators. Cohen [14] showed how to achieve the same bound for directed planar graphs. Previously no algorithm handling negative lengths was known that ran faster than $O(n^{3/2})$. Our algorithm overcomes this apparent barrier, improving the time by a (fractional) polynomial factor when the negative lengths are not too large in magnitude.

For planar graphs, shortest-path computation is closely related to network flow. Given a planar digraph and source and sink nodes s and t respectively on different faces of the graph, Miller and Naor [77] (see also Johnson and Venkatesan [56]) show how to solve the max-flow problem by computing single-source shortest-path computation with negative lengths. They also use the same approach in solving the following problem: given multiple sources and multiple sinks, each with a specified supply or demand, find a feasible flow in the network. Bipartite perfect matching in a planar graph, for example, can be formulated in this way. The previous best algorithms for all of these problems required $\Omega(n^{3/2})$ time. Our shortest-path algorithm thus yields polynomial-factor improvements for these problems. In particular, we obtain an $O(n^{4/3} \log n)$-time algorithm for planar bipartite perfect matching. We obtain an $O(n^{4/3} \log n \log C)$-time algorithm for maximum flow, where C is the sum of (integral) capacities.

The approach used in obtaining the shortest-path algorithm for arbitrary lengths also enables us to obtain parallel and dynamic algorithms for this problem and for max-flow and matching.

To obtain our algorithm we use the cluster decomposition of Chapter 2. Our algorithms are extremely simple and are likely to be useful in practice. Our shortest-path algorithms for planar graphs extend to all families of graphs that have $O(\sqrt{n})$ separators.

For this algorithm we will assume that the input planar digraph can have negative edge-weights but no negative cycles are allowed. Note that if there are negative cycles in the graph then the weight of some shortest-path will be unbounded. This assumption is just for the ease of exposition; our algorithm can detect negative cycles and halt execution. The bounds for shortest-paths are as follows.

Theorem 20 *Let G be a n-node planar directed graph such that the sum of the absolute values of the edge-weights is at most D. Given a source node s in G the shortest paths from s to all the other nodes in G can be found in time $O(n^{4/3} \log^{2/3} D)$.*

Miller and Naor [77] show how to solve a number of flow-related problems on planar graphs by performing shortest-path computations on the dual graph. Our improved shortest-path algorithm gives improved algorithms for these problems as well. In particular, we have the following bounds for the computation of maximum flow.

Theorem 21 *The maximum flow in a directed planar graph with a single source and a single sink can be computed in $O(n^{4/3} \log n \log^{2/3} C)$ time where C is the sum of all the edge-capacities. A minimum cut can also be computed within the same time bounds.*

A problem related to maximum flow is that of computing circulations. In this problem we are given a directed network with demands (both positive and negative) on the nodes such that the sum of the demands is zero. Our task is to compute a feasible flow assignment to the edges of the network (if one exists) such that the sum of the flow into any node is equal to the demand at that node. Miller and Naor [77] show that a legal-circulation can be computed in a planar graph G by doing a single-source shortest-path computation in the dual graph. using our shortest-path algorithm we get the following bounds for computing a legal-circulation.

Theorem 22 *A legal circulation can be computed in a planar graph in time $O(n^{4/3} \log^{2/3} C)$ time where C is the sum of all the edge-capacities.*

As a consequence of Theorem 22 we get the following bounds for computing a perfect matching in planar bipartite graphs.

Corollary 6 *Given a planar bipartite graph, a perfect matching can be found in $O(n^{4/3} \log^{2/3} n)$ time.*

Our algorithm can be used to get a faster maximum-flow algorithm when there are multiple sources and sinks that are all distributed over a small number of faces. Using the algorithm due to Miller and Naor along with our shortest-path algorithm we get the following bounds.

Theorem 23 *If G is a planar graph with sources and sinks that lie on at most f faces of G then a maximum flow for G can be computed in time $O(f^2 n^{4/3} \log n \log^{2/3} C)$ time, where C is the sum of all the edge-capacities.*

Our techniques can also be used to get more work-efficient parallel algorithms for all these problems. Using Gabow's [34] reduction of the shortest-path problem to to the problem of computing a maximum matching, along with the parallel algorithm for finding a maximum matching due to Gabow and Tarjan [35], we can get a parallel algorithm for the single-source shortest-path problem with the following bounds.

Theorem 24 *Let G be a n-node planar directed graph such that the sum of the absolute values of the edge-weights is at most D. The single-source shortest-path problem in G from a given source-node s can be solved in time $O(n^{2/3}\log^{7/3} n(\log n + \log D))$ and work $O(n^{4/3}\log n(\log n + \log D))$.*

As corollaries we also get improved parallel algorithms for computing a legal-circulation, max-flow, and planar-bipartite matching.

Corollary 7 *Let G be a n-node planar directed graph such that the sum of the absolute values of the edge-weights is at most D. A legal circulation in G can be computed in time $O(n^{2/3}\log^{7/3} n(\log n + \log D))$ and work $O(n^{4/3}\log n(\log n + \log D))$.*

Corollary 8 *A maximum st-flow, in a graph G with edge capacities that sum up to C, can be found in time $O(n^{2/3}\log^{10/3} n(\log n + \log C))$ and work $O(n^{4/3}\log^2 n(\log n + \log C))$.*

Corollary 9 *If G is planar bipartite graph then we can find a perfect matching in G in time $O(n^{2/3}\log^{13/3} n)$ with $O(n^{4/3}\log^3 n)$ work.*

We can also use these techniques to get an improved algorithm for maintaining all-pairs shortest-paths in planar graphs in a fully dynamic environment. Using the techniques of Chapter 2 we can derive the following bounds.

Theorem 25 *Given an n-node planar directed graph G, such that the sum of the absolute-values of the edge-weights is at most D, there exists a $O(n)$-space fully dynamic data structure to maintain the all-pairs shortest-path information in G. The time per operation is $O(n^{9/7}\log D)$. The time for queries, edge-deletion, and changing lengths is worst-case while the time for adding edges is amortized.*

Our dynamic algorithm can be easily extended to an algorithm for maintaining a legal-circulation (*the dynamic circulation problem*) in a planar graph, and to the problem of dynamically maintaining a perfect matching (*the dynamic perfect-matching problem*) in a planar bipartite graph. For the circulation problem, we are interested in a fully-dynamic data structure which can be queried to determine the amount of flow (in some legal circulation) on an arbitrary edge of the network at any time. In the dynamic perfect-matching problem a query would amount to determining whether a given edge is in the matching or not? We can use our dynamic shortest-path data structure to get dynamic algorithms for these problem with the following bounds.

Corollary 10 *Let G be a n-node planar directed graph such that the sum of the edge-capacities is at most C. There exists a $O(n)$-space fully dynamic data structure to maintain a legal circulation in G. The time per operation is $O(n^{9/7}\log C)$. The time for queries (to determine the amount of flow in a particular edge), edge-deletion, and changing capacities is worst-case while the time for adding edges is amortized.*

Corollary 11 *Given an n-node planar bipartite graph G there exists a fully dynamic $O(n)$-space data structure to maintain a perfect matching in G. Using this data structure we can determine whether a given edge is in the matching in $O(n^{9/7}\log n)$ time. Also, the data structure can be modified in $O(n^{9/7}\log n)$ time to reflect additions and deletions of edges. The time for deletions is worst-case while the time for additions is amortized.*

Subsequent work: Recently Weihe [100] has given an $O(n \log n)$-time algorithm for computing max-flow in a directed planar graph. Thus his bounds improve upon our bounds for computing the max-flow and the related problem of bipartite perfect matching. However, his algorithm relies on the use of dynamic trees [89] while our algorithm does not use any complicated data structures. Thus, our algorithm may perform better in practice.

The remainder of the chapter is organized as follows: In Section 6.1 we give the sequential and parallel algorithms for the single-source shortest-path problem, and discuss their implications on computing maximum flow, circulations, and perfect matchings in planar graphs. In Section 6.2 we discuss our dynamic data structure for maintaining shortest paths, circulations, and perfect-matchings, and in Section 6.3 we give our conclusions and discuss some open problems.

6.1 A three-phase algorithm for single-source shortest-paths

Let G be an n-node planar directed graph such that the sum of the absolute values of the edge-weights is at most D. The *length* (or *weight*) of a directed path (dipath) π from u to v is simply the sum of the weights of the edges in π. A minimum weight dipath from u to v is called a shortest path. Given a source-node s the single-source shortest-path problem is the problem of finding shortest-dipaths from s to all the other nodes in G. In this section we give a simple three-phase algorithm for finding single-source shortest paths in G

6.1.1 Preliminaries

Throughout the chapter by a path we will mean a directed path unless otherwise specified. We will use the term *size* (as opposed to weight) of a path π to refer to the number of edges on π.

Our algorithm will proceed in three phases and will use the cluster decomposition technique of Chapter 2. As in Chapter 2 we will construct a k-cluster partition of G by dividing G into edge-induced subgraphs $G_1, G_2, \cdots, G_j$ such that

1. Each subgraph G_i contains no more than k edges.

2. The number of *boundary nodes* in G_i for any i is $O(\sqrt{k})$.

Unlike in Chapter 2 we will not insist that the boundary nodes be distributed on a bounded number of faces. Such a cluster decomposition was used first by Frederickson [29] to speed up single-source shortest-path computation in planar graphs when edges have nonnegative weights. Recently, in joint work with Klein, Rao, and Rauch [62] we have used a recursive cluster decomposition scheme to derive an optimum linear time algorithm for single source shortest paths in planar digraphs when edge weights are restricted to be nonnegative.

6.1.2 The three-phase algorithm

Our algorithm uses the partitioning technique described above. The main idea is as follows: In the first phase we find a k*-cluster partition* of G (the optimum value of k will be determined later). For each subgraph G_i in the cluster partition we use nested dissection [82,83] to compute the all-boundary-pair shortest-paths within G_i. While computing the boundary-to-boundary shortest paths we also compute a compact graph $C(G_i)$ with the following properties:

- $V(C(G_i)) = V(G_i)$.
- Each edge in $C(G_i)$ corresponds to a path in G_i.
- $|E(C(G_i))| = |E(G_i)| \log(|E(G_i)|)|$.
- For any two nodes $u, v \in G_i$ there exists a shortest u-to-v path in $C(G_i)$ that has size $O(\log n)$.

Such a compact graph can be easily constructed while performing a nested dissection computation (see Cohen [14]). For the sake of completeness we now give a brief account of how the nested-dissection (hereafter referred to as the *ND-algorithm*) algorithm can be used to compute all-boundary-pair shortest-paths and for computing the compact representation. For this purpose we use a simple version of the ND-algorithm that is not optimum in terms of the logarithmic factors.

To compute the all-boundary pair shortest-paths in a given l-node subgraph G_α with a boundary set $B(G_\alpha)$ of $O(\sqrt{l})$ boundary nodes, the ND-algorithm works as follows:

The subgraph G_α is first divided using an $O(\sqrt{l})$ planar separator X into up to three pieces P_1, P_2, and P_3 such that for all i P_i contains at most two-thirds of the nodes in G_α. Also, each P_i contains at most two-thirds of the boundary nodes in $B(G_\alpha)$. For each i the boundary set $B(P_i)$ of P_i is set to be $(B(G_\alpha) \cup X) \cap P_i$. In other words the boundary nodes are the nodes in P_i that are either in the separator X or are in the boundary set of G_α. We also define $S = B(G_\alpha) \cup X$ to be the *splitting set* of G_α. The splitting sets of P_1, P_2, and P_3 are defined recursively.

After dividing G_α, the ND-algorithm recursively computes all-boundary-pair shortest-paths in each of the subgraphs P_1, P_2, and P_3. We also recursively compute the compact graphs $C(P_1)$, $C(P_2)$, and $C(P_3)$. For $i = 1, 2, 3$, let M_i denote the all-boundary-pair shortest-path matrix of subgraph P_i. The entry (a, b) in M_i contains the shortest-path distance between a and b in P_i. To find the all-boundary-pair shortest-paths in R we need to compute the matrix M from the matrices M_1, M_2, and M_3. To do that we first construct a combined matrix C (by unioning M_1, M_2, and M_3) that represents all-boundary-pair shortest in all the subgraphs. In constructing C for any pair (a, b) that occurs in more than one matrix M_i, we pick the minimum entry from all the matrices. It is easy to see that by performing a transitive closure of C (using the operations $+$ and *min* instead of $\times$ and $+$) we can find all-pair shortest-paths in G_α for the nodes in S. Let C^* denote the transitive closure of C.

To find the all-boundary-pair shortest-paths in G_α we just pick out the submatrix M from C^*. To represent the all-boundary pair shortest-paths in G_α we construct a complete graph $\hat{G}_\alpha$ on the node set $B(G_\alpha)$ such that the edge $ab \in \hat{G}_\alpha$ corresponds to the shortest a-to-b path in G_α. The weight of ab is set to be the $M[a, b]$. It is not hard to show that each of M_1, M_2, and M_3 is an $O(\sqrt{l}) \times O(\sqrt{l})$ matrix. Therefore, C is also a $O(\sqrt{l}) \times O(\sqrt{l})$ matrix and thus finding the transitive closure of C takes $O(l^{1.5} \log l)$ work. It can also be shown that at any level in the recursion the total amount of work needed to perform all the transitive closure computations is $O(l^{1.5} \log l)$. Thus the total amount of work required to find all-boundary-pair shortest-paths in G_α is $O(l^{1.5} \log^2 l)$. A more involved method [14,83] can be used to cut the work down to $O(l^{1.5})$.

The compact graph $C(R)$ of R is formed by unioning $C(P_1)$, $C(P_2)$, and $C(P_3)$ along with C^*. It is not hard to show that for any two nodes a and b in G_α there exists a shortest a-to-b path in $C(G_\alpha)$ that contains $O(\log l)$ edges (see Cohen [14]).

```
procedure ShortPath(G)
  [The dividing step]
  1. Use planar separators to find a k-cluster
     partition of G with subgraphs G_1, G_2, ..., G_k.
  2. Mark s as a boundary node.

  [The nested dissection phase]
     for each subgraph G_i do
     1. Use nested dissection to find all-boundary-pair shortest-paths
        and the compact C(G_i).
     2. Let Ĝ_i be a complete graph with V(Ĝ_i) = B(G_i) and
        edges corresponding to all-boundary-pair shortest-paths in G_i.
        end [ for]

  [The boundary-path phase]
  1. Let H be the union of Ĝ_i over all i.
  2. Use Goldberg's algorithm to compute shortest-paths in H
     from s to all the other nodes.

  [Extending the information to the non-boundary nodes]
     for each subgraph G_i do
     1. Add a pseudo-source node ŝ to C(G_i).
     2. Add edges from ŝ to all the boundary nodes in B(G_i)
        corresponding to shortest-paths from s to the boundary nodes in H.
     3. Compute shortest-path distances from s to all the nodes in C(G_i)
        by running the Bellman-Ford shortest-path algorithm from s for O(log n) steps.
     end [ for ]
end [ procedure ShortPath ]
```

Figure 6.1: The three-phase shortest-path algorithm

After the nested dissection phase we construct an auxiliary graph H that contains the boundary nodes from all the subgraphs and edges corresponding to all-boundary-pair shortest-paths in each subgraph. We use the single-source shortest path algorithm due to Goldberg [44] in H to compute shortest paths from s to all the boundary nodes of G. This constitutes the second phase of our algorithm.

In the last phase, for each subgraph G_i we use the single-source shortest-path information of the second phase along with the compact graph $C(G_i)$ to find shortest paths to all the other nodes in G_i. To do this we first augment $C(G_i)$ by introducing a pseudo-source node $\hat{s}$ in $C(G_i)$ and introducing edges from $\hat{s}$ to the boundary nodes in $B(G_i)$ to reflect their shortest-path distances from s in H. We then run the Bellman-Ford shortest-path algorithm [10,57] for $O(\log n)$ steps from the source node $\hat{s}$. This gives us shortest-path distances to all the nodes in G_i. A pseudo-code version of the algorithm is given in Figure 6.1.

To prove the correctness of the algorithm we note that at the end of the nested dissection phase, for each subgraph G_i, $\hat{G}_i$ represents the all-pairs shortest-paths among the boundary-nodes within G_i.

We claim that in order to find shortest-paths from s to the boundary nodes of all the subgraphs we need only to perform a single-source shortest-path computation in H. To see this consider a shortest-path π in G from s to some boundary node b. We can divide π into a sequence of maximal sub-paths $\pi_1, \pi_2, \cdots, \pi_j$ such that each π_i lies entirely within a single subgraph. For any π_i let $G(\pi_i)$ be the subgraph that contains π_i. By maximality π_i is a shortest path between two boundary-nodes x and y in $G(\pi_i)$. Thus, there is an edge with the same weight between x and y in $\hat{G}(\pi_i)$. By replacing each sub-path π_i by the corresponding edge (between its endpoints) in $\hat{G}(\pi_i)$ we get an alternate s-to-b path π' in H that has the same weight as π. Therefore, at the end of the second phase we know the shortest-path distances from s to all the boundary nodes of each subgraph.

We now only need to find shortest-path distances to the internal nodes of each subgraph G_i. To do this we make use of the compact graph $C(G_i)$ augmented with edges corresponding to shortest-paths from s to the boundary nodes.

Let u be any node in G_i and let π be a shortest-path from s to u in G. It is easy to see that π can be split into two sub-paths π_1 and π_2 where π_1 is a shortest path from s to some boundary-node b in G_i and π_2 is a shortest path from b to u that lies entirely within G_i. By definition there is a corresponding shortest-path π_2' from b to u in $C(G_i)$ with size $O(\log n)$. Also there is a single edge $\hat{s}b$ in the augmented version of $C(G_i)$ that corresponds to the shortest path from s to b in G. Thus the path π' consisting of $\hat{s}b$ and ${\pi_2}'$ is an $O(\log n)$-size path that corresponds to a shortest s-to-u path in G.

Cohen [15] has observed that by running the Bellman-Ford algorithm for q-stages from a source s one can find all shortest-paths that have only q-edges. Therefore by running the Bellman-Ford algorithm for $O(\log n)$ stages on the augmented $C(G_i)$ we can find the shortest paths from s to all the nodes of G_i. Doing this for all the subgraphs gives us the shortest-path distances to all the nodes in G. The correctness of our algorithm therefore follows.

To bound the amount of time taken by our algorithm we note that in the nested dissection phase we need $O(k^{3/2})$ time for each subgraph in the cluster partition (see [14, 82,83]). Therefore, the time required to perform nested dissection in all the subgraphs is $O(k^{3/2}\frac{n}{k})$. To bound the time required for the single-source shortest-path computation on the auxiliary graph H we note that H has $O(n)$ edges and $O(n/sqrtk)$ nodes, therefore running Goldberg's algorithm on H requires $O(\frac{n^{3/2}}{k^{1/4}}\log D)$ time. The time required to run a Bellman-Ford computation on one subgraph for $O(\log n)$ iteration is $O(k\log k)$ (see for instance [18]). Therefore, the time required to run the third-phase on all the subgraphs is $O(n\log k)$. Letting k be equal to $n^{2/3}\log^{4/3} D$ we see that the entire algorithm requires $O(n^{4/3}\log^{2/3} D)$ time. We thus get the bounds of Theorem 20.

To get our bounds for the related problems of computing a circulation, computing a maximum flow, etc., we just use our shortest-path algorithm as a subroutine in the algorithms given by Miller and Naor [77]. Hassin [48] (see also [49]) showed that computing a circulation in planar directed graph is equivalent to performing a single-source shortest computation in the dual graph. Miller and Naor [77] showed that the problem of computing a perfect matching in a planar bipartite graph can be reduced to the problem of finding a circulation in the same graph. Therefore, we can use our shortest-path algorithm to get the bounds in Theorem 22 and corollary 6. The algorithms for maximum flow in [77] use $O(\log n)$ single-source computations in the dual ($O(k^2\log n)$ if the sources and sinks are distributed over k faces). Therefore, using our shortest-path algorithm we get the bounds in Theorems 21 and 23.

To get an efficient parallel algorithm we follow the same strategy but instead of using Golberg's algorithm in the second phase we use an algorithm due to Gabow and

Tarjan [35]. Given an n-node m-edge graph H with integral edge-weights that are at most N in magnitude the algorithm in [35] can compute single-source shortest-paths in H in time $O(\sqrt{n}m \log(nN)(\log 2p)/p)$ using $p \leq m/(\sqrt{n} \log^2 n)$ processors. Using this algorithm in the boundary-path stage and setting k to be $n^{2/3} \log^{4/3} n$ and p to be $\frac{n^{2/3}}{\log^{5/3} n}$ we can get a parallel algorithm with the bounds of Theorem 24. Using this shortest-path algorithm we can also get parallel algorithms for computing a legal-circulation, the maximum st-flow, and a perfect matching in a planar bipartite graph with the bounds shown in corollaries 7, 8, and 9.

6.2 A fully-dynamic data structure for shortest paths in planar graphs when edges can have arbitrary integral values

In this section we describe a fully dynamic data structure for maintaining all-pairs shortest paths in a planar directed graph that can have both negative and nonnegative-weight edges. As with the sequential algorithm our dynamic algorithm also uses the decomposition technique of Frederickson. We again use the cluster-partitioning approach of Chapter 2. Throughout this section we assume that the edge additions preserve planarity. This can be easily enforced within the same time and space bounds by running the planarity testing algorithm by Galil, Italiano, and Sarnak [37] in the background and only allowing edge additions that preserve planarity.

Our dynamic algorithm supports the following operations:

1. **distance**(u, v): Find the distance between u and v in G.
2. **add**(u, v, w): Results in the addition of the edge uv with weight w.
3. **delete**(u, v): Results in the deletion of the edge uv.

We start by finding an k-cluster partition of G (the optimal value of k will be determined later). As in the sequential algorithm we use nested dissection to find all-boundary-pair shortest-paths in each of the subgraphs. We also construct the auxiliary graph H as before by unioning the complete graphs $\hat{G}_1, \hat{G}_2, \cdots . \hat{G}_j$. Hereafter we will refer to $\hat{G}_i$ as the *substitute graph* of G_i. Our data structure consists of the auxiliary graph H and the subgraphs $G_1, G_2, \cdots, G_j$. The auxiliary graph H is used in the query-stage to compute the distance between the two query points.

To find the distance between two given query points u and v we form a new auxiliary graph S by unioning the subgraphs containing u and v (G_u and G_v respectively) along with H. We then run a sequential shortest path algorithm on S to compute the distance between u and v.

To prove the correctness of our query-procedure, we need to show that the distance between u and v in S is the same as the distance between u and v in G. Let π be a shortest u-to-v path in G. Mark all the boundary nodes in π. This marking divides π into a sequence of subpaths such that the first and the last subpaths lie entirely within G_u and G_v respectively and each intermediate subpath is a boundary-to-boundary shortest-path that lies entirely within one of the subgraphs G_1 through G_j. Since the boundary-to-boundary shortest paths in G_i are represented by edges in $\hat{G}_i$, it follows that S contains a path from u to v whose weight is the same as that of π.

To perform an update (whenever an edge is added, deleted, or its cost is changed) we need to recompute the substitute graph of each subgraph G_i that is affected by the change in the input. Lemma 35 shows that only a constant number of subgraphs are affected during a single update operation.

Lemma 35 *Adding, deleting, or changing the weight of an edge affects only a constant number of subgraphs in the cluster partition. Therefore, only a constant number of substitute graphs need to be recomputed to modify H.*

Proof: We discuss the case of a delete operation. The arguments for addition and changing edge-weights are similar. Consider deleting the edge uv. This results in a change in the subgraph G_{uv} that contains uv. Furthermore, it is possible that deleting this edge could make either u (or v) a non-boundary node (this could happen if all the remaining edges incident at u (v) now lie in a single subgraph). Such a change in the boundary-status can affect only the subgraphs G_u and G_v that contain u and v

respectively. Therefore, to modify H we need only recompute $\hat{G}_{uv}$, $\hat{G}_u$, and $\hat{G}_v$. $\square$

As in Chapter 2 repeated requests for adding edges can result in an excess of boundary nodes. Therefore, we recompute the cluster partition after the number of *add* operations exceeds the value of a preset parameter *limit*.

We are now ready to discuss our bounds for maintaining all-pairs shortest paths. We set k to be $n^{6/7}$ and *limit* to be $n^{3/7}$. Since we recompute the cluster partition every $n^{3/7}$ adds no subgraph has more than $n^{3/7}$ boundary nodes at any time.

To bound the query time we note that the number of nodes in H at any time is $O(n^{4/7})$ and the number of edges in H at any time is $O(n)$. It is easy to see that the same bounds hold for the size of the auxiliary graph S as well. Therefore, Goldberg's algorithm [44] will require $O(n^{9/7} \log D)$ time to find the shortest u-to-v path.

To bound the update time we note that the all-boundary-pair shortest-paths can be found in $O(n^{9/7})$ time for any subgraph G_i using nested dissection. Since only a constant number of subgraphs are affected during an update operation, the time-bound for an update is also $O(n^{9/7})$. We also recompute the cluster-partition and all the substitute graphs once every $n^{3/7}$ *adds*. The time taken to recompute the cluster partition is $O(n)$ while the time taken to build all the substitute graphs is $O(n^{10/7})$. Amortizing this over $n^{3/7}$ *adds* we find that the amortized time of rebuilding the data structure is $O(n^{9/7})$ per *add* operation. We therefore get a fully dynamic algorithm for maintaining exact all-pairs shortest paths that requires $O(n^{9/7} \log D)$ time per operation. The bounds of Theorem 25 therefore follow.

Our dynamic shortest-path algorithm can be used to get algorithms for the dynamic circulation and the dynamic perfect-matching problem as well. However, for these two problems we restrict the edge additions to be embedding preserving. In other words a new edge can be added if and only if it is consistent with the existing embedding.

To get a fully-dynamic algorithm for maintaining a legal-circulation in an embedded planar network G we note that due to a reduction by Hassin [48] (see also [49,77]) the problem of computing a legal circulation in a planar network is equivalent to computing the single-source shortest-path distances from an arbitrary source s in the dual network

G'. The dual network contains a node for every face f in G, and a dual edge e' for every edge e in G. The dual edge of e connects the face f_1 to the left of e (in the embedding) to the face f_2 that is to the right of e. The weight of the dual edge is $f_1 f_2$ is the same as the weight of e. Note that the dual network may contain more than one edge between the same pair of nodes. However, all but the lightest edge can be ignored for the shortest-path computations.

To determine the flow on a particular edge e in the graph; we compute the shortest-path distances of f_1 and f_2 from the source s in the dual network. Here $f_1 f_2$ is the dual edge corresponding to e. Let $d(f_1)$ and $d(f_2)$ denote the shortest-path distances to f_1 and f_2 respectively. Then the flow on e in the circulation is set to be $d(f_2) - d(f_1)$. Hassin [48] shows that this procedure gives a legal circulation in G.

Thus to determine the flow on any particular edge in G we need to perform two shortest-path queries in the dual graph G'. Therefore, the query time for flow-determination is also $O(n^{9/7} \log C)$ (where C is the sum of the edge-capacities). Changes in the primal network correspond to obvious changes in the dual network, and the update algorithms remain the same. We therefore get the bounds in Corollary 10. The same algorithm essentially works for dynamic maintenance of perfect-matchings as well. By using the reduction due to Miller and Naor [77] we can reduce the perfect-matching problem to the problem of computing a legal circulation in the same graph such that an edge is in the matching if and only if the flow on the edge is non-zero. The bounds of Corollary 11 therefore follow.

6.3 Comments

In this chapter we have given improved sequential, parallel, and dynamic algorithms for solving shortest-path problems in planar graphs when edges can have negative and non-negative weights. Due to the simplicity of our algorithms, they are likely to be efficient in practice as well.

It is conventional wisdom that the single-source shortest-path problem in general graphs is harder when negative edge-weights are allowed. It would be interesting to see whether the same holds true for planar graphs as well. In this paper we have shown that the single-source shortest-path problem (with negative edge-weights) can be solved considerably faster if the underlying graph is planar thus narrowing the gap in the running time between the two versions of the problem: (1) Where edge weights are restricted to be non-negative and (2) Where the edge-weights need not necessarily be non-negative. It would be interesting to see if this gap can be further narrowed.

Our algorithm uses a simple two-level decomposition strategy to get a better running time. It might be possible to get faster algorithms by using a more sophisticated multi-level decomposition strategy. We leave the construction of such an algorithm as an open problem.

References

[1] A. Aggarwal and J. Park, "Notes on searching multidimensional monotone arrays," *Proc. 29th Annual IEEE Symposium on Foundations of Computer Science* (1988), 496–512.

[2] A. Aho, J. Hopcroft, and J. Ullman, *The design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[3] R. Ahuja, K. Mehlhorn, J. Orlin, and R. E. Tarjan, "Faster algorithms for the shortest path problem," *Journal of the Association for Computing Machinery* 37 (1990), 213–223.

[4] N. Alon, P. Seymour, and R. Thomas, "A separator theorem for graphs with an excluded minor and its applications," *Proc. 22nd Annual ACM Symposium on Theory of Computing* (1990), 293–299.

[5] A. Apostolico, M. J. Atallah, L. Larmore, and H. S. Mcfaddin, "Efficient parallel algorithms for string editing and related problems ," *SIAM Journal on Computing* 19 (1990), 968–988.

[6] M.J. Atallah, "A faster parallel algorithm for a matrix searching problem," *Proc. 2nd Scandinavian Workshop on Algorithm Theory* (1990), 193–200.

[7] G. Ausiello, G.F. Italiano, A. Marchetti-Spaccamela, and U. Nanni, "Incremental algorithms for minimal length paths," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1990), 12–21.

[8] B. S. Baker, "Approximation algorithms for NP-complete problems on planar graphs," *Proc. 24th Annual IEEE Symposium on Foundations of Computer Science* (1983), 265–273.

[9] H. Bast, M. Dietzfelbinger, and T. Hagerup, "A perfect parallel dictionary," *Symposium on Mathematical Foundations of Computer Science* (1992).

[10] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics* 16 (1958), 87–90.

[11] G. Birkhoff, "Lattice Theory," *American Mathematical Society Colloquium Publications* 25 (1979).

[12] A.L. Buchsbaum, P.C. Kanellakis, and J.S. Vitter, "A data structure for arc insertion and regular path finding," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1990), 22–31.

[13] E. Cohen, "Fast algorithms for constructing t-spanners and paths with stretch t," *Proc. 34th Annual IEEE Symposium on Foundations of Computer Science* (1993).

[14] E. Cohen, "Efficient parallel shortest-paths in digraphs with a separator decomposition," *Proc. 5th Annual Symposium on Parallel Algorithms and Architectures* (1993), 57–67.

[15] E. Cohen, "Parallel algorithms with improved work for shortest-paths from multiple sources," *Proc. 2nd Israel Symposium on Theory of Computing and Systems* (1993), 57–67.

[16] E. Cohen, "Polylog-time and near-linear work approximation scheme for undirected shortest paths," *Proc. 26th Annual ACM Symposium on Theory of Computing (to appear).* (1994).

[17] R. Cole and U. Vishkin, "Optimal parallel algorithms for expression tree evaluation and list ranking ," *Procedings of the third Agean Workshop on Computing:* (1988), 91–100.

[18] T. H. Cormen, C. E. Leiserson, and R. E. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[19] A. L. Delcher and S. R. Kosaraju, "An NC algorithm for evaluating monotone planar circuits," manuscript submitted for publication.

[20] G. Di Battista and E. Nardelli, "An algorithm for testing planarity of hierarchical graphs," *Proc. In. Workshop WG86, Bernierd, June 1986.* (1987), 277–289.

[21] H. N. Djidev, "A separator theorem," *C. R. Bulgare Sci.* 34 (1981), 643–645.

[22] H. N. Djidev, "A linear algorithm for partitioning graphs," *C. R. Bulgare Sci.* 35 (1982), 1053–1056.

[23] H. N. Djidev, "On the problem of partitioning planar graphs," *SIAM J. ALg. Discrete Methods* 3 (1982), 229–240.

[24] H. N. Djidev, G. E. Pantziou, and C. D. Zaroliagis, "Computing shortest paths and distances in planar graphs," *Proc. 18th International Colloquium on Automata, Languages and Programming* (1991), 327–338.

[25] D. Eppstein and Z. Galil, "Parallel algorithmic techniques for combinatorial computation," Columbia University, Preprint, 1988.

[26] D. Eppstein, Z. Galil, G.F. Italiano, and T. Spencer, "Separator based sparsification for dynamic planar graph algorithms," *Proc. 25th Annual ACM Symposium on Theory of Computing* (1993).

[27] S. Even and H. Gazit, "Updating distances in dynamic graphs," *Methods of Operations Research* 49 (1985), 371–387.

[28] G.N. Frederickson, "Data structures for on-line updating of minimum spanning trees, with applications," *SIAM J. Computing* 14 (1985), 781–798.

[29] G.N. Frederickson, "Fast algorithms for shortest paths in planar graphs, with applications," *SIAM Journal on Computing* 16 (1987), 1004–1022.

[30] G.N. Frederickson, "Ambivalent data structures for dynamic 2-edge connectivity and k-smallest spanning trees," *Proc. of the 32nd Symposium on Foundations of Computer Science* (1991), 632–641.

[31] M. L. Fredman and D. E. Willard, "Trans-dichotomous algorithms for minimum spanning trees and shortest paths," *Proc. 31st Annual IEEE Symposium on Foundations of Computer Science* (1990), 719–725.

[32] M.L. Fredman and R.E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the Association for Computing Machinery* 34 (1987), 596–615.

[33] E. Fuerstein and A. Marchetti-Spaccamela, "Dynamic algorithms for shortest path problems in planar graphs," *Proc. 17th International Workshop on Graph Theoretic Concepts in Computer Science* (1991), 187–197.

[34] H. N. Gabow, "Scaling algorithms for network problems," *Journal of Computer and System Sciences* 31 (1985), 148–168.

[35] H. N. Gabow and R. E. Tarjan, "Almost-optimum speed-ups of algorithms for bipartite matching and related problems," *Proc. 20th Annual ACM Symposim on Theory of Computing* (1988), 514–527.

[36] Z. Galil and G. F. Italiano, "Maintaining biconnected components of dynamic planar graphs," *Proc. 18th Int. Colloquium on Automata, Languages, and Programming.* (1991), 339–350.

[37] Z. Galil, G.F. Italiano, and N. Sarnak, "Fully dynamic planarity testing," *Proc. 24th Annual ACM Symposium on Theory of Computing* (1992), 495–506.

[38] H. Gazit and G. L. Miller, "An $O(\sqrt{n}\log n)$ optimal parallel algorithm for a separator in planar graphs," manuscript.

[39] H. Gazit and G. L. Miller, "A deterministic parallel algorithm for finding a separator in planar graphs," manuscript.

[40] H. Gazit and G. L. Miller, "A parallel algorithm for finding a separator in planar graphs," *Proc. 28th Annual IEEE Symposium on Foundations of Computer Science* (1987), 238–248.

[41] H. Gazit and G. L. Miller, "An improved parallel algorithm that computes the BFS numbering of a directed graph," *Information Processing Letters* 28 (1988), 61–65.

[42] J. Gil, Y. Matias, and U. Vishkin, "Towards a theory of nearly constant time parallel algorithms," *Proceedings of 33rd IEEE Symposium on Foundations of Computer Scvience* (1991), 628–637.

[43] J. R. Gilbert, J. P. Hutchinson, and R. E. Tarjan, "A separation theorem for graphs of bounded genus," Journal of Algorithms, 1984.

[44] A.V. Goldberg, "Scaling algorithms for the shortest path problem," *SIAM Symposium on Discrete Algorithms.* (1993).

[45] M. Goodrich, "Planar separators and parallel polygon triangulation," *Proc. 24th Annual ACM Symposium on Theory of Computing* (1992), 507–516.

[46] S. Guattery and G. L. Miller, "A contraction procedure for planar directed graphs," *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures* (1992), 431–441.

[47] T. Hagerup and R. Raman, "Waste makes haste: Tight bounds for loose parallel sorting," *Proceedings of the 33rd IEEE Symposium on Foundations of Computing* (1992), 628–637.

[48] R. Hassin, "Maximum flow in (s, t) planar networks," *Information Processing Letters* 13 (1981), 107.

[49] R. Hassin and D. B. Johnson, "An $O(n \log\ 2n)$ algorithm for maximum flow in undirected planar networks," *SIAM Journal on Computing* 14 (1985), 612–624.

[50] G.F. Italiano, "Amortized efficiency of a path retrieval data structure," *Theoretical Computer Science* 48 (1986), 273–281.

[51] G.F. Italiano, "Finding paths and deleting edges in directed acyclic graphs," *Information Processing Letters* 28 (1988), 5–11.

[52] G.F. Italiano, A. Marchetti-Spaccamela, and U. Nanni, "Dynamic data structures for series-parallel graphs," *Proc. WADS' 89,* LNCS 382 (1989), 352–372.

[53] D.B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *Journal of the Association for Computing Machinery* 24 (1977), 1–13.

[54] D.B. Johnson, "Parallel algorithms for minimum cuts and maximum flows in planar networks," *Journal of the Association for Computing Machinery* 34 (1987), 950–967.

[55] D.B. Johnson and S. M. Venkatesan, "Parallel algorithms for minimum cuts and maximum flows in planar networks," *Proc. 23rd Annual IEEE Symposium on Foundations of Computer Science* (1982), 244–254.

[56] D.B. Johnson and S.M. Venkatesan, "Using divide and conquer to find flows in directed planar networks in $O(n^{1.5} \log n)$ time," *Proc. 20th Annual Allerton Conf. on Communication, Control, and Computing* (1982), 898–905.

[57] L. R. Ford Jr. and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, 1962.

[58] T. Kameda, "On the vector representation of the reachability in planar directed graphs," *Information Processing Letters* 3 (1975), 75–77.

[59] M. Kao and P. N. Klein, "Towards overcoming the transitive closure bottleneck: efficient parallel algorithms for planar digraphs," *Proc. 22nd Annual ACM Symposium on Theory of Computing* (1990), 181–192.

[60] R.M. Karp and V. Ramachandran, "A survey of parallel algorithms for shared memory machines," in *Handbook of Theoretical Computer Science*, North Holland, 1990, 871–941.

[61] D. Kelly, "On the Dimension of Partially Ordered Sets," *Discrete Mathematics* 35 (1981), 135–156.

[62] P. N. Klein, S. Rao, M. Rauch, and S. Subramanian, "Faster shortest-path algorithms for planar graphs," *Proc. 26th Annual ACM Symposium on Theory of Computing (STOC)* (1994).

[63] P. N. Klein and S. Sairam, "A parallel randomized approximation scheme for shortest paths," *Proc. 24th Annual ACM Symposium on Theory of Computing* (1992), 750–758.

[64] P. N. Klein and S. Subramanian, "A linear-processor polylog-time algorithm for shortest-paths in planar graphs," *Proc. 1993 IEEE Symposium on Foundations of Computer Science.* (1993), 259–270.

[65] P. N. Klein and S. Subramanian, "A randomized parallel algorithm for single-source shortest paths," Technical Report, Brown University, submitted for publication, 1994.

[66] P.N. Klein, "On the Gazit and Miller separator algorithm: a work-efficient algorithm for finding cycle separators in planar graphs," *Proc. 5th Annual Symposium on Parallel Algorithms and Architectures* (1993).

[67] J. Lagergren, "Efficient parallel algorithms for tree-decomposition and related problems," *Proc. 30th Annual IEEE Symp. on Foundations of Computer Science* (1990), 173–182.

[68] F.T. Leighton, *Complexity Issues in VLSI: Optimal Layouts for the Shuffle-Exchange Graph and other Networks*, MIT Press, 1983.

[69] C. Leiserson, *Area-efficient VLSI computation*, Foundations of Computing, MIT Press, Cambridge, MA, 1983.

[70] A. Lempel, S. Even, and I. Cederbaum, "An algorithm for planarity testing of graphs," *Theory of Graphs, Int. Symposium* (1966), 215–232.

[71] A. Lingas, "Fast parallel algorithms for planar directed graphs," *Proc. SIGAL International Symposium on Algorithms* (1990), 447–457.

[72] R. Lipton, D. Rose, and R.E. Tarjan, "Generalized nested dissection," *SIAM J. Numerical Analysis* 16 (1979), 346–358.

[73] R. J. Lipton and R. E. Tarjan, "Applications of a planar separator theorem," *SIAM Journal on Computing* 9 (1980), 615–627.

[74] R.J. Lipton and R.E. Tarjan, "A separator theorem for planar graphs," *SIAM Journal of Applied Mathematics* 36 (1979), 177–189.

[75] N. Megiddo, "Parallel algorithms for finding the minimum and the median almost surely in constant time," preprint, 1982.

[76] G. L. Miller, "Finding small simple cycle separators for 2-connected planar graphs," *Journal of Computer and System Sciences* 32 (1986), 265–279.

[77] G. L. Miller and J. Naor, "Flows in planar graphs with multiple sources and sinks," *Proc. 30th IEEE Symposium on Foundations of Computer Science* (1989), 112–117.

[78] G. L. Miller, S. Teng, and S. Vavasis, "A unified geometric approach to graph separators," *Proc. 31st Annual IEEE Symposium on Foundations of Computer Science* (1991), 538–547.

[79] G. L. Miller and W. Thurston, "Separators in two and three dimensions," *Proc. 22nd Annual ACM Symposium on Theory of Computing* (1990), 300–309.

[80] G. L. Miller and S. A. Vavasis, "Density graphs and separators," *Proc. 2nd Annual ACM-SIAM Symposium on Discrete Algorithms* (1991).

[81] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia, "Complexity models for incremental computation," *Theoretical Computer Science: Special issue on Dynamic and On-line Algorithms* (1994), (to appear).

[82] V. Pan and J. H. Reif, "Fast and efficient solution of path algebra problems," *Journal of Computer and System Sciences* 38 (1989), 494–510.

[83] V. Pan and J. H. Reif, "The parallel computation of minimum cost paths in graphs by stream contraction," *Information Processing Letters* 40 (1991), 79–83.

[84] J.A. La Poutré and J. van Leeuwen, "Maintenance of transitive closures and transitive reductions of graphs," *Proc. WG '87,* LNCS 314 (1988), 106–120.

[85] V. Ramachandran and J.H. Reif, "An optimal parallel algorithm for graph planarity," *Proc. IEEE Symp. on Foundations of Computer Science* (1989).

[86] V. Ramachandran and H. Yang, "An efficient parallel algorithm for general planar monotone circuit value problem," *Proc. Fifth Annual SIAM Symposium on Discrete Algorithms* (1994), 622–631.

[87] J. H. Reif, "Minimum s-t cut of a planar undirected network in $O(n \log 2n)$ time," *SIAM Journal on Computing* 12 (1983), 71–81.

[88] H. Shi and T. Spencer, "Time-work tradeoffs for the single-source shortest paths problem," Department of Computer Science University of Nebraska at Omaha, Technical Report CS-TR-94-1, 1994.

[89] D.D. Sleator and R.E. Tarjan, "A data structure for dynamic trees," *Journal of Computer Systems Sciences* 24 (1983), 362–381.

[90] J. Spencer, "Ten Lectures on the Probabilistic Method," *Society for Industrial and Applied Mathematics* (1987).

[91] T. H. Spencer, "More time-work tradeoffs for parallel graph algorithms," *Proc. 3rd Annual ACM Symposium on Parallel Algorithms and Architectures* (1991), 81–93.

[92] R. Tamassia and F.P. Preparata, "Dynamic maintenance of planar digraphs, with applications," *Algorithmica* 5 (1990), 509–527.

[93] R. Tamassia and I.G. Tollis, "Dynamic reachability in planar digraphs," *Theoretical Computer Science* (1993), (to appear).

[94] R. Tamassia and J.S. Vitter, "Parallel transitive closure and point location in planar structures," *SIAM Journal on Computing* 20 (1991), 708–725.

[95] J. Ullman, *Computational Aspects of VLSI*, Computer Science Press, Rockville, MD, 1984.

[96] J. D. Ullman and M. Yannakakis, "High-probability parallel transitive-closure algorithms," *SIAM Journal on Computing* 20 (1991), 100–125.

[97] J. Valdes, R.E. Tarjan, and E.L. Lawler, "The recognition of series parallel digraphs," *SIAM Journal on Computing* 11 (1982), 298–313.

[98] L. Valiant, "Universality considerations in VLSI circuits," *IEEE Transactions on Computers* C-30 (1981), 135–140.

[99] S. Vavasis, "Automatic domain partitioning in three dimensions," Department of Computer Science, Cornell, Technical Report 90-1082, 1990.

[100] K. Weihe, "Maximum (s,t)-flows in planar networks in $O(V \log V)$ time," Dept. of Mathematics, T.U. Berlin, Technical report, 1994, (to appear in the proceedings of FOCS 1994).

[101] S. Whitesides, "Forms of hierarchy: A selected bibliography," *Gen. Syst.* 14 (1969), 3–15.