Combine and Conquer

Robert F. Cohen
Ph.D. Dissertation

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-92-45
October 1992

Combine and Conquer

by
Robert F. Cohen
A. B., Brandeis University, 1981
M. S., Boston University, 1988
Sc. M., 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 1992

© Copyright 1992
by
Robert F. Cohen

Vita

Bob Cohen was born in Boston, Massachusetts on December 13, 1959. He received his A.B. in Mathematics from Brandeis University in 1981. After graduation, he bought a suit, joined IBM, and stayed a decade. He held a number of positions at IBM in systems development, marketing, and research. In 1983 he began his graduate education, taking courses at night. He received a M.S. in Computer Science from Boston University in May 1988. Later that year, Bob became a full-time graduate student in the Computer Science Department at Brown University. He completed his Ph.D. in 1992.

Bob Cohen now lives with his family in Belmont, Massachusetts. He hopes that this will be the year that the Red Sox finally win their next World Series.

Dedication

To my children:
Michael Enrique Cohen
and
Samantha Ruth Cohen

Abstract

The development of dynamic algorithms and data structures is a challenging area of research that has received much attention in the last years. For example, generalized techniques have been developed to dynamize large classes of geometric algorithms. In the area of graph algorithms such techniques are lacking. The goal of this thesis is to demonstrate generalized techniques to maintain the solutions of dynamic algorithms for graph problems, and to present dynamic algorithms based on our techniques.

We provide a framework, called *tree attribute system*, for maintaining the values of attributes on trees in a fully-dynamic environment. Our technique extends and generalizes the dynamic trees of Sleator and Tarjan and the decomposable search problems of Overmars. We use this technique to show two new dynamic data structures, *linear expression tree* and *linear attribute grammar*, for maintaining the solutions to tree-based expressions. These data structures are used to present fully-dynamic algorithms for a large class of problems on tree-width two graphs, a class of graphs which include trees and series-parallel graphs. Additionally, we present a framework for the dynamic drawing of planar graphs, and demonstrate a number of fully-dynamic algorithms to draw trees, series-parallel graphs, planar st-graphs, and general planar graphs.

Acknowledgments

I am grateful to my friend and advisor Roberto Tamassia for his guidance and assistance. I am fortunate to have worked with him. All research in this thesis was performed jointly with Roberto.

Additionally, I appreciate the involvement of the following colleagues of mine. The research in chapter 4 was done jointly with S. Sairam and Jeff Vitter. The research in chapter 5 was done jointly with Paola Bertolazzi, Pino Di Battista, and Yannis Tollis.

I thank my thesis committee, Jeff Vitter and Franco Preparata, for their helpful suggestions and comments. I also thank Mary Andrade and Phoebe Martone of the Computer Science Department for their help during all stages of the preparation of this dissertation.

I am deeply grateful to my wife, Amy Prashker, for her consistent support and comfort. I hope she will finally be able to make all the clothing purchases she postponed until I got a job.

I am fortunate to have wonderful family and friends who have been there for me during this undertaking. In particular, I thank my oldest and most trusted advisor, my mother, Ruth Cohen, for everything. I also thank my brothers, Jeff and Larry Cohen, for all their help. I am grateful to my in-laws, Bob and Helena Prashker, for lending me a computer so I could do much of this work at home. Additionally, the following friends have listened to my kvetching and have given their support: Jeff Achter, Kirk Beaty, Randy Callistri-Yeh, Tina Cantor, Lisa Cronin, Mark Freidberg, Herb Kay, Beth Kerr, Sridhar Ramaswamy, Steve Siegal, Dan Steinberg, Theresa St. John-Siegal, Kevin Walton, Kelly Washburn, and Wayne Wong.

Finally, I thank my father, Sumner Cohen, who died while I was at Brown. Though not a computer scientist, he worked hard to read and understand an early version of my research. I wish he could read this dissertation.

Contents

Vita ii

Dedication iii

Abstract iv

Acknowledgments v

1 Introduction 1
- 1.1 Dynamic Data Structures . . . 1
- 1.2 Overview . . . 1

2 Dynamic Graph Algorithms 5
- 2.1 Dynamic Tree Algorithms . . . 5
- 2.2 Transitive Closure . . . 6
- 2.3 Connectivity . . . 6
- 2.4 Shortest Path . . . 7
- 2.5 Minimum Spanning Tree . . . 8
- 2.6 Planarity Testing . . . 8

3 Attribute Systems 9
- 3.1 Introduction . . . 9
- 3.2 Attribute Systems on Paths and Trees . . . 10
  - 3.2.1 Path Attribute Systems . . . 10
  - 3.2.2 Tree Attribute Systems . . . 18
- 3.3 Linear Expression Trees . . . 31
- 3.4 Linear Attribute Grammars . . . 35
- 3.5 Applications . . . 40
  - 3.5.1 Tree Structures for Graph Problems . . . 40
  - 3.5.2 Summation Heaps . . . 40
  - 3.5.3 Blocking Heaps . . . 41
  - 3.5.4 Point Location in Unbalanced Binary Space Partition Trees . . . 42
  - 3.5.5 Slicing Floorplan Compaction . . . 44

4 Bounded Tree-Width Graphs **47**
4.1 Introduction 47
4.2 Tree Width Two Graphs 48
4.2.1 Preliminaries 49
4.2.2 Biconnected Series-Parallel Graphs 51
4.2.3 Representing Tree Width Two Graphs 54
4.2.4 Dynamic Operations 58
4.2.5 Auxiliary Operations 68
4.2.6 Dynamic Operations for Series-Parallel Graphs and Trees 73
4.3 Other Dynamic Problems 76
4.3.1 All Connectible-Pairs Shortest Path / Minimum Cut 76
4.3.2 Shortest *st*-Path / Minimum *st*-Cut in Series-Parallel Digraphs . 78
4.3.3 Common Neighbor 81
4.3.4 Minimum (Maximum) Spanning Tree 82

5 Dynamic Graph Drawing **86**
5.1 Introduction 86
5.1.1 Definitions 86
5.1.2 Model 87
5.1.3 Overview 88
5.2 Dynamic Tree Drawing 89
5.2.1 $\Box$-drawings 89
5.2.2 Dynamic Environment 90
5.2.3 Data Structure 93
5.2.4 Dynamic Operations 94
5.3 Series Parallel Digraphs 98
5.3.1 Δ-drawings 98
5.3.2 Dynamic Environment 100
5.3.3 Data Structure 102
5.3.4 Dynamic Operations 105
5.4 Planar Graphs 110
5.4.1 Upward Drawings 111
5.4.2 Visibility Drawings 114
5.4.3 Biconnected Planar Graphs 115

6 Conclusions and Open Problems **118**

Bibliography **120**

List of Tables

3.1 The rules to calculate the values of operator $\odot_r$. 23
3.2 Restructuring done by operation *push* 23
3.3 The equations to calculate the compaction of floorplans. 46

4.1 The equations to calculate the values of *mis*. 68

5.1 The equations for tree drawing. 93
5.2 The equations for drawing a P-node. 103
5.3 The equations for drawing a P_Q-node. 104
5.4 The equations for drawing an S-node. 104
5.5 The equations for drawing a Q-node. 105

List of Figures

3.1 The rotations performed in operation *splaystep* 16
3.2 The path-tree for a path attribute system. 19
3.3 Expanding a node of unbounded degree. 28
3.4 A node in an unbounded degree tree. 29
3.5 A linear expression tree with two binary linear operators. 32
3.6 An example of the concatenation of summary graphs. 39
3.7 A binary space partition of the plane 43
3.8 Finding a query point in a binary space partition 44
3.9 An example of a slicing floorplan 45

4.1 A tree-width two graph. 50
4.2 Making a cross pair the terminals of a biconnected series-parallel graph. 53
4.3 Example of the final step of the induction in the proof of lemma 4.4 . . 54
4.4 Representation of a tree-width two graph that meets the weight invariant 56
4.5 The SPQC-tree for a tree-width two graph. 57
4.6 A tree decomposition associated with an S-node. 58
4.7 The items considered in operation *InsertEdge*. 63
4.8 Inserting an edge from v' to v''. 64
4.9 Restructuring after edge insertion. 65
4.10 Restoring the weight invariant after edge insertion. 66
4.11 The forbidden homeomorphic subgraph for a series-parallel digraph . . . 80

5.1 Geometric constructions in the $\Box$-algorithm 91
5.2 Geometric construction of a Δ-drawing 99
5.3 A pictoral representation of the calculations of *position*. 106
5.4 Finding the face containing a query point p. 108
5.5 Updating the SPQ-tree — Part 1 110
5.6 Updating the SPQ-tree — Part 2 111

Chapter 1

Introduction

1.1 Dynamic Data Structures

The development of dynamic algorithms and data structures is a challenging area of research that has received much attention in the last years. Dynamic computation involves updating the solution to a problem when the input changes incrementally. The goal of an efficient dynamic algorithm is to achieve considerable savings over recalculating the solution from scratch. In the area of graph problems, dynamic algorithms have attracted recent interest, motivated by many important applications in network optimization, VLSI layout, computational geometry, and distributed computing. We consider dynamic algorithms where processing is done *on-line*, which means that a sequence of update and query operations are performed over time, and we must complete each operation before beginning the next.

Generalized techniques have been developed to dynamize large classes of geometric algorithms (summarized in [81]). In the area of graph algorithms such techniques are lacking. The goal of this thesis is to demonstrate generalized techniques to maintain the solutions of dynamic algorithms for graph problems, and to present dynamic algorithms based on our techniques.

Throughout this thesis, n and m denote the number of vertices and edges of the graph being considered, respectively.

1.2 Overview

The remaining four chapters of this thesis are organized as follows. Chapter 2 presents a survey of dynamic graph algorithms. The past decade has seen much interest in the development of these algorithms. However, for many problems, the search for efficient dynamic algorithms have been elusive. Research has focused on six areas.

- *Tree algorithms* are concerned with maintaining values associated with nodes of a tree. For example, suppose T is the tree representing an arithmetic expression $\mathcal{E}$. Each node has a value associated with a subexpression of $\mathcal{E}$. Dynamic tree

algorithms were initially introduced as internal structures to solve maximum flow problems, but since have been used in a wide range of dynamic algorithms.

- *Transitive closure* queries answer questions of the form "Is there a directed path from a vertex u to a vertex v in given directed graph?" This problem is also referred to as *reachability*.

- *Connectivity* is a fundamental property of graphs. There are two types of connectivity queries. A *k-vertex connectivity* (or simply *k-connectivity*) query consists of determining if there are k vertex-disjoint paths between a pair of vertices. Similarly, a *k-edge connectivity* query consists of determining if there are k edge disjoint paths between a pair of vertices.

- The *shortest path* problem on a directed graph consists of answering queries of the type "What is the length of a shortest path between a pair of vertices?" Sometimes the path itself is also requested.

- A *spanning tree* of a connected graph G is an edge-induced subgraph T of G such that T is a tree and every vertex of G is in T. Suppose the edges of G are weighted, then a *minimum spanning tree* is a spanning tree of minimum total weight. A minimum spanning tree query determines if a given edge is in the correct minimum spanning tree. Another query returns the weight of a minimum spanning tree.

- Suppose we are given a planar graph G. A *planarity testing* query determines if the graph that results from inserting an edge e in graph G is planar.

Chapter 3 introduces a generalized dynamization technique for trees, called *tree attribute system*. We show how to maintain the values of attributes of nodes, paths, and subtrees of a tree T. For example, suppose T is an expression tree. A node attribute for a leaf λ of T is the value of the variable associated with λ. A node attribute for an internal node μ is the operator performed at μ. A tree attribute for μ is the value of the subexpression represented by μ. A path attribute for a path Π from node σ to node τ gives the dependency of the value of the subexpression represented by τ on the value of the subexpression represented by σ. The title of this thesis, *Combine and Conquer* comes from this data structure. The important property of tree attribute systems is the ability to determine the values of tree attributes from the combination of node and path attributes.

We provide an extensive collection of query and update operations. Queries include returning the value of attributes. Additionally, we extend the decomposable search problems of [81] to trees in order to find a distinguished node of a tree. Updates include changing the values of node attributes. We also describe update operations that alter the structure of trees. *Linking* trees consists of adding and edge from the root of tree T_1 to a node of tree T_2. The inverse operation *Cutting*, separates one tree into two by removing the edge from a node to its parent. *Expanding* a node inserts an additional node μ between a node ν and a consecutive collection $\mu_i, \ldots, \mu_j$ of children

of ν. Node μ becomes a child of ν and nodes $\mu_i, \ldots, \mu_j$ become children of node μ. The inverse operation *Contracting* a node μ with parent node ν deletes μ and replaces μ with the children of μ in the order of children of ν. Finally, *everting* tree T at node μ makes μ the root of T by reversing the parent-child relationship on the path from μ to the root of T. Our algorithm performs queries and updates in $O(\log n)$ time using $O(n)$ space. This is a significant extension of the *dynamic trees* of Sleator and Tarjan [100].

We introduce two new data structures based on tree attribute systems. Suppose $\mathcal{S}$ is a semiring and r is a constant. A *linear expression tree* is a tree-based expression where each node μ contains an attribute taken from $\mathcal{S}^r$. The values of these attributes are calculated by a linear operation of the values of the attributes at the children of μ. A *linear attribute grammar* is also a tree based expression over $\mathcal{S}^r$. Here, the values of the attributes at a node μ are calculated from the values of the attributes at the parent, siblings and children of μ. All dependencies are linear. We implement both data structures as tree attribute systems. Therefore, we can maintain the values of attributes in a linear expression tree or a linear attribute grammar in $O(\log n)$ time per query and update using $O(n)$ space.

Chapter 4 uses a tree attribute system to develop fully dynamic data structures to answer a large class of queries on tree-width two graphs. Many of these queries are NP-complete in general. The concept of bounded tree-width graph was introduced by Robertson and Seymour [94] as a generalization of series-parallel graphs. Graphs of tree-width two have been extensively studied owing to their role in fault-tolerant communication networks [43,44,78,119], concurrent broadcasting in common medium networks [23], reliability evaluation in complex systems [3], and evaluation of queries in relational data base systems [4]. Note that tree-width two graphs include the important subclasses of trees and series-parallel graphs. Bounded tree-width graphs also have theoretical interest, since a large number of generally NP-complete problems have polynomial time solutions when restricted to graphs of a bounded tree-width [94]. While extensive research has been accomplished to find parallel algorithms to recognize bounded tree-width graphs [12,71,76], no equivalent dynamic result has been presented.

We demonstrate a fully-dynamic data structure to maintain the decomposition of tree-width two graphs. The data structure requires $O(m)$ space for a graph with m edges. Updates require $O(\log^2 m)$ time, while queries can be performed in $O(\log m)$ time. Update times can be reduced to $O(\log m)$ time if we restrict ourselves to series-parallel graphs. Supported queries include the following decision problems that are generally NP-complete: Vertex Cover, Dominating Set, Chromatic Number, Partial Feedback Edge Set, Independent set, Bipartite Subgraph, Maximum Cut, Minimum Maximal Matching, Partition into Triangles, Partition into Hamiltonian Subgraphs, Partition into Cliques, Monochromatic Triangle, Transitive Subgraph, Cubic Subgraph, Kernel, and Chromatic Index. Additionally, we support the solutions to the common neighbor, minimum spanning tree, and all-connectible pairs shortest path problems.

Chapter 5 presents fully-dynamic algorithms for drawing planar graphs. Applications can be found in a variety of areas including circuit layout, network management, software engineering, and graphics. The motivation for investigating dynamic graph

drawing algorithms arises when very large graphs need to be visualized in a dynamic environment, where vertices and edges are inserted and deleted, and subgraphs are displayed. Previous work [77] only considers trees and presents a technique that restructures the drawing of a tree in time proportional to its height, and hence linear in the worst case.

We further the study of dynamic graph drawing by devising a model for dynamic graph drawing algorithms. Our model is based on performing queries and updates on an implicit representation of the drawing. Our drawings maintain certain aesthetic qualities, such as planar (no edge crossings), upward (directed edges oriented in the positive y-direction), and grid (vertices drawn at integer coordinates). We present several efficient dynamic drawing algorithms for trees, series-parallel digraphs, planar st-digraphs, and general planar graphs.

Given a planar drawing, a point location query determines the face, edge or vertex containing a query point p. A *window* W is a rectangle on the plane with edges parallel to the axes. A window query returns the location of the edges and vertices of G that are drawn inside W. We support window queries in our drawing of trees and series-parallel digraphs, and point location queries in our drawing of series-parallel digraphs.

Our algorithms produce drawings with $O(n^2)$ area. Using a linear attribute grammar, we find that updates take $O(\log n)$ time, while most queries require $O(k \log n)$ time to draw k vertices and edges. Window queries on the drawing of series-parallel digraphs require $O(k \log^2 n)$ time to draw k vertices and edges.

Finally, Chapter 6 presents conclusions and some open problems.

Chapter 2

Dynamic Graph Algorithms

In this chapter, we survey past research in dynamic graph algorithms. We consider dynamic algorithms where processing is done *on-line*, which means that a sequence of update and query operations are performed over time and we must complete each operation before beginning the next one. Performance is measured in storage space and time complexity for queries and updates. Time complexity is either worst-case or *amortized*. Suppose we are given an initial configuration of a data structure and q operations that result in some final configuration. Amortized analysis of a dynamic algorithm consists of determining the worst-case time complexity $T(n, q)$ to perform these operations. The amortized time to perform each operation is then $T(n, q)/q$. For a full discussion of amortized analysis, see [115].

A dynamic algorithm or data structure is *semi-dynamic* if the repertory of update operations consists of only "insertions" or "deletions", while a *fully dynamic* algorithm supports an intermixed sequence of insertions and deletions. For many problems, efficient semi-dynamic algorithms are known, even though no efficient fully-dynamic counterpart has been found.

A general lower bound technique for incremental algorithms, with applications to dynamic graph algorithms, is discussed in [8].

2.1 Dynamic Tree Algorithms

A fundamental data structure for the dynamization of graph algorithms is the dynamic tree. Suppose we are given a collection of trees such that each node has associated numeric values such as a weight. Dynamic trees maintain these values under update operations such as:

- *Linking* — add an edge from the root of tree T_1 to a node of tree T_2.
- *Cutting* — separate one tree into two by removing the edge from a node to its parent.
- *Everting* — make a node μ the root of tree T by reversing the parent-child relationship on the path from μ to the root of T.

and query operations such as:

- *Least Common Ancestor* — return the least common ancestor of two tree nodes.
- *Find Minimum* — Find the minimum weight node on a path.

Dynamic trees were introduced as internal data structures in sequential maximum flow algorithms [24,55,100]. Since then, a large number of dynamic algorithms [27,28, 48,49,50,51,69,83,122] have used dynamic trees as part of their data structures. Initial data structures [24,55] based on balanced binary trees (e.g. AVL-trees [1] or Red-Black trees [57]), take $O(\log^2 n)$ time per operation. Sleator and Tarjan improve this to $O(\log n)$ time per operation by basing their data structure on the weight-biased binary trees [7].

2.2 Transitive Closure

Given a directed graph G and two vertices u and v of G, a *transitive closure* query determines if there is a directed path in G from u to v. This is also referred to as a *reachability* query. A basic data structure is presented in [63]. Each transitive closure query takes $O(1)$ time, while a sequence of additions takes $O(n^3)$ time, which amortizes to $O(n)$ time per edge addition in dense digraphs. A sequence of deletions takes $O(n^2(n+m))$ time, which amortizes to $O(n^2)$ time per edge deletion in dense digraphs. The data structure uses $O(n^2)$ space. Semi-dynamic algorithms were presented which improve this result. A data structure for arbitrary digraphs that support edge insertions in amortized $O(n)$ time, queries in $O(1)$ time, and use $O(n^2)$ space are given in [64]. An extension of this result to finding regular paths for a regular language is given in [14]. A semi-dynamic algorithm for acyclic digraphs that support edge deletions in in amortized $O(n)$ time, queries in $O(1)$ time, and use $O(n^2)$ space is presented in [65]. La Poutré and van Leeuwen [85] present two semi-dynamic data structures supporting transitive closure on general digraphs. The first supports edge insertions in amortized $O(n)$ time. The second supports edge deletions in amortized $O(n^2)$ time. Both data structures use $O(n^2)$ space and can be used to answer transitive closure queries in $O(1)$ time.

Better results can be achieved for specific classes of digraphs. In particular, there are fully dynamic data structures for dynamic reachability in *planar st-graphs* [87, 108], *spherical st-graphs* [112], and *series-parallel digraphs* [66]. Each of these require $O(\log m)$ time per query and update using $O(m)$ space.

2.3 Connectivity

Connectivity is a fundamental property of graphs. Dynamic algorithms have been presented for both vertex and edge connectivity. Given a graph G with vertices u and v, the *k-vertex connectivity* (or simply *k-connectivity*) query consists of determining if there are k vertex disjoint paths between u and v. Similarly, the *k-edge connectivity* query consists of determining if there are k edge disjoint paths between u and v.

Vertices u and v are 1-connected, or simply *connected* if there is a path between u and v. This is the undirected version of the reachability query. If we allow only edge insertions, this problem reduces to the set union-find problem [116], where a sequence of q queries and insertions takes $O(qn + m\alpha(m,n))$ time, where $\alpha(m,n)$ is a very slowly increasing function [116]. Therefore, we have amortized $O(\alpha(m,n))$ time per operation.

The fully dynamic data structure of [48] takes $O(\sqrt{m})$ time per operation. This is the best result handling edge deletions. Other semi-dynamic techniques supporting only edge deletions are shown in [41], where a data structure is presented supporting constant time queries and $O(q + mn)$ time to perform q edge deletions; and in [89] where q edge deletions are performed in $O(mg + m \log m)$ time for a graph of genus g.

These results have been extended to the maintenance of 2, 3, and 4-connectivity. Each of the following is implemented with a semi-dynamic data structure, supporting only insertions. Suppose we perform a sequence of q queries or edge insertions starting from an empty graph. For 2-connectivity, Westbrook and Tarjan [120,122] present two algorithms. The first is a simple algorithm which takes $O(q + n \log n)$ time. The second is an optimal algorithm with amortized $O(q\alpha(q,n))$ time per edge insertion. In the case of 3-connectivity, Di Battista and Tamassia [28] give an algorithm which requires a total of $O(q + n \log n)$ time in general. If we start with an initially biconnected graph, then the algorithm runs optimally in amortized $O(\alpha(q,n))$ time per dynamic operation. La Poutré [83,84] extends this to an optimal algorithm with amortized $O(q\alpha(q,n))$ time per edge insertion for general graphs. For 4-connectivity, the algorithm of [69] takes $O(q + n \log n)$ time.. If we start with an initially triconnected graph, then the algorithm runs optimally in amortized $O(\alpha(q,n))$ time per dynamic operation.

The results of [83,120,122] also apply to 2-edge connectivity. To perform q dynamic operations takes $O(q\alpha(q,n))$ time. A fully dynamic data structure for 2-edge connectivity is presented in [50,53]. In general, dynamic operations take each $O(m^{2/3})$ time. For planar graphs this time bound is reduced to $O(\sqrt{n \log\log n})$. Recently, Frederickson [49] presented a fully dynamic data structure such that queries take $O(\log n)$ time, and edge insertions and deletions are performed in $O(\sqrt{m})$ time. If we restrict the class of graphs to planar graphs, then edge insertions and deletions can be performed in $O(\log^3 n)$ time.

The only dynamic result higher order edge connectivity insertion-only semi-dynamic algorithm for maintaining 3-edge connectivity by Galil and Italiano [50,54]. The data structure takes $O((n+q)\alpha(q,n))$ time to perform q dynamic operations.

2.4 Shortest Path

The *shortest path* query on a digraph G consists of answering queries of the type "What is the length of a shortest path from v_1 to v_2?" Sometimes the path itself is also requested. The fully dynamic data structures for the shortest path query presented in [40,95] have $O(n^2)$ space, constant query time, $O(n^2)$ time for edge insertion, and $O(mn + n^2 \log n)$ time for edge deletion. Such performance bounds, especially for deletion, are quite unsatisfactory and indicate the difficulty of the query. The best

known semidynamic data structures supporting insertions in digraphs with unit edge lengths use $O(n^2)$ space and have constant query time; the total time to process all edge insertions is $O(n^3 \log n)$, which amortizes to $O(n \log n)$ time per insertion for dense graphs [6,73].

2.5 Minimum Spanning Tree

A *spanning tree* of a connected graph G is an edge-induced subgraph T of G such that T is a tree and every vertex of G is in T. (If G is not connected then the spanning trees of each connected component form a *spanning forest*.) If the edges of G are weighted, then a *minimum spanning tree* of G is a spanning tree of minimum total weight. Dynamic updates include changing edge weights, and inserting and deleting edges and vertices. Queries include determining if a given edge is in the minimum spanning tree and returning the total weight of a minimum spanning tree. The dynamic maintenance of minimum spanning trees has the interesting property that, after an update operation, at most one edge needs to be replaced in the minimum spanning tree.

Early data structures for this problem [19,102] have $O(n)$ space, $O(n)$ time for edge insertions, $O(n^2)$ time for edge deletions and the changing of edge weights, and $O(1)$ time queries. The best result for general graphs is [48], which presents a fully-dynamic data structure with $O(\sqrt{m})$ time per update and $O(m)$ space. When only considering planar graphs, the time bound is reduced to $O(\log^2 n)$ time per update. For fixed embeddings of planar graphs, this result is improved to $O(\log n)$ time per dynamic operation in [39]. The semi-dynamic data structure of [27] for embedding-independent, biconnected planar graphs uses $O(n)$ space and supports insertions in $O(\log n)$ time (amortized for edge-insertions).

2.6 Planarity Testing

In a static environment we can test planarity and compute a planar embedding in optimal $O(n)$ time (see, e.g. [13,42,47,60,72]). The dynamic maintenance of a planar embedding under a sequence of edge insertions can be done in time $O(\log n)$ per operation [107]. Di Battista and Tamassia [27] have investigated how to test if a new edge can be added to a planar graph G so that G remains planar, and how to add vertices and edges so that planarity is preserved. The semidynamic data structure of [27] for biconnected graphs uses $O(n)$ space and supports queries and insertions in $O(\log n)$ time (amortized for edge-insertions). The extension to general graphs is reported in [28]. Westbrook [121] has improved this result to achieve $O(\alpha(m, n))$ amortized time per operation. This time bound is deterministic for queries and expected for updates. The randomization is due to the use of on dynamic perfect hashing [31]. Recently, a fully dynamic planarity testing technique with $O(n^{2/3})$ query and update time has been discovered [52].

Chapter 3

Attribute Systems

3.1 Introduction

In the past years considerable progress has been made in the development of dynamic algorithms for geometric searching problems (see, e.g., the survey paper [16]). However, considerably fewer results exist on the dynamization of graph algorithms. A crucial development in the area of dynamic geometric searching has been the identification of general methods that apply to a large class of problems. In particular, the techniques developed by Overmars et al. (summarized in [81]) for the class of decomposable search problems constitute a fundamental contribution toward the dynamization of large classes of geometric problems. The availability of such general techniques appears instead to be lacking in the area of dynamic graph algorithms. This chapter provides such generalized techniques in the realm of dynamic graph problems.

Our approach is motivated by the observation that a number of dynamic graph algorithms [27,28,48,49,50,51,69,83,122], developed mostly for connectivity problems, appear to be based on the following fundamental idea: Decompose a graph into subgraphs with limited overlap, and represent such a decomposition by means of a tree so that dynamic operations on the graph are reflected into corresponding dynamic tree operations, which are in turn supported by variations of the link-cut trees of Sleator and Tarjan [100].

We associate values, called *attributes*, with the nodes, paths, and subtrees of our trees. Path attributes form a *path attribute system*, if they are maintained in constant time under path concatenation. Additionally, attributes form a *tree attribute system* if the tree attributes of the tail of a path Π are determined in constant time from the path attributes of Π.

We also introduce a new data structure called a *linear attribute grammar*. An *attribute grammar* is a tree-based expression where the values at a node μ are calculated from the values at the parent, siblings, and/or the children of μ. A linear attribute grammar, is an attribute grammar where all dependencies are linear.

The results of this chapter can be summarized as follows:

- We provide a framework for maintaining attribute systems on trees in a fully

dynamic environment. Our technique extends and generalizes the dynamic trees of [100].

- We show that given a semiring $\mathcal{S}$, a set of linear expressions with binary and unary operators over $\mathcal{S}^k$ can be dynamically maintained in a fully dynamic environment using linear space and logarithmic time per operation.
- We show that a linear attribute grammar can be dynamically maintained in a fully dynamic environment using linear space and logarithmic time per operation. Linear attribute grammars can be used as the data structure for dynamic algorithms for several problems in graph drawing (see Chapter 5).

The rest of this chapter is organized as follows. Section 3.2 describes fully dynamic algorithms to maintain attributes on paths and trees. Separate algorithms are presented for trees of bounded and unbounded degrees. Section 3.3 applies the results of the previous section to present a fully dynamic algorithm for maintaining the solutions of linear expressions. Section 3.4 presents a fully dynamic algorithm for linear attribute grammars. Finally, section 3.5 presents other applications including two types of generalized heaps.

3.2 Attribute Systems on Paths and Trees

In this section we present a framework for dynamically maintaining an important class of path and tree attributes. We introduce the concept of path attribute systems and tree attribute systems which significantly extend and generalize the dynamic trees of [100]. We begin by discussing dynamic algorithms on a collection of paths, and then show that trees can be maintained as a collection of paths.

3.2.1 Path Attribute Systems

Paths are directed. The first and last nodes of a path Π are called the *head* and *tail* of Π, respectively, and are denoted with $head(\Pi)$ and $tail(\Pi)$. The *reversed path* $\overline{\Pi}$ of Π is the path obtained by reversing all edges of Π. Formally, we have:

Definition 3.1 A *path* is a collection of nodes and directed edges such that:

- The empty path with no nodes is a path.
- A single node μ is a path with $head(\mu) = tail(\mu) = \mu$.
- If Π' and Π'' are paths then the concatenation of Π' and Π'' is a path Π formed by adding a directed edge from $tail(\Pi')$ to $head(\Pi'')$.

Definition 3.2 A *node attribute* N is a function on nodes. The values N can take are arbitrary, but can be stored in $O(1)$ space. A *path attribute* P is a function on paths. The value of $P(\Pi)$ for a path Π is dependent on the values of $N(\mu)$ for each node μ

on Π, and on the order of the nodes of Π. The value of $P(\Pi)$ can be stored in $O(1)$ space.

Definition 3.3 If the values of a node attribute N are taken from a monoid, then N is said to be *globally updatable*, otherwise N is *locally updatable*. We define $\mathcal{N}_G(\mu)$ and $\mathcal{N}_L(\mu)$ to be the set of globally and locally updatable node attributes of μ.

Definition 3.4 A *node attribute set* $\mathcal{N}$ is a finite collection of node attributes $N_1, \ldots, N_r$. Similarly, a *path attribute set* $\mathcal{P}$ is a finite collection of path attributes $P_1, \ldots, P_s$. For a node μ, the *value* of $\mathcal{N}(\mu)$ is the vector $(N_1(\mu), \ldots, N_r(\mu))$. For a path Π, the value of $\mathcal{P}(\Pi)$ is the vector $(P_1(\Pi), \ldots, P_s(\Pi))$.

For each path Π, we consider the node attribute sets $\mathcal{N}(\mathit{head}(\Pi))$ and $\mathcal{N}(\mathit{tail}(\Pi))$ to be included in path attribute set $\mathcal{P}(\Pi)$.

Definition 3.5 Suppose $\mathcal{N}$ is a node attribute set $\mathcal{P}$ is a path attribute set, and $F : \mathcal{P} \times \mathcal{P} \to \mathcal{P}$ is a function . The triple $(\mathcal{N}, \mathcal{P}, F)$ is a *path attribute system* $\mathcal{Q}$ if for any path Π that is the concatenation of paths Π' and Π'', $\mathcal{P}(\Pi)$ can be determined in $O(1)$ time as $F(\mathcal{P}(\Pi'), \mathcal{P}(\Pi''))$. Function F is called the *concatenation function* of $\mathcal{Q}$.

Example 3.1 Suppose for each node μ we keep the following node attribute:

- $\mathit{weight}(\mu)$ — the real-valued weight of node μ.

Additionally, we have the following path attributes for a path Π:

- $\mathit{sum}(\Pi)$ — the sum of the weights of the nodes of Π.
- $\mathit{squaredsum}(\Pi)$ — the square of the sum of the weights of the nodes of Π.

Suppose path Π is the concatenation of paths Π' and Π''. If our path attribute set $\mathcal{P}(\Pi)$ consists solely of $\mathit{squaredsum}(\Pi)$, then we can not find a concatenation function to maintain $\mathcal{P}(\Pi)$, since we cannot determine $\mathit{squaredsum}(\Pi)$ from $\mathit{squaredsum}(\Pi')$ and $\mathit{squaredsum}(\Pi'')$. However, if $\mathcal{P}(\Pi)$ is the pair $(\mathit{sum}(\Pi), \mathit{squaredsum}(\Pi))$, then we can use the concatenation function F where:

$$\begin{aligned} \mathit{sum}(\Pi) &= \mathit{sum}(\Pi') + \mathit{sum}(\Pi'') \\ \mathit{squaredsum}(\Pi) &= (\mathit{sum}(\Pi))^2 \end{aligned}$$

□

Each node attribute N has a *secondary value* $N^*(\mu)$ in addition to its "normal" *primary value* $N(\mu)$. For node attribute set $\mathcal{N} = (N_1, \ldots, N_r)$, $\mathcal{N}^*$ denotes $(N_1^*, \ldots, N_r^*)$. We also have corresponding secondary values for path attributes, and similarly define $\mathcal{P}^*(\Pi)$ for a path Π.

A decomposable search problem [81] π on a set D, locates a distinguished element $x = \pi(D)$ such that given any partition of D into subsets D' and D'', we can determine in constant time if x is in D' or D''. A number of methods are presented to maintain the solution for a decomposable search problem.

We extend this notion to paths as follows:

Definition 3.6 Given a path attribute system $\mathcal{Q} = (\mathcal{N}, \mathcal{P}, F)$, a *path-selection query* Q for $\mathcal{Q}$ maps a path Π and a query argument q into a node $\mu = Q(\Pi, q)$ of Π. Suppose path Π is the concatenation of Π' and Π''. A *path-selection function* $S(\Pi, q)$ for Q, determines in $O(1)$ time whether μ is in Π' or Π'' from q and the values $\mathcal{P}(\Pi')$ and $\mathcal{P}(\Pi'')$.

We consider a rather general dynamic environment for a set of paths equipped with a path attribute system $\mathcal{Q}$ and a collection $\mathcal{S}$ of path-selection functions. Update operations include: *splitting*, *concatenating*, and *reversing paths*; and *updating node attributes*; and *exchanging* primary and secondary attribute values for all the nodes on a path. Query operations include: *evaluating* node and path attributes, and *computing path-selection queries*.

We allow both "local" and "global" updates of node attributes. A *local update* consists of changing the value $N(\mu)$ of a single node μ. A *global update* can be performed on a globally updatable node attribute N with values taken from a monoid with operator $\odot$, and consists of applying an incremental change δ to the attribute N of every node in a path Π. Formally, a global update sets $N(\mu) = N(\mu) \odot \delta$ for every node μ of Π, where we assume that $x \odot y$ can be computed in $O(1)$ time. If a path attribute P depends on a globally updatable node attribute N, we require that the new value of $P(\Pi)$ can be computed in $O(1)$ time given the variation δ of N in Π.

We support the following repertory of operations on a set of (uniquely identified) paths equipped with a path attribute system $\mathcal{Q} = (\mathcal{N}, \mathcal{P}, F)$ and a collection $\mathcal{F}$ of path-selection functions for $\mathcal{Q}$:

- *makepath*(*node* ν; *value* x) — return a new path consisting of a single node μ and set node attribute set $\mathcal{N}(\nu)$ to be x.
- *deletepath*(*node* ν) — remove the single-node path consisting of node ν.
- *getpath*(*node* ν) — return the unique identifier of the path containing ν.
- *head*(*path* Π) — return the head of path Π.
- *tail*(*path* Π) — return the tail of path Π.
- *before*(*node* μ, *integer* k) — return the k nodes preceding μ on *getpath*(μ). If there are fewer than k nodes preceding μ, then return the nodes found. If μ is the head of its path then return *nil*.
- *after*(*node* μ, *integer* k) — return the k nodes succeeding μ on *getpath*(μ). If there are fewer than k nodes succeeding μ, then return the nodes found. If μ is the tail of its path then return *nil*.
- *precedes*(*node* μ, ν) — return **true** if nodes μ and ν are on the same path with μ preceding ν. Otherwise, returns **false**.

- *pathreport*(*path* Π) — report the value of path attribute set $\mathcal{P}$ for path Π.
- *nodereport*(*node* μ) — report the value of node attribute set $\mathcal{N}$ for node μ.
- *concatenate*(*path* Π', Π'') — concatenate paths Π' and Π'' to get path Π.
- *split*(*node* μ) — split path Π into three subpaths by removing the edges from μ to *before*$(\mu, 1)$ and μ to *after*$(\mu, 1)$.
- *reverse*(*path* Π) — reverse the direction of path Π, i.e., set $\Pi = \overline{\Pi}$.
- *exchange*(*path* Π) — exchange the values $\mathcal{N}(\mu)$ and $\mathcal{N}^*(\mu)$ for each node μ on Π.
- *localupdate*(*node* μ, *nodeattribute* N, *value* x) — set the value of node attribute N of $\mathcal{N}$ to be x.
- *globalupdate*(*path* Π, *nodeattribute* N, *value* δ) — for each node μ on path Π, set $N(\mu) = N(\mu) \odot \delta$, where N is a globally updatable node attribute with monoid operator $\odot$.
- *find*(*path* Π, *selectionfunction* S, *value* q) — find node μ of Π returned by the path-query associated with path-selection function S using query argument q. If query argument q is omitted, then we assume that S does not require a query argument.

Our data structure consists of representing each path Π as a balanced binary tree $\mathcal{B}_\Pi$, called the *path-tree* of Π, and identifying Π by the root ζ of $\mathcal{B}_\Pi$. Each leaf λ of $\mathcal{B}_\Pi$ represents a node of Π, and the left-to-right order of the leaves of $\mathcal{B}_\Pi$ corresponds to the head-to-tail order of the nodes of Π. We store at λ the values $\mathcal{N}(\lambda)$ and $\mathcal{N}^*(\lambda)$. For a globally updatable node attribute N_i of $\mathcal{N}$, the value stored at λ is not the actual value of N_i, but a "partial value" that can be used to compute $N_i(\lambda)$, as will become apparent later.

Each internal node η of $\mathcal{B}_\Pi$ represents the subpath $\Pi(\eta)$ of Π associated with the leaves in the subtree of η. Note that we identify path $\Pi(\eta)$ with η. We store at η pointers *head*(η) and *tail*(η) to the head and tail of $\Pi(\eta)$. Additionally, we store at node η the four values $\mathcal{P}(\Pi(\eta))$, $\mathcal{P}(\overline{\Pi(\eta)})$, $\mathcal{P}^*(\Pi(\eta))$, and $\mathcal{P}^*(\overline{\Pi(\eta)})$. Notice that we can consider the pointers *head* and *tail* as part of path attribute set $\mathcal{P}$.

For the path attributes that depend on globally updatable node attributes, we store partial values. We also keep in η a vector $\Delta(\eta)$ (and a secondary vector $\Delta^*(\eta)$) such that for any globally updatable node attribute N_i of $\mathcal{N}$ the i-th component $\Delta_i(\eta)$ represents a change to the value of N_i for each node μ on $\Pi(\eta)$. (If N_j is a locally updatable node attribute, then we conventionally set $\Delta_j(\eta) = 0$.) Therefore, for any node μ of Π, $N_i(\mu)$ is the combination (using operation $\odot_i$, the monoid operator for values of N_i) of the "partial value" of N_i stored at μ and of the "offsets" $\Delta_i(\eta)$ of all the nodes η on the path from μ to the root ζ of $\mathcal{B}_\Pi$. Similarly, $\mathcal{P}(\mu)$ can be computed from the partial value of $\mathcal{P}$ stored at μ and the "offsets" $\Delta(\eta)$ of all the nodes η on

the path from μ to ζ in $\mathcal{B}_\Pi$. (Analogous arguments apply to the reverse and secondary values of the attributes.)

At each node μ of a path Π, we store pointers *left*(μ) and *right*(μ) to the left and right neighbors of μ on Π. At each path-tree node η we store pointers to *head*($\Pi(\eta)$) and *tail*($\Pi(\eta)$). We keep at node μ two globally updatable bits *reversed*(μ) and *exchanged*(μ). At each path tree node η, we keep the offset values *reversed*(η) and *exchanged*(η). If *reversed*(μ) = 1, then *left*(μ) points to the right neighbor of μ and *right*(μ) points to the left neighbor of μ. Similarly, if *exchanged*(μ) = 1, then the meaning of $\mathcal{N}(\mu)$ and $\mathcal{N}^*(\mu)$ is swapped. The meaning of the partial values *reversed*(η) and *exchanged*(η) is similar. If *reversed*(η) = 1, then *head*(η) points to *tail*($\Pi(\eta)$) and *tail*(η) points to *head*($\Pi(\eta)$). Also, *reversed*(η) = 1 swaps the meaning of the left and right child pointers in the subtree of η. The values of these two bits also determine the actual meaning of the four values kept for $\mathcal{P}$ at η.

Operations *makepath* and *deletepath* are trivially implemented in $O(1)$ time. Operation *pathreport*(Π) is performed in $O(1)$ time and consists of returning the value of $\mathcal{P}$, computed from the "partial value" stored at the root ζ of $\mathcal{B}_\Pi$ and the vector $\Delta(\zeta)$. Operations *reverse*(ζ) and *exchange*(ζ) are implemented by flipping the *reversed* and *exchanged* bits at η. Operations *head*(ζ) and *tail*(ζ) determine the correct node to return based on the value of *reversed*(ζ) and the *head* and *tail* pointers stored at ζ. Operation *globalupdate*(Π, N_i, δ) is performed by setting $\Delta_i(\zeta) = \Delta_i(\zeta) \odot \delta$, where ζ is the root of the path-tree of Π. Each of these operations is implemented in $O(1)$ time.

Operation *getpath*(μ) is implemented by following pointers from μ to the root of the path-tree containing μ. Operation *before*(μ, k) is implemented by calling operation *push* on each node of the path from *getpath*(μ) to μ. The immediate predecessor of μ is then pointed at by *left*(μ). We use the *left* and *right* pointers in the following way to return the remaining (at most) $k-1$ predecessors of μ: Suppose we have returned $i < k$ nodes with μ_{i-1} and μ_i the $i-1$st and ith nodes returned. Then one of *left*(μ_i) and *right*(μ_i) will point to μ_{i-1}. The other will point to the predecessor of μ_i. (We cannot simply follow the *left* pointers, since each μ_i only stores a partial value of *reversed*(μ_i).) The implementation of operation *after*(μ, k) is symmetric. Each of these operations takes $O(d_\mu + k)$ time where d_μ is the depth of μ in $\mathcal{B}_\Pi$. Note that for a path of length ℓ, d_μ is $O(\log \ell)$.

Operation *precedes*(μ, ν) is implemented by finding the least common ancestor η of μ and ν in the path tree containing both μ and ν. We then calculate the reversal state of η to determine which of μ and ν is a descendant of the left child of η. If μ is a descendant of the left child then return **true** else return **false**.

The following operation moves the values of Δ towards the leaves of a path-tree in $O(1)$ time:

- *push*(*node* η) — For internal path-tree node η with children η' and η'' combine $\Delta(\eta')$ and $\Delta(\eta'')$ with $\Delta(\eta)$ and set $\Delta(\eta) = 0$.

Operation *push*(η) is implemented by using the monoid operator for each globally updatable node attribute to apply the partial value at η to the partial values at η' and η''. We also swap values depending on *reversed*(η) and *exchanged*(η). If *reversed*(η) =

1, then we swap pointers $head(\eta)$ and $tail(\eta)$, the values $\mathcal{P}(\eta)$ and $\overline{\mathcal{P}(\eta)}$, and the values $\mathcal{P}^*(\eta)$ and $\overline{\mathcal{P}^*(\eta)}$. If $exchanged(\eta) = 1$, then we swap the values $\mathcal{P}(\eta)$ and $\mathcal{P}^*(\eta)$, and the values $\overline{\mathcal{P}(\eta)}$ and $\overline{\mathcal{P}^*(\eta)}$. If η is a tree node μ, then it is a leaf of its path tree. If $reversed(\mu) = 1$, we swap pointers $left(\mu)$ and $right(\mu)$. If $exchanged(\mu) = 1$, then we swap the values $\mathcal{N}(\mu)$ and $\mathcal{N}^*(\mu)$. (Equivalently, we can include the pointers *left* and *right* in node attribute set $\mathcal{N}$ and the pointers *head* and *tail* in path attribute set $\mathcal{P}$. In this way, operation *push* would only swap attribute sets, not individual pointers.)

Therefore, operation $nodereport(\mu)$ is implemented by finding the path from μ to the root ζ of the path-tree containing μ and then calling *push* along this path starting at ζ. In this way, the partial values of the globally updatable node attributes of $\mathcal{N}(\mu)$ will be the actual values. Therefore, we then return $\mathcal{N}(\mu)$.

Operations *concatenate* and *split* use the following *elementary tree operations* which can be performed in $O(1)$ time:

- $join(node\ \zeta', \zeta'')$ — Given the roots ζ' and ζ'' of the path-trees for solid paths Π' and Π'', combine the trees into a new tree by creating a new root ζ with left child ζ' and right child ζ''. We set the value of $\Delta(\zeta)$ to be 0. We set the *left* and *right* neighbor pointers at $tail(\zeta')$ and $head(\zeta'')$ to point at each other. Choose the one of *left* or *right* that is initially *nil* at $tail(\zeta')$ and set that value to $head(\zeta'')$. If both *left* and *right* are *nil* at $tail(\zeta')$ then we can determine in constant time the value of *reversed* at $tail(\zeta')$, so we set the *right* pointer to be $head(\zeta')$. We perform a similar update at $head(\zeta'')$. We calculate the values of $\mathcal{P}$ stored at ζ in $O(1)$ time using concatenation function F.

- $separate(node\ \zeta)$ — Given the root ζ of a binary tree, first call $push(\zeta)$ then divide the tree into two trees with roots ζ' and ζ'', where ζ' is the root of the left subtree and ζ'' is the root of the right subtree. The values of $\mathcal{P}$ at ζ' and ζ'' do not change.

- $rotateleft(node\ \eta)$ ($rotateright(node\ \eta)$) — Perform a left (right) rotation at node η. This operation can be performed with $O(1)$ *separate* and *join* operations.

We conclude operation $split(\mu)$ by calling $push(\mu)$. This step is not needed to maintain the values of paths, yet is needed in our implementation of trees (see section 3.2.2).

Operation $localupdate(\mu, x)$ is implemented by first replacing the value of $\mathcal{N}(\mu)$ with x. Suppose path tree $\mathcal{B}$, with root ζ, contains node μ. Suppose η is a path tree node in $\mathcal{B}$. The value of $\mathcal{P}(\eta)$ may change only if η is on the path from μ to ζ in $\mathcal{B}$. Therefore, we calculate these values starting at μ and ending at ζ. This takes $O(d_\mu)$ time where d_μ is the depth of μ in $\mathcal{B}$.

We need the following operation from [101] in our implementation of $find(\Pi, S, q)$:

- $splaystep(node\ \eta)$ — For binary tree $\mathcal{B}$ with root ζ and node η a grandchild of ζ, restructure $\mathcal{B}$ such that the relative order of the leaves of $\mathcal{B}$ remain fixed, and every node in the subtree rooted at η has its depth reduced by one or two. We accomplish this as follows. Let η' be the child of ζ that is the parent of η. If η'

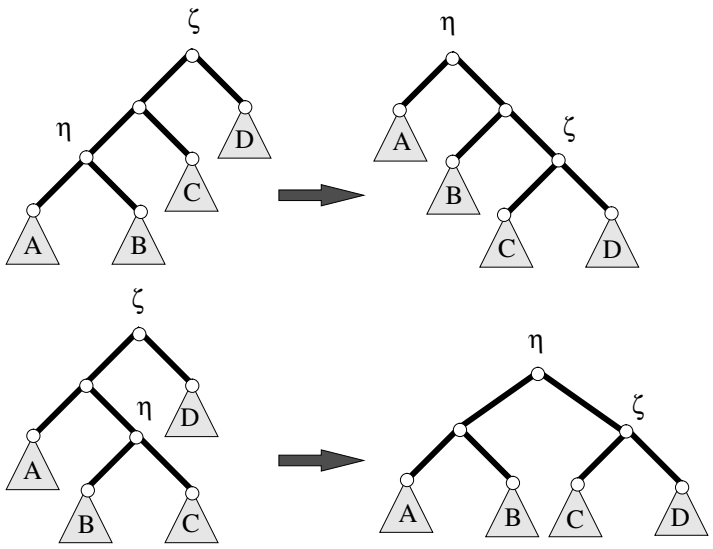

Figure 3.1: The rotations performed in operation *splaystep*(η).

and η are both left children then perform *rotateright*(ζ) twice. If η' is a left child and η is a right child, then perform *rotateleft*(η') followed by *rotateright*(ζ). The other two cases are symmetric. (See Fig. 3.1.)

Operation *splaystep* is implemented with a constant number of *rotateleft* and *rotateright* operations. Hence, *splaystep* takes $O(1)$ time.

Lemma 3.1 *Suppose S is a path-selection function for path attribute set $\mathcal{P}$ and ζ is the root of a path-tree Π. Then given query argument q, we can determine in $O(1)$ time if $S(\Pi, q)$ is a child or grandchild of ζ, or which grandchild of ζ is the root of the subtree containing $S(\Pi, \zeta)$.*

Proof: Suppose η_1 and η_2 are the children of ζ. By definition, in $O(1)$ time we can determine which subtree rooted at η_1 or η_2 contains $S(\Pi, q)$. If the subtree found is a single node, then it is $S(\Pi, q)$. Without loss of generality, suppose we know that $S(\Pi, q)$ is in the subtree rooted at η_1, and η_1 is not a leaf. Then let η_{11} and η_{12} be the children of η_1. We then perform $O(1)$ rotations on $\mathcal{B}_\Pi$ such that η_{11} becomes the left child of the root. Then, in $O(1)$ time we can determine if $S(\Pi, q)$ is in the subtree rooted at η_{11}. Otherwise, $S(\Pi, q)$ is in the subtree rooted at η_{12}. □

Operation *find*(ζ) is implemented as follows, starting at node ζ and repeating until a value is returned: if μ is a child or grandchild of ζ then return μ. Otherwise, determine the subtree rooted at the grandchild η of ζ which contains μ. Do *splaystep* at η and recur. After μ is found, we undo all the *splaystep*s to restore the balance of the path-tree. This takes $O(d_\mu)$ time where d_μ is the depth of the returned node μ.

Theorem 3.1 *Let $\mathcal{Q} = (\mathcal{P}, \mathcal{N}, F)$ be a path attribute system and $\mathcal{S}$ a collection of path-selection functions for $\mathcal{Q}$. There is a fully dynamic data structure for maintaining $\mathcal{Q}$ and $\mathcal{S}$ over a set of paths with the following performance:*

1. *a path of length ℓ uses $O(\ell)$ space;*
2. *operations* makepath, deletepath, head, tail, reverse *and* exchange *each take $O(1)$ time;*
3. *operations* before *and* after *each take $O(\log \ell + k)$ time to return k nodes from a path of length ℓ;*
4. *operations* getpath, concatenate, split, precedes, localupdate, globalupdate, pathreport, nodereport *and* find *each take $O(\log \ell)$ time for a path of length ℓ.*

Example 3.2 Suppose we associate an attribute $cost(\mu)$ with each node μ of a path Π. We want to answer the following query in a dynamic environment (including operations *reverse*, *concatenate*, and *join*):

- *FindMin*(*path* Π) — return the minimum cost node of path Π closest to $head(\Pi)$.

We keep the following value for each path Π:

- *mincost*(*path* Π) — the minimum cost of any node of path Π.

Therefore, we keep path attribute system $\mathcal{Q} = (\mathcal{N}, \mathcal{P}, F)$, where for node μ and path Π, $\mathcal{N}(\mu) = (cost(\mu))$ and $\mathcal{P}(\Pi) = (mincost(\Pi))$. Suppose path Π is the concatenation of paths Π' and Π''. Concatenation function F calculates $mincost(\Pi) = \min(mincost(\Pi'), mincost(\Pi''))$.

Operations *FindMin* is the selection query associated with the following path selection function.

Selection Function S_1 *Suppose η is the root of a path tree with children η' and η''. If* mincost(η') = mincost(η) *then return η', else return η''.*

Operation *FindMin*(Π) is then implemented as *find*(Π, S_1). □

Example 3.3 Suppose each node is associated one of a finite number of possible types. We wish to support the following query.

- *FindLongest*(*path* Π, *nodetype* X) — return the tail of the longest subpath Π' of Π such that $head(\Pi') = head(\Pi)$ and Π' does not contain a node of type X.

We keep the following path attribute system $\mathcal{Q} = (\mathcal{N}, \mathcal{P}, F)$. For each node μ, node attribute set $\mathcal{N}(\mu)$ consists of the single node attribute $type(\mu)$, the type of node μ. For path Π, path attribute set $\mathcal{P}(\Pi)$ consists of the set of boolean valued path attributes $contains(\Pi, X)$. The value of $contains(\Pi, X)$ is **true** if path Π contains a node of type X. Suppose path Π is the concatenation of paths Π' and Π''. Concatenation function

F sets *contains*(Π, X) to be **true** if either *contains*(Π', X) is **true** or *contains*(Π'', X) is **true**.

We utilize the following path selection function with query argument X.

Selection Function S_2 *Suppose η is the root of a path tree with children η' and η''. If* contains(η', X) *is* **true**, *or if* head(η'') *has type X, then return η', else return η''.*

Query *FindLongest*(Π, X) is then implemented as follows. If *head*(Π) has type X, then return *nil*. Otherwise, return *find*(Π, S_2, X). □

Example 3.4 We consider a problem that arises in dynamically maintaining trees (see Section 3.2.2). Consider paths whose nodes have positive real weights. A node μ is *heavy* if the weight of μ is greater than the sum of the weights of the predecessors of μ on Π. Each path Π has at least one heavy node since *head*(Π) is heavy. We want to answer the following queries in a dynamic environment (including operations *reverse*, *concatenate*, and *join*):

- *totalpathwt*(*path* Π) — return the total weight of all the nodes in path Π.
- *heavy*(*path* Π) — return the heavy node μ of Π closest to *tail*(Π).

Let the *tilt* of a node μ of path Π be the sum of the weights of the nodes preceding μ in Π, minus the weight of μ. Therefore a node μ is heavy if $tilt(\mu) < 0$. Let *mintilt*(Π) be the minimum value of *tilt* among all nodes of Π. Notice that *totalpathwt*(Π) = *totalpathwt*$(\overline{\Pi})$, but in general *negtiltnode*(Π) is different from *negtiltnode*$(\overline{\Pi})$.

To solve this problem we keep the following path attribute system $\mathcal{Q} = (\mathcal{N}, \mathcal{P}, F)$: the node attribute of a node μ is the weight of μ, i.e., $\mathcal{N}(\mu) = (weight(\mu))$; the path attributes of a path Π are $\mathcal{P}(\Pi) = (totalpathwt(\Pi), mintilt(\Pi))$ (see Fig. 3.2). Suppose path Π is the concatenation of paths Π' and Π''. Concatenation function F calculates $\mathcal{P}(\Pi)$ as follows:

$$\begin{aligned} totalpathwt(\Pi) &= totalpathwt(\Pi') + totalpathwt(\Pi'') \\ mintilt(\Pi) &= \min(mintilt(\Pi'), totalpathwt(\Pi') - weight(head(\Pi'')), \\ &\quad totalpathwt(\Pi') + mintilt(\Pi'')) \end{aligned}$$

Operation *totalpathwt*(Π) is implemented with *pathreport*(Π). Operation *heavy*(Π) is implemented as a selection-query via the following path-selection function without query argument.

Selection Function S_3 *Suppose η is the root of a path tree with children η' and η''. if* totalpathwt(Π') − weight(head$(\Pi''))$ < 0 *or* totalpathwt(Π') + mintilt$(\Pi'') < 0$ *then return η''. Otherwise, return η'.* □

3.2.2 Tree Attribute Systems

We now discuss a general class of dynamic algorithms on trees. We consider rooted ordered trees with edges directed from the child to parent. Hence, a path in a tree is always directed from a descendant to an ancestor.

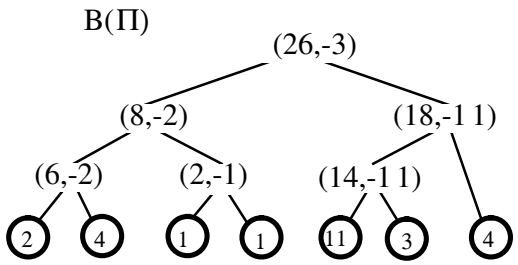

Figure 3.2: Path-tree for the path attribute system of Example 3.4. Each leaf is labeled with its weight. Each internal node η is labeled with $\mathcal{P}(\eta) = (totalpathwt(\eta), mintilt(\eta))$.

Definition 3.7 Rooted tree T has *bounded degree* if there is a constant d such that no node of T has more than d neighbors, otherwise it is of *unbounded degree*. Tree T is *ordered* if for each node μ the left to right order of the children of μ is fixed.

If T is a tree and μ is a node of T we use the notation T_μ to indicate the subtree rooted at μ.

We develop our dynamic algorithm for trees in the same manner as for paths. We introduce the concept of tree attribute systems, and show a large collection of dynamic operations on a tree storing a tree attribute system.

Definition 3.8 A node attribute set $\mathcal{N}$ for a tree T consists of a finite collection of node attributes for the nodes of T. Similarly, a path attribute set $\mathcal{P}$ for T consists of a fixed collection of path attributes for the paths of T. A *tree attribute* R is a function on trees such that, for a tree T, $R(T)$ depends on the values $\mathcal{N}(\mu)$ for each node μ on T, on the parent-child relationships between nodes of T, and on the order of the children of each node of T. We assume that $R(T)$ can be stored in $O(1)$ space. A *tree attribute set* $\mathcal{R}$ is a fixed collection of tree attributes. For a node μ of tree T, $\mathcal{R}(\mu)$ denotes the tree attribute set of the subtree of T rooted at μ.

We consider both bounded and unbounded degree trees. We first present the algorithm for bounded degree trees, then extend this result to unbounded degree trees.

Bounded Degree Trees

In this section, we present a fully dynamic algorithm for tree attribute systems on bounded degree trees. We begin by formally defining a tree attribute system.

Definition 3.9 Suppose $\mathcal{N}$ is a node attribute set, $\mathcal{P}$ is a path attribute set, $\mathcal{R}$ is a tree attribute set, and F is a concatenation function. Given a bounded degree tree T and a path Π of T, we denote with $\tilde{\mathcal{N}}$ the extended node attribute set that, for a node μ of Π, consists of $\mathcal{N}(\mu)$, $\mathcal{R}(\mu')$ for each child μ' of μ not on Π, and the ordering of the children of μ. We say that the 4-tuple $\mathcal{Z} = (\mathcal{N}, \mathcal{P}, \mathcal{R}, F)$ is a *tree attribute system* if:

- The triple $(\tilde{\mathcal{N}}, \mathcal{P}, F)$ is a path attribute system for the set of paths of T (where

concatenations are restricted to paths Π' and Π'' in T such that $head(\Pi'')$ is the parent of $tail(\Pi')$ in T).

- The value of $\mathcal{R}(head(\Pi))$ can be determined in constant time from the values $\mathcal{P}(\Pi)$, $\tilde{\mathcal{N}}(head(\Pi))$, and $\tilde{\mathcal{N}}(tail(\Pi))$.

We extend the concept of selection to trees. That is we define a tree selection function, which, if available, allows us to quickly find a distinguished node of tree T.

Definition 3.10 Given a tree attribute system $\mathcal{Z} = (\mathcal{N}, \mathcal{P}, \mathcal{R}, F)$, a *tree-selection query* Q for $\mathcal{Z}$ maps tree T and a query argument q into a node $\mu = Q(T, q)$ of T. Suppose path Π is the concatenation of Π' and Π'' in T such that $tail(\Pi)$ is the root of T. Suppose node μ' is the node closest to $tail(\Pi)$ such that μ is a descendant of μ'. Then *tree-selection function* $S(\Pi, q)$ for Q is a path-selection function to find node μ'.

Definition 3.11

Consider a tree T containing a node ν with children $\mu_1, \ldots, \mu_d$ in left-to-right order. Suppose Π is a path of T containing both ν and some child μ_i. The *left-tree* of ν with respect to Π, $T_\ell(\nu)$, is defined as follows. The root of $T_\ell(\nu)$ is a node ν_ℓ with $\mathcal{N}(\nu_\ell) = \mathcal{N}(\nu)$. Root ν_r has children $\mu_1, \ldots, \mu_{i-1}$. The *right-tree* of ν with respect to Π, $T_r(\nu)$, is defined similarly. The root of $T_r(\nu)$ is a node ν_r with $\mathcal{N}(\nu_r) = \mathcal{N}(\nu)$. Root ν_r has children $\mu_{i+1}, \ldots, \mu_d$.

We provide three types of global restructuring of trees: *reflecting*, *everting*, and *cycling*. Consider a tree T. We allow a node μ of T to be *fixed* indicating that the structure of the subtree rooted at μ may not be restructured.

Definition 3.12

Given a tree T and a node μ of T that is not fixed, reflecting T at μ consists of reversing the left-to-right order of the children of μ and the children of all descendants that can be reached from μ without passing through a fixed node.

Definition 3.13

Given a tree T with node μ not the root, *everting* T at μ consists of reversing the parent-child relationships between all nodes on the path Π from μ to the root of T. If Π contains a fixed node, then everting at μ is not allowed.

Each node μ also keeps an *eversion rule* which describes changes to the order of the children of μ. Possibilities include:

- *Simple* — the left and right-trees of μ remain unchanged.
- *Steady* — the left and right-trees of μ are exchanged.
- *Mirrored* — the left and right-trees of μ are reflected.

Consider a node μ, its child μ', and its parent ν, all on path Π. Simple eversion at μ substitutes ν for μ' in the ordering of the children of μ. Steady eversion at μ maintains the clockwise order of the neighbors of μ. Mirrored eversion at μ reverses the clockwise

order of the neighbors of μ. Each node of a tree T may use a different eversion rule. Therefore, we consider the eversion rule a node property.

Definition 3.14 Given a tree T and a node μ and its parent ν of T that is not fixed, cycling ν at μ consists of cyclicly permuting the children of ν such that node μ is the final node in the order.

Our algorithm supports the following operations on a collection of trees with a tree attribute system $\mathcal{Z} = (\mathcal{N}, \mathcal{P}, \mathcal{R})$ and a collection $\mathcal{F}$ of path-selection and tree-selection functions:

- *Evaluate*(*node* μ) — Return the values of $\mathcal{N}(\mu)$ and $\mathcal{R}(\mu)$.
- *PathFind*(*node* μ', μ''; *selectionfunction* S) — Let Π be the path of a tree from node μ' to node μ''. Find the node of Π returned by the path-selection function S.
- *TreeFind*(*node* ν; *selectionfunction* S) — Find the node of the subtree rooted at μ returned by the tree-selection function S.
- *LocalUpdate*(*node* μ, *nodeattribute* N, *value* x) — Set the value of node attribute N of $\mathcal{N}$ to be x.
- *GlobalUpdate*(*node* μ', μ'', *nodeattribute* N, *value* δ) — Let Π be the path from μ' to μ'' in T. For each node μ on path Π, set $N(\mu) = N(\mu) \odot \delta$, where $\odot$ is the operator of the monoid for N.
- *MakeTree*(*value* x) — Create a new tree consisting of a single leaf whose node attribute set has value x.
- *DeleteTree*(*node* λ) — Remove the single-node tree consisting of leaf λ.
- *LCA*(*node* μ', μ'') — Returns the least common ancestor of nodes μ' and μ''. Nodes μ' and μ'' are assumed to be nodes of the same tree.
- *Evert*(*node* μ) — Let T be the tree containing node μ. Evert T at μ.
- *Link*(*node* ρ, μ, μ', μ'') — This operation assumes that nodes μ and ρ are in different trees, and ρ is the root of its tree T. Add an edge from ρ to μ with μ' its immediate left sibling and μ'' its immediate right sibling. If either μ' or μ'' is omitted then ρ becomes the first or last child of μ. Tree T becomes a subtree of the tree containing μ.
- *Cut*(*node* μ) — This operation assumes that μ is not the root of a tree. Remove the edge from μ to its parent, thus separating the subtree rooted at μ.
- *Parent*(*node* μ) — Return the parent of μ or *nil* if μ is a tree root.
- *Sibling*(*node* μ, *integer* k, *direction* X) — Return the k adjacent siblings of node μ in the X direction, where X is either *left* or *right*. If there are fewer than k siblings in the X direction, then return the nodes found. If there are no siblings in the X direction, then return *nil*.

- *Reflect*(*node* μ) — Reflect the subtree rooted at μ.
- *Cycle*(*node* ν, μ) — Cycle the children of ν such that node μ is the final node in the order. Node ν is assumed to be a child of node ν.

Our data structure consists of representing an n-node tree T as a collection of disjoint paths. We partition the edges of T into *solid* or *dashed* such that at most one solid edge is incoming into a node. Therefore, every node is in exactly one maximal path of solid edges (of length 0 or more), called a *solid path* of T.

The partition of edges into solid and and dashed is obtained by means of the following *size invariant*: let $size(\mu)$ denote the number of nodes in the subtree of T rooted at node μ. An edge (μ, ν) of T is solid if $size(\mu) > size(\nu)/2$. If the partition satisfies the size invariant, then every path of T has $O(\log n)$ dashed edges.

In order to achieve the logarithmic time per dynamic operation, we use biased search trees [7] to represent the path-trees. For a node ν of a solid path Π we define the node-attribute $weight(\nu)$ as one plus the sum of the sizes of the subtrees connected to μ by dashed edges. For each solid path Π, we also define the corresponding path attribute $weight(\Pi)$ as the sum of the weights of the nodes of Π. The path-tree $\mathcal{B}_\Pi$ is then implemented as a search tree biased by the node-weights of its leaves.

The size invariant may be temporarily violated by the algorithms that operate on the solid paths of T. Hence, we set-up a mechanism to restore it, using a path attribute system on the solid paths.

Consider a node μ of T on solid path Π. We keep a globally updatable node attribute $reflected(\mu)$ which indicates the *path reflection status*, which is the left-to-right direction of the children of μ, ignoring any reflection done by ancestors of $tail(\Pi)$. At each path tree node η, we keep the associated offset value $reflected(\eta)$. Rather than use the *exchange* bits for paths, we use the *reflected* bits which are handled somewhat differently. We also keep a locally updatable node attribute $fixed(\mu)$ which indicates if node μ is fixed.

Even though the values for *reflected* are boolean, we do not use simple boolean addition as the monoid operator for calculating the actual value of $reflected(\mu)$, for a node μ. At a path tree node η, we consider $reflected(\eta)$ to be part of a pair of values $(a(\eta), b(\eta))$, where $a(\eta) = reflected(\eta)$ and $b(\eta) = fixed(tail(\eta))$. The monoid operator $\odot_r$ on these pairs is described in table 3.1. The value $(1, 1)$ is not allowed since we do not allow reflection of a fixed node. Note that operator $\odot_r$ does not modify the value of $fixed(tail(\eta))$.

Let μ be a node of tree T. Let ν be the node closest to the root of T such that ν is reached by a dashed edge on the path Π from μ to the root of T, and ν is a descendant of the closest ancestor of μ that is fixed. If μ has no fixed ancestors, then μ will be the parent of the last dashed edge of Π. The *reflection status* of node μ is then calculated by finding the exclusive-or of the path reflection status of μ and the nodes reached over dashed edges on the path from μ to ν.

We use the *reflected* bits much like the *exchanged* bits of section 3.2.1. Let $\mathcal{Z} = (\mathcal{N}, \mathcal{P}, \mathcal{R})$ be the tree attribute system we are maintaining for tree T. For node μ, let

η'	η''	$\eta' \odot_r \eta''$
$(x,0)$	(y,z)	$(x+y,0)$
$(0,1)$	(y,z)	$(0,1)$

Table 3.1: The rules to calculate the values of operator $\odot_r$. Variables x, y, and z take boolean values. Operator + is boolean addition.

	Initial Value		*Restructured*	
Rule	*reversed*	*reflected*	$T_\ell(\mu)$	$T_r(\mu)$
any	0	0	$T_\ell(\mu)$	$T_r(\mu)$
any	0	1	$T^*_r(\mu)$	$T^*_\ell(\mu)$
simple	1	0	$T_\ell(\mu)$	$T_r(\mu)$
simple	1	1	$T^*_r(\mu)$	$T^*_\ell(\mu)$
steady	1	0	$T_r(\mu)$	$T_\ell(\mu)$
steady	1	1	$T^*_\ell(\mu)$	$T^*_r(\mu)$
mirrored	1	0	$T^*_\ell(\mu)$	$T^*_r(\mu)$
mirrored	1	1	$T_r(\mu)$	$T_\ell(\mu)$

Table 3.2: Restructuring done by operation $push(\mu)$ for tree node μ with left-tree $T_\ell(\mu)$ and right-tree $T_r(\mu)$. Eversion rules are given in definition 3.13. The notation $T^*_\ell(\mu)$ and $T^*_r(\mu)$ indicate the reflections of $T_\ell(\mu)$ and $T_r(\mu)$.

$\mathcal{R}^*(\mu)$ be the tree attribute set for the reflection of μ. We have defined the (primary) extended node attribute set $\tilde{\mathcal{N}}(\mu)$ to be the union of $\mathcal{N}(\mu)$ and $\mathcal{R}(\mu')$ for each child μ' of μ not on Π. The secondary extended node attribute set $\tilde{\mathcal{N}}^*(\mu)$ is the union of $\mathcal{N}(\mu)$ and $\mathcal{R}^*(\mu')$ for each child μ' of μ not on Π. Then for each path tree node η we keep the four values $\mathcal{P}(\Pi(\eta))$, $\mathcal{P}(\overline{\Pi(\eta)})$, $\mathcal{P}^*(\Pi(\eta))$, and $\mathcal{P}^*(\overline{\Pi(\eta)})$, the meaning of which depend on the values of $reversed(\eta)$ and $reflected(\eta)$.

We keep a value in $\mathcal{P}(\Pi)$ called $containsfixed(\Pi)$ indicating if path Π contains a fixed node. If so, then the values of $\mathcal{P}(\overline{\Pi(\eta)})$ and $\mathcal{P}^*(\overline{\Pi(\eta)})$ are undefined.

For node μ on path Π in tree T, we extend the restructuring done by operation $push(\mu)$. If the value of $reversed(\mu) = 1$ or $reflected(\mu) = 1$ prior to the call to $push(\mu)$, then we may swap the left and right-trees of μ, reflect the left and right trees of μ, or both. The restructuring done is based on the eversion rule for μ as described in Definition 3.13. Table 3.2 describes this restructuring. For bounded degree tree T, we swap the left and right-trees of μ by reordering the children of μ. We reflect the left (right) tree of μ by reversing the order of the children of μ in the left (right) tree and then flipping the *reflected* bit at the roots of the path trees containing the children of μ in the left (right) tree. Since T has bounded degree, we can perform all this in $O(1)$ time.

Now, we show how to perform operation *Evaluate*(μ). Let Π be the solid path of length ℓ containing μ. If μ is the tail of Π then, by the definition of tree attribute systems, we can use path attribute system $(\tilde{\mathcal{N}}, \mathcal{P})$ to find $\mathcal{R}(\mu)$ in $O(1)$ time. Otherwise, we make μ the tail of a path by calling *split*(μ). We then restore path Π with the *concatenate* operation. Hence, this can all be accomplished in $O(\log \ell)$ time.

Operation *LocalUpdate*(λ, x) affects only the values of the nodes in the path from λ to the root ρ. If such path contains dashed edges, then we temporarily convert it into a solid path by changing dashed edges to solid and solid edges to dashed such that any node continues to have at most one incoming solid edge. This is done with the following operations, derived from dynamic trees [100]:

- *splice*(*path* Π) — This operation assumes that Π is a solid path ending at $\mu \neq \rho$. Convert the dashed edge leaving μ to solid and convert the solid edge (if it exists) entering the parent ν of μ to dashed. Let Π' be the solid path containing ν and μ' be the sibling of μ with the solid edge (μ', ν). To implement *splice*(Π) we need to convert edge (μ', ν) from solid to dashed. Let Π'' be the resulting solid path starting at ν. We obtain Π'' by splitting Π'. We complete *splice*(Π) by concatenating Π and Π''.

- *expose*(μ) — Convert to dashed the solid edge entering μ, if such edge exists. Create a solid path from μ to the root by converting to solid all the dashed edges (ν', ν'') of such path, and converting to dashed the edges $(\mathrm{sib}(\nu'), \nu'')$. Operation *expose*($\mu$) consists of a sequence of *splice* operations on the solid paths containing the nodes on the path from μ to ρ. This operation is always followed by a *conceal* operation, which undoes its effect.

- *conceal*(μ) — Restore the original type (solid or dashed) of the edges entering the nodes on the path Π from node μ to the root ρ. This operation is the inverse of *expose*, and also consists of a sequence of *splice* operations. The topmost node where to *splice* is given by *light*(Π), as described in Example 3.4. Operation *light*(Π) returns the node ν of Π closest to *tail*(Π) such that the weight of ν is larger than the sum of the weights of the nodes preceding ν in Π. As shown in Example 3.4, *light*(Π) is found using a path-selection function.

Since path-trees are biased by the weights, the sum of the number of elementary tree operations at each *splice* operation telescopes, so that operations *expose* and *conceal* can be performed with $O(\log n)$ elementary tree operations.

Operation *LocalUpdate*(μ, x) is realized as follows. First, we issue *expose*(μ). Next, we change the value of the constant node attribute at μ to x. Finally, we perform *conceal*(μ) to restore the original solid and dashed edges. Hence, operation *LocalUpdate* is implemented in $O(\log n)$ time.

Operation *GlobalUpdate*(μ', μ'', N, δ) first makes two calls to *expose* to get the path Π from μ' to μ''. Let Δ_N be the entry for node attribute N in the Δ vector at the root of the path-tree for Π. We set Δ_N to $\Delta_N \odot \delta$. We restore the weight invariant by calling *expose*(μ') followed by *conceal*(μ'). All this can be done in $O(\log n)$ time.

To implement $LCA(\mu', \mu'')$ we first issue *expose*(μ') followed by *expose*(μ''). If μ' and μ'' are on the same resulting solid path, then μ' is an ancestor of μ'', so we return μ''. Otherwise, when we *expose*(μ''), we split the path containing μ' at the least common ancestor. Therefore, $LCA(\mu', \mu'')$ will be the parent of the tail of the path containing μ'. We restore the size invariant by calling *conceal*(μ''), *expose*(μ'), then *conceal*(μ'). All this can be performed in $O(\log n)$ time.

For a path Π, we use the eversion rules described in table 3.2 to calculate the value of $P(\bar{\Pi})$ and $P^{(}\bar{\Pi})$. We perform operation *Evert*(μ) by first issuing *expose*(μ) to get the path Π from μ to the root ρ, flipping the reverse bit of Π, and finally calling operation *conceal*(ρ) to restore the size invariant. Therefore, operation *Evert* is performed in $O(\log n)$ time.

Operation *Reflect*(μ) is performed as follows. First, we call *expose*(μ). The left edge-path of μ then contains nodes for all the children of μ. We then reverse the order of the children of μ by flipping the *reflected* and *reversed* bits at the root of the path-tree representing $\Pi_\ell(\mu)$. Finally, we perform *conceal*(μ) to restore the original solid and dashed edges. Hence, operation *Reflect* is implemented in $O(\log n)$ time.

Operation *Cycle*(ν, μ) is performed by first calling *expose*(ν). We then reorder the children of ν. Suppose $T_r(\nu)$ is the right-tree of ν with respect to path Π before the *Cycle* operation is performed. We flip the *Reflect* bit at the roots of the path trees containing the children of ν in $T_r(\nu)$. Finally, we perform *conceal*(ν) to restore the original solid and dashed edges. All this can be done in $O(\log n)$ time.

Operation *PathFind*(μ', μ'', S, q) makes two calls to *expose* to get the path Π from μ' to μ'', then calls *find*(Π, S, q). We then restore the weight invariant by calling *expose*(ν') followed by *conceal*(ν').

Consider tree selection function S. We implement operation *TreeFind*(ν, S, q) as follows. If ν is not the root of its tree, we first call *expose* at *Parent*(ν). Node ν then becomes the tail of its solid path Π. Let μ be the node we are searching for. We repeat the following until node μ is found. Let node $\mu' =$ *find*(Π, S_p, q). Create path Π' with head μ' and tail ν. This is done with single *split* and *concatenate* operations.

Suppose $\mu_1, \ldots, \mu_d$ are the children of μ' not on Π and $\Pi_1, \ldots, \Pi_d$ are the solid paths containing each μ_j. Repeat the following until we find a child μ_i of μ' such that node μ is a descendant of μ_i. Consider child node μ_j. Let path Π' be the result of joining paths Π_j and Π'. By the definition of tree and path selection functions, we can determine in $O(1)$ time if node μ is in the subtree rooted at μ_j.

If no child of μ' has μ as a descendant, then $\mu' = \mu$. Otherwise, we reset path Π to be Π' and continue. After μ is found, we undo the restructuring to restore the path trees.

At each iteration, the time to find μ' and create Π' is $O(d_{\mu'})$, where $d_{\mu'}$ is the depth of μ' in $\mathcal{B}_\Pi$, the path tree for path Π. Operation *join* increases the depth of a path tree node by 1. Therefore, $d_{\mu'}$ is at most one more than the depth of μ' in its original path-tree. Therefore, the time to find μ is $O(\log n)$ plus the time to perform *expose*(μ). The time to restore the path trees is equivalent. Therefore operation *TreeFind* is implemented in $O(\log n)$ time.

Operations *MakeTree* and *DeleteTree* can be trivially implemented in $O(1)$ time. Operation *Parent*(μ) is performed in $O(\log n)$ time by calling *expose*(μ), returning *after*(μ) on the solid path containing μ, and calling *conceal*(μ).

Operation *Sibling*(μ, k, X) is realized as follows. If node μ is the root of T, then return *nil*. Otherwise, we call *expose*(μ). Let node ν be the parent of μ. Let ζ be the root of the resulting path tree. We call operation *push* on the path from ζ to ν. Node ν then stores the actual value of *reflected*(ν). We then return the (at most) k siblings of μ in the X direction. Hence, operation *Sibling* is performed in $O(\log n + k)$ time.

Operations *Link* and *Cut* are implemented in $O(\log n)$ time using variations of the same named dynamic tree operations. We implement operation *Link*(ρ, μ, μ', μ'') by first calling *expose*(μ). We then add ρ as a child of μ between μ' and μ''. Finally, we call *conceal*(μ) to maintain the size invariant.

Operation *Cut*(μ) is the inverse of *Link*. We first call *expose*$(Parent(\mu))$. We then remove node μ as a child of its parent by removing μ from the order of children stored at μ and destroying the dashed edge. We conclude by calling *conceal*(μ) to maintain the size invariant.

Theorem 3.2 *Let $\mathcal{Z} = (\mathcal{N}, \mathcal{P}, \mathcal{R}, F)$ be a tree attribute system, and $\mathcal{S}$ be a collection of path-selection and tree-selection functions. There is a fully dynamic data structure for maintaining $\mathcal{Z}$ and $\mathcal{S}$ over a set of trees with the following performance:*

1. *a tree of size n uses $O(n)$ space;*
2. *operations* MakeTree *and* DeleteTree *take each $O(1)$ time;*
3. *operations* Evaluate, LocalUpdate, GlobalUpdate, Link, Cut, Evert, Reflect, Cycle, *LCA*, PathFind, TreeFind, Sibling, *and* Parent *take each $O(\log n)$ time, where n is the total size of the trees involved in the operation.*

Example 3.5 A *concatenable heap* consists of a collection of *heaps* which are sets of weighted nodes that support the following operations:

- *FindMin*(*heap* H) — returns the cost of a minimum cost node in heap H.
- *InsertHeap*(*node* μ; *heap* H) — insert node μ in heap H.
- *DeleteHeap*(*node* μ; *heap* H) — delete node μ from heap H.
- *ConcatenateHeap*(*heap* H_1, H_2) — combine heaps H_1 and H_2 into a single heap.

We can implement a concatenable heap as either a path or a tree. If we store each heap as a path, then operation *FindMin*(H) is implemented as operation *FindMin* in example 3.2. Operations *InsertHeap*(μ, H), *DeleteHeap*(μ, H), and *ConcatenateHeap*(H_1, H_2) are directly implemented using operations *concatenate* and *split*.

The implementation of a concatenable heap as a tree is more complicated. We present this technique since it is used in developing more generalized heaps that can not be implemented as paths (see sections 3.5.2 and 3.5.3).

The nodes of a heap are stored as leaves of an arbitrary tree. A heap is identified as its representative tree. We maintain the following tree attribute for the nodes of tree T.

- *mincost*(μ) — the minimum cost of a node in the subtree rooted at μ.

We also maintain the following path attribute for the paths of tree T.

- *minpathcost*(Π) — the minimum cost of a leaf that is a descendant of *tail*(Π), but is not *head*(Π) or a descendant of *head*(Π).

It is easy to see that these values can be maintained as part of a tree attribute system. We implement operation *FindMin* using the following tree selection function which takes a cost q as a query argument:

Selection Function S_4 *Suppose node η the root of the path tree with children η' and η''. If* minpathcost(tail(η)) = minpathcost(tail(η')) *then return η', else return η''.*

For tree T associated with heap H, let c be the value of *mincost* at the root of T. Operation *FindMin*(H) is implemented as *TreeFind*(ρ, S_4, c). Operations *InsertHeap*(μ, H), *DeleteHeap*(μ, H), and *ConcatenateHeap*(H_1, H_2) are directly implemented using operations *Link*, *Cut*, and *MakeTree*.

Therefore, both implementations of concatenable heaps are implemented with $O(\log n)$ time per operation, where n is the total size of the heaps involved. □

Unbounded Degree Trees

The algorithm presented in the previous section for bounded degree trees takes advantage of the fact that reordering and reflecting the children of a node can be accomplished in constant time. In the case of unbounded degree trees, we need to develop additional techniques to accomplish the needed restructuring.

Definition 3.15 Suppose T is a (unbounded-degree) tree containing node ν with children $\mu_1, \ldots, \mu_d$. *Expanding* ν between μ_i and μ_j ($1 \leq i \leq j \leq d$) consists of restructuring T such that we replace nodes $\mu_i, \ldots \mu_j$ in the ordering of the children of ν with a new node μ. Node μ has children $\mu_i, \ldots \mu_j$. We set $\mathcal{N}_L(\mu) = \mathcal{N}_L(\nu)$ and $\mathcal{N}_G(\mu) = 0$. The inverse operation is called *contracting* child μ of ν. In order to contract μ, $\mathcal{N}_L(\mu)$ must equal $\mathcal{N}_L(\nu)$ and $\mathcal{N}_G(\mu)$ must equal 0.

For unbounded degree trees, we require that tree attribute set $\mathcal{R}$ meet the following *expansion invariant*: For any node ν of tree T, the value of $\mathcal{R}(\nu)$ is unchanged as a result of expanding ν or contracting a child of ν. Therefore, given an unbounded degree tree T we can create an equivalent tree T' of bounded degree 2, called a *binary expansion* of T, by expanding nodes with more than two children (see Fig. 3.3). Note that if T has n nodes, then T' will have $O(n)$ nodes.

Definition 3.16 Suppose T is an unbounded degree tree. Then $\mathcal{Z} = (\mathcal{N}, \mathcal{P}, \mathcal{R}, F)$ is a tree attribute system on T if and only if node attribute set $\mathcal{R}$ keeps the expansion invariant and $\mathcal{Z}$ is a tree attribute system on any binary expansion T' of T.

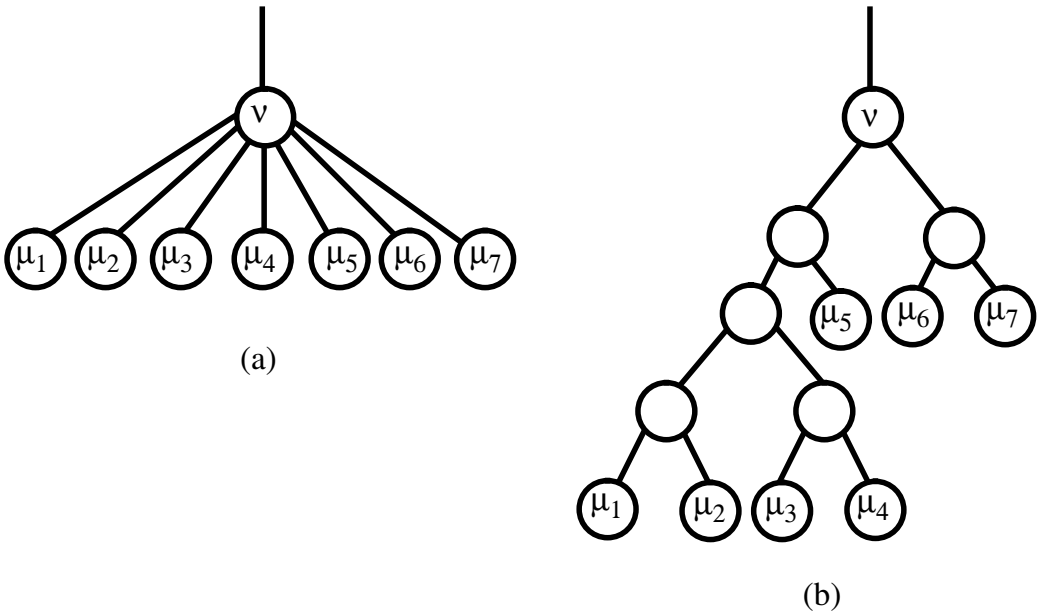

Figure 3.3: Expanding a node of unbounded degree. (a) The original node. (b) The expansion of the node.

For each node ν in an unbounded degree tree T, we keep two paths $\Pi_\ell(\nu)$ and $\Pi_r(\nu)$ associated with the children to the left and right of a specified child μ_i of ν (see Fig. 3.4). We write $\nu(\mu_j)$ to represent the node in either $\Pi_\ell(\nu)$ or $\Pi_r(\nu)$ associated with μ_j. Formally,

Definition 3.17 Consider an unbounded degree tree T containing a node ν with children $\mu_1, \ldots, \mu_d$ in left-to-right order. Suppose Π is a path of T containing ν. The *left edge-path* of ν with respect to Π, $\Pi_\ell(\nu)$, consists of a node $\nu(\mu_j)$ for each child μ_j of ν in the left-tree of ν with respect to Π. Path $\Pi_\ell(\nu)$ is directed in the left-to-right order of the children. The value of $\mathcal{N}_L(\nu(\mu_j))$ is defined to be $\mathcal{N}_L(\nu)$. The *right edge-path* $\Pi_r(\nu)$ is similarly defined for the children of ν in the right-tree of ν.

Consider an unbounded degree tree T and tree attribute system $\mathcal{Z} = (\mathcal{N}, \mathcal{P}, \mathcal{R}, F)$ on T. A path Π in T is directed from a node σ to its ancestor τ. Node σ is called the *head* of Π and is written $head(\Pi)$. Node τ is called the *tail* of Π and is written $tail(\Pi)$. Let Π be a solid path in T and $\Pi_X(\nu)$ be an edge path of some node ν. We define the extended node attribute set $\hat{\mathcal{N}}$ that, for a node μ of Π, $\hat{\mathcal{N}}(\mu)$ consists of $\mathcal{N}(\mu)$, $\mathcal{R}(\mu_\ell)$, and $\mathcal{R}(\mu_r)$ for the roots μ_ℓ and μ_r of the left and right-trees of μ. For a node $\nu(\mu')$ of $\Pi_X(\nu)$, $x = \ell$ or r, extended node attribute set $\hat{\mathcal{N}}(\nu(\mu'))$ consists of $\mathcal{N}(\nu)$ and $\mathcal{R}(\mu')$.

Lemma 3.2 *Suppose $\mathcal{Z} = (\mathcal{N}, \mathcal{P}, \mathcal{R}, F)$ is a tree attribute system on unbounded degree tree T. Then,*

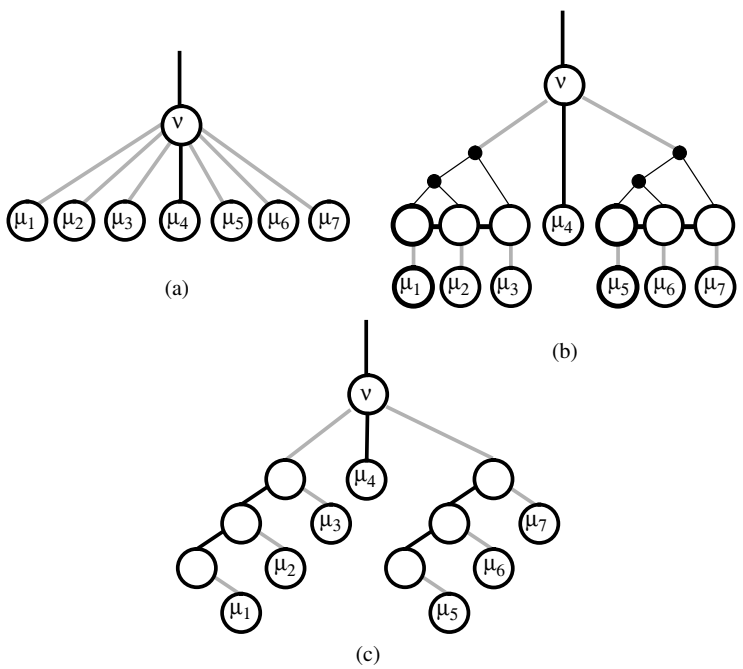

Figure 3.4: A node ν in an unbounded degree tree. (a) The node its solid path, and its children. (b) The left and right edge-paths. (c) An equivalent bounded degree representation.

- *The triplet $(\hat{\mathcal{N}}, \mathcal{P}, F)$ is a path attribute system for paths in T where concatenations are restricted to adjacent subpaths.*
- *The triplet $(\hat{\mathcal{N}}, \mathcal{P}, F)$ is a path attribute system for edge-paths where concatenations are restricted to adjacent subpaths of the edge-paths.*

Proof: Suppose Π is a solid path from node σ to node τ. Consider the path $\hat{\Pi}$ formed by expanding every node μ on Π into two nodes μ' and μ'' on $\hat{\Pi}$ with μ' the parent of μ''. Recall that nodes μ_ℓ and μ_r are the roots of the left and right-trees of node μ with respect to Π. Suppose we know the values $\mathcal{R}(\mu_\ell)$ and $\mathcal{R}(\mu_r)$. We make μ_ℓ the left child of μ' and μ_r the right child of μ''. Path $\hat{(\Pi)}$ is the path from σ'' to τ' in a binary expansion of T. Therefore, the first part of the lemma is true if we define $\mathcal{P}(\Pi)$ to be $\mathcal{P}(\hat{\Pi})$.

The argument for edge-paths is even more direct. Suppose ν is a node and X is either ℓ or r. Each node of edge-path $\Pi_X(\nu)$ has degree bounded by 2. Therefore each edge path is a path in a binary expansion of T, so the lemma holds by definition 3.16. □

In addition to the operations previously defined for bounded degree trees, we support the following operations on unbounded degree trees:

- *Expand*(*node* μ, μ', μ'') — Expand node μ between children μ' and μ''. Node μ' is assumed to precede μ'' in the left-to-right order of the children of μ.
- *Contract*(*node* μ) — Contract μ into its parent node ν. The values $\mathcal{N}(\mu)$ and $\mathcal{N}(\nu)$ are assumed to be equal.

Edge-paths are stored in the same structure as solid paths. If $\Pi_X(\nu)$ is the left or right edge-path of node ν, then reversing $\Pi_X(\nu)$ is equivalent to reflecting the left or right-tree of ν. Therefore, path tree nodes use mirrored eversion (see definition 3.13).

Recall that operation *splice* converts a dashed edge to solid and possibly a solid edge to dashed. Suppose μ is a node connected to its parent ν by a solid edge. Let Π be the solid path containing nodes μ and ν. Let $\Pi_\ell(\nu)$ and $\Pi_r(\nu)$ be the left and right edge-paths of ν. We convert solid edge (μ, ν) to dashed as follows. We begin by splitting path Π at ν. As described in section 3.2.2, operation *split* calls operation push along the path from the root of the path tree for Π to node ν. Operation $push(\nu)$ may require the left and right-trees of ν to be swapped or exchanged (see table 3.2). This is performed by either exchanging the meaning of $\Pi_\ell(\nu)$ and $\Pi_r(\nu)$, or by flipping the *reversed* and *reflected* bits at the root of the path tree representing $\Pi_\ell(\nu)$ and $\Pi_r(\nu)$. We conclude by creating a path tree node $\nu(\mu)$ for the new dashed edge, and setting the left edge-path of ν to be the concatenation $\Pi_\ell(\nu)$, $\nu(\mu)$, and $\Pi_r(\nu)$ and the right edge-path to be the empty path.

Converting a dashed edge to solid is performed similarly, reversing the roles of the solid and edge-paths. Suppose the edge between node μ and its parent ν is now dashed, and there is no solid edge entering ν. Let Π' and Π'' be the paths with $\mu = tail(\Pi')$ and $\nu = head(\Pi'')$. Let $\Pi_\ell(\nu)$ be the left edge-path of ν. Split $\Pi_\ell(\nu)$ at $\nu(\mu)$. The resulting paths to the left and right of $\nu(\mu)$ become the left and right edge-paths of ν. We then concatenate Π' and Π'' to create the solid edge.

For a node μ of a solid path Π we now define the node-attribute $weight(\mu)$ as one plus the sum of the sizes of the left and right edge-paths of μ. For a node $\nu(\mu')$ of an edge-path $\Pi_X(\nu)$ we define the node-attribute $weight(\nu(\mu'))$ as one plus the sum of the sizes of solid path containing μ'. We bias by weight the path trees for both the edge-paths and solid paths. Therefore, the sum of the number of elementary tree operations at each *splice* operation telescopes, so that operations *expose* and *conceal* can be performed with $O(\log n)$ elementary tree operations.

The techniques to perform operations *TreeFind*, *Sibling*, *Reflect* and *Cycle* are slightly modified for unbounded degree trees. All other operations remain unchanged.

We implement operation *TreeFind*(ν, S, q) as in the bounded-degree tree case, with following difference. Let μ be the node we a searching for. Suppose at an intermediate step node μ' is the deepest node on path Π such that the node μ is a descendant of μ'. We proceed as in the bounded case, except we next consider the edge-paths connected to μ', rather than the solid paths containing the children of μ'. Hence, we join an edge-path to Π. This is well defined by the expansion invariant. We perform a similar operation when μ' is an edge-path node.

Operation *Sibling*(μ, k, X) is realized by first calling *expose*(μ). We then use operation *push* to get the value of the *reversed* and *reflected* bits at *Parent*(μ). We find the

siblings by either calling *before* or *after* on the left or right path trees of ν, depending on direction X. We conclude by restoring the size invariant by calling *conceal*(μ).

Operation *Reflect*(μ) is performed as follows. First, we call *expose*(μ). We then flip the *reverse* bit for the left edge-path of μ. Finally, we perform *conceal*(μ) to restore the original solid and dashed edges.

Operation *Cycle*(ν, μ) is performed by first calling *expose*(ν). We then split the left path tree of ν at path tree node $\nu(\mu)$. Let Π' and Π'' be the resulting subpaths representing the children of ν before and after μ. We reorder the children of ν by flipping the *reflected* and *reversed* bits at the root of the path tree representing Π'', concatenating Π'', Π' and $\nu(\mu)$, and making this new edge-path the left edge-path of ν. Finally, we perform *conceal*(ν) to restore the original solid and dashed edges.

Operation *Expand*(μ, μ', μ'') begins by calling *expose*(μ) which results in the left edge-path of μ, $\Pi_\ell(\mu)$ containing all the edges from children to μ. Then we perform a constant number of split and join operations to form three subpaths Π', Π'', and Π''' of $\Pi_\ell(\mu)$, with Π' containing the edge-path nodes associated with the children of μ preceding μ, Π'' containing the edge-path nodes associated with the children of μ between μ' and μ'', and Π''' containing the edge-path nodes associated with the children of μ following μ''. We extend the solid path containing μ by concatenating a new node ν before node μ. We make Π'' the left edge-path of ν, and paths Π' and Π'' the left and right edge-paths of node μ. Finally, we call *conceal*(ν) to restore the size invariant.

Operation *Contract*(μ) is performed similarly. We begin by calling *expose*(μ) which results in the left edge-path of μ, $\Pi_\ell(\mu)$ containing all the edges from children to μ. Let node ν be the parent of μ. We remove node μ. We set $\Pi_\ell(\nu)$ to the concatenation of $\Pi_\ell(\nu)$, $\Pi_\ell(\mu)$, and $\Pi_r(\nu)$. We set $\Pi_r(\nu)$ to the empty path. Finally, we call *conceal*(ν) to restore the size invariant.

Theorem 3.3 *Let $\mathcal{Z} = (\mathcal{N}, \mathcal{P}, \mathcal{R}, F)$ be a tree attribute system, and $\mathcal{S}$ be a collection of path-selection and tree-selection functions. There is a fully dynamic data structure for maintaining $\mathcal{Z}$ and $\mathcal{S}$ over a set of unbounded degree trees with the following performance:*

1. *a tree of size n uses $O(n)$ space;*
2. *operations* MakeTree *and* DeleteTree *take each $O(1)$ time;*
3. *operations* Sibling *takes $O(\log n + k)$ time to return k nodes.*
4. *operations* Evaluate, LocalUpdate, GlobalUpdate, Link, Cut, Evert, Reflect, Cycle, *LCA*, PathFind, TreeFind, Parent, Expand, *and* Contract *take each $O(\log n)$ time, where n is the total size of the trees involved in the operation.*

3.3 Linear Expression Trees

In this section, we consider the dynamic evaluation of linear expressions, which extends and generalizes the dynamic expression trees of [22]. We show how to solve this problem using a tree attribute system.

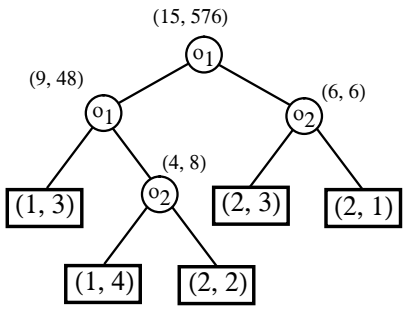

Figure 3.5: A linear expression tree with two binary linear operators. For $x = (x_1, x_2)$ and $y = (y_1, y_2)$ operators are defined to be: $o_1(x, y) = (x_1 + y_2, 2y_1y_2)$ and $o_2(x, y) = (2x_1 + x_1y_2, x_2y_1)$. The unary operators associated with the edges are the identity operators.

Let $\mathcal{S}$ be a semiring with binary operators $\oplus$ and $\otimes$, such that $\otimes$ is distributive with respect to $\oplus$. We assume that elements of $\mathcal{S}$ can be stored in $O(1)$ space and binary operations can be performed in $O(1)$ time. The extension to the general case is straightforward. We consider operations in $\mathcal{S}^r$, the space of r-tuples of $\mathcal{S}$ for some fixed r. Let $x = (x_1, \ldots, x_r)$ and $y = (y_1, \ldots, y_r) \in \mathcal{S}^r$. A *binary linear operator* $\odot : \mathcal{S}^r \times \mathcal{S}^r \to \mathcal{S}^r$ is defined by

$$(x \odot y)_i = x^T A_i y \oplus b_i^T x \oplus c_i^T y \oplus d_i$$

for $i = 1, \cdots, r$, where $A_i \in \mathcal{S}^r \times \mathcal{S}^r$, $b_i, c_i \in \mathcal{S}^r$, and $d_i \in \mathcal{S}$. A *unary linear operator* $\nabla : \mathcal{S}^r \to \mathcal{S}^r$ is defined by

$$\nabla x = Ax \oplus b$$

where $A \in \mathcal{S}^r \times \mathcal{S}^r$, and $b \in \mathcal{S}^r$. (In the above definitions we have used standard linear algebra notation, and convention that xy denotes $x \otimes y$.) Note that, given a binary linear operator $\odot$, for any fixed $\bar{y}$, $x \odot \bar{y}$ is a unary linear operator. Also, given a unary linear operator ∇, there exists a binary linear operator $\odot$ such that $\nabla x = x \odot 0$.

A *linear expression* $\mathcal{E}$ over $\mathcal{S}^r$ is an expression with variables in $\mathcal{S}^r$ involving binary and unary linear operators. A linear expression is represented by a tree T called a linear expression tree, such that the internal nodes are the operators and the leaves are the variables of $\mathcal{E}$ (see an example in Fig. 3.5). We associate a binary linear operator with each internal node of T. If μ is an internal node associated with a binary linear operator, but with only one child, then μ takes the value of its child.

Example 3.6 Fig. 3.5 shows an example of linear expression and its associated binary tree. □

We allow unbounded degree linear expression trees. Suppose $\odot$ is a binary linear operator. The unbounded version of $\odot$ takes a variable and unbounded number of

operands. We write $\odot(x_1, \ldots, x_d)$ to indicate applying $\odot$ to values $x_1, \ldots, x_d$. If $d = 1$ then the value of $\odot(x_1)$ is $x_1 \odot 0$. Otherwise, the value of $\odot(x_1, \ldots, x_d)$ is

$$((x_1 \odot x_2) \ldots) \odot x_d$$

. An *unbounded degree linear expression* is an expression containing unbounded versions of binary linear operators.

An unbounded linear expression is represented by an unbounded linear expression tree. In order to keep the value invariant each binary linear operator must be associative.

We consider a dynamic environment where linear expressions are manipulated by changing the values of the variables and by linking and cutting the corresponding trees. We also allow a somehow unorthodox modification of an expression $\mathcal{E}$ by everting its tree T.

In order to give a meaning to the eversion of T, we require the following. Suppose μ is a node that is not the root of T. Everting at node μ results in μ having an additional child. We do not allow eversion at a leaf of a linear expression tree. In order to give meaning to eversion at a node with 1 child, each internal node calculates a binary linear operator. A unary linear operator is simulated by a binary linear operator with only one child. Finally, we allow the root ρ of T to have three children. If $\odot$ is the binary linear operator at ρ, then the value of ρ is calculated as in the unbounded version of $\odot$. As with general unbounded linear operators, we require that if μ is a node of T with 2 children, then eversion is allowed at μ only if the operator $\odot$ at μ is associative.

Given a linear expression tree T we will show that we can maintain a tree attribute system to find the values of subexpressions of T.

Lemma 3.3 *Suppose $\mathcal{E}$ is a linear expression. The dependence of the value y of $\mathcal{E}$ on a single variable x can be expressed by a unary linear operator $y = Ax \oplus b$.*

Proof: The proof is by induction on the level of the subexpression containing x. If y is x, then the unary linear operator is $y = x$. Now suppose $y = y' \odot y''$ and x is used in the calculation of y'. By the inductive hypothesis, we have $y' = A'x \oplus b'$ for some matrix A' and vector b'. When we fix the value of y'', we get by the earlier observation matrix A'' and vector b'' such that $y = A''y' \oplus b''$. By composition, we have $y = A''A'x \oplus (A''b' \oplus b'')$ □

We call *transfer pair* a (matrix, vector) pair (A, b) that by Lemma 3.3 characterizes the dependency of a linear expression on a variable. In particular, given a (solid or edge) path Π of T, we denote with $P(\Pi)$ the transfer pair associated with the dependency of the value of $tail(\Pi)$ from the value of $head(\Pi)$.

Lemma 3.4 *Let $\mathcal{E}$ be a linear expression represented by a tree T, and let $x(\mu)$ be the value of the subexpression given by the subtree of T rooted at μ. Then the transfer pairs $P(\Pi)$ of the solid or edge paths Π of T form a path attribute system, and there exists a tree attribute system that supports tree attribute $x(\mu)$ by means of the path attribute $P(\Pi)$.*

Proof: Suppose path Π of T is the concatenation of paths Π' and Π'' with transfer pairs (A', b') and (A'', b''), respectively. The value of $head(\Pi'')$ is obtained from the values of the children of $head(\Pi'')$ by means of a linear operator; hence $x(head(\Pi'')) = Cx(tail(\Pi'))\oplus d$. By Lemma 3.3 we have $x(head(\Pi)) = A''(C(A'x(tail(\Pi))\oplus b')\oplus d)\oplus b''$. Thus, $x(head(\Pi)) = A''CA'x(tail(\Pi)) \oplus A''(Cb' \oplus d) \oplus b''$, so that the transfer pair $P(\Pi) = (A, b)$ is given by $A = A''CA'$ and $b = A''(Cb' \oplus d) \oplus b''$. This shows that the transfer pairs form a path attribute system. Clearly, transfer pair $P(\Pi)$ allows to compute in $O(1)$ time $x(tail(\Pi))$ from $x(head(\Pi))$. Also, the transfer pairs depend on an extended node attribute set that for a node μ of a path Π consists of the operator of μ and the value $x(\mu')$ of the child μ' of μ not in Π. Hence, the triplet (O, P, x), where $O(\mu)$ denotes the operator of node μ, is a tree attribute system, and the lemma follows from Theorem 3.2. □

Theorem 3.4 *Let $\mathcal{S}$ be a semiring and r an integer. There is a fully dynamic data structure for maintaining a set of expressions with (unary and binary) linear operators over $\mathcal{S}^r$ with the following performance:*

1. *an expression of size n uses $O(n)$ space;*
2. *operations* MakeTree *and* DeleteTree *take each $O(1)$ time;*
3. *operations* Evaluate, LocalUpdate, Link, Cut, Evert, Reflect, Cycle, PathFind, TreeFind, Parent, *and* Sibling, *take each $O(\log n)$ time, where n is the total size of the trees involved in the operation.*

If we restrict ourselves to associative binary linear operators, then the expansion invariant holds for linear operators. Therefore, by lemma 3.2, lemma 3.4, and Theorem 3.2, we get:

Theorem 3.5 *Let $\mathcal{S}$ be a semiring and r an integer. There is a fully dynamic data structure for maintaining a set of unbounded linear expressions with associative binary linear operators over $\mathcal{S}^r$ with the following performance:*

1. *an expression of size n uses $O(n)$ space;*
2. *operations* MakeTree *and* DeleteTree *take each $O(1)$ time;*
3. *operations* Sibling *takes $O(\log n + k)$ time to return k nodes.*
4. *operations* Evaluate, LocalUpdate, Link, Cut, Evert, Reflect, Cycle, LCA, PathFind, TreeFind, Parent, Expand, *and* Contract *take each $O(\log n)$ time, where n is the total size of the trees involved in the operation.*

3.4 Linear Attribute Grammars

In this section, we consider the dynamic evaluation of linear attribute grammars and show that this problem can also be solved using a tree attribute system.

An *attribute grammar* T is a rooted, ordered tree such that for some fixed r, each node μ of T stores an r-tuple of attributes $x(\mu) = (x_1(\mu), \ldots, x_r(\mu))$. *Synthesized* attributes are calculated from the children of μ. *Inherited* attributes are calculated from the attributes of the parent of μ, and the synthesized attributes of μ and its siblings. We assume, without loss of generality, that the synthesized attributes of μ precede the inherited attributes of μ. A *binary attribute grammar* is an attribute grammar on a binary tree. We consider attribute grammars where the values of each attribute are taken from a semiring $\mathcal{S}$.

Attribute grammars [70] are much studied and have applications in areas such as language based editors [11,92], and VLSI design [68]. Incremental algorithms for attribute grammars have been presented [67,91] which give a $O(a \cdot t)$ running time per operation where a is the number of attributes affected by a change and t is the time to make a single change. In general, $a = O(n)$ for an attribute grammar of size n. Incremental evaluation of general circuits is studied in [2].

The precedence graph for an attribute grammar T is a digraph consisting of a vertex for each attribute $x_i(\mu)$ and a directed edge from $x_i(\mu)$ to $x_j(\nu)$ if the value $x_i(\mu)$ is used in the calculation of $x_j(\nu)$.

For a node μ of attribute grammar T, let $s(\mu) = (s_1(\mu), \ldots s_r(\mu))$ and $h(\mu) = (h_1(\mu), \ldots h_r(\mu))$ be the r-tuples defined by:

$$s(\mu) = \begin{cases} x_j(\mu) & \text{if } x_j(\mu) \text{ is synthesized} \\ 0 & \text{if } x_j(\mu) \text{ is inherited} \end{cases}$$

$$h(\mu) = \begin{cases} x_j(\mu) & \text{if } x_j(\mu) \text{ is inherited} \\ 0 & \text{if } x_j(\mu) \text{ is synthesized} \end{cases}$$

where $1 \leq j \leq r$.

Suppose node μ has $p < r$ synthesized attributes. Then $s^R(\mu) = (x_1(\mu), \ldots, x_p(\mu))$, and $h^R(\mu) = (x_{p+1}(\mu), \ldots, x_r(\mu))$.

Definition 3.18 Consider a node μ of a binary attribute grammar T. Let node ν be the parent of μ, node μ' be the sibling of μ, and nodes μ_1 and μ_2 be the children of μ. A *bidirectional binary operator* $\odot = (\odot_S, \odot_I, \odot_\ell, \odot_r)$ is a 4-tuple of binary linear operators such that:

$$s(\mu) = x(\mu_1) \odot_S x(\mu_2)$$

and, if μ is the left child of ν:

$$h(\mu) = s(\mu) \odot_I (x(\nu) \odot_r s(\mu'))$$

otherwise, if μ is the right child of ν:

$$h(\mu) = s(\mu) \odot_I (x(\nu) \odot_\ell s(\mu'))$$

Definition 3.19 A *binary linear attribute grammar* T is a binary attribute grammar such that:

- The value of all attributes come from a semiring $\mathcal{S}$.
- Each node μ of T has an associated bidirectional operator to calculate the value $x(\mu)$.
- The precedence graph of G is acyclic.
- The dependence of the value an attribute $x_i(\mu)$ on the value of any other attribute $x_j(\nu)$ is linear. That is, we can express the dependence in the form $x_i(\mu) = ax_j(\nu) + b$ where a and b are members of $\mathcal{S}$.

A (binary) linear expression tree is a (binary) linear attribute grammar without inherited attributes.

Suppose G is the precedent graph of a linear attribute grammar T and $x_i(\nu_1)$ and $x_j(\nu_2)$ are two nodes of G. Suppose π' and π'' are two paths in G from $x_i(\nu_1)$ to $x_j(\nu_2)$ that only meet at their endpoints. Let $x_h(\mu')$ and $x_k(\mu'')$ be the predecessors of $x_j(\nu_2)$ on π' and π''. Then the attributes $x_h(\mu')$ and $x_k(\mu'')$ are not multiplied by the operation performed at ν_2.

Suppose $x = (x_1, \ldots, x_h)$ is a vector. The *power vector* of x, $power(x)$, is a vector with 2^h entries $(1, x_1, \ldots, x_h, x_1x_2, \ldots, x_{h-1}x_h, \ldots, x_1 \cdots x_h)$.

Lemma 3.5 *Suppose G is a precedence graph such that all dependencies are linear. Consider attributes $y_0, \ldots, y_f$ of G. The dependence of the value of y_0 on the values of $y_1, \ldots, y_f$ can be expressed as $y_0 = a \cdot \text{power}((y_1, \ldots, y_f))$ for some vector a with 2^f components.*

Proof: Let G be the precedence graph of T. Our proof is by induction on the length of a longest directed path from some y_i $(1 \leq i \leq f)$ to y_0 in G. If the length is 0, then all y_i $(0 \leq i \leq f)$ are identical, and the lemma follows with $a = (0, 1, 0, \ldots)$.

Otherwise, let $(\nu_1, \ldots, \nu_k)$ be the attributes of T such that for each ν_j $(1 \leq j \leq k)$ there is an edge from ν_j to y_0 and a path of length greater than or equal to 0 from some y_i to ν_j. Then for each ν_j there is a vector a_j such that $\nu_j = a_j \cdot power((y_1, \ldots, y_j))$. Additionally, there is a vector b such that $y_0 = b \cdot power((\nu_1, \ldots, \nu_k))$. By substitution, we get a polynomial formula for y_0 in terms of $y_1, \ldots, y_j$. No term of this formula can contain y_j^g, $g \geq 2$, for $1 \leq j \leq k$ since this would violate the linear dependency of attributes in a linear attribute grammar. □

Corollary 3.1 *Suppose μ is a node of linear attribute grammar T. The dependence of a synthesized attribute $x_i(\mu)$ on the inherited attributes $h^R(\mu)$ can be expressed as $x_i(\mu) = a \cdot \text{power}(h^R(\mu))$.*

The vector a depends only on the calculations of attribute values performed at nodes in the subtree rooted at μ, excluding the calculation of the inherited attributes of μ.

Corollary 3.2 *Suppose* Π *is a path in linear attribute grammar* T *with head* σ *and tail* τ. *Let* y *be the vector consisting of the attributes* $s^R(\sigma)$ *followed by the attributes* $h^R(\tau)$. *The dependence of an inherited attribute* $x_i(\sigma)$ *on the attributes of* y *can be expressed as* $x_i(\sigma) = a \cdot \text{power}(y)$ *for some vector* a. *Similarly the dependence of synthesized attribute* $x_j(\tau)$ *on the attributes of* x *can be expressed as* $x_j(\tau) = b \cdot \text{power}(y)$ *for some vector* b.

The vectors a and b depends on the calculations of attribute values performed at nodes in the subtree rooted at τ excluding the subtree rooted at σ.

Suppose we are given a directed acyclic graph G. Graph G is an *evaluation graph* if each vertex v of G calculates a value $value(v)$ by an equation $value(v) = a \cdot power(y)$, where y is the vector of values of vertices u with edges in G from u to v. A *topological sort* of G orders the vertices of G such that if there is an edge from vertex u to vertex v in G, then u precedes v in the ordering. A *simple evaluation scheme* calculates the values for each vertex of G in topological order. In this way, the input values are available when calculating the value of a vertex.

For a node μ in linear attribute grammar T, corollary 3.1 shows that we can summarize the dependence of the attributes of $s^R(\mu)$ on the attributes of $h^R(\mu)$ as an evaluation graph $summarygraph(\mu)$. There is a vertex of $summarygraph(\mu)$ for each attribute of μ. Edges are directed from attributes of $s^R(\mu)$ to attributes of $h^R(\mu)$. Similarly, for a path Π in T with head σ and tail τ, corollary 3.2 shows that we can summarize the dependence of the attributes of $h^R(\sigma)$ and $s^R(\tau)$ on the attributes of $s^R(\sigma)$ and $h^R(\tau)$ as an evaluation graph $summarygraph(\Pi)$. There is a vertex of $summarygraph(\Pi)$ for each attribute of σ and τ. Edges are directed from attributes of $s^R(\sigma)$ and $h^R(\tau)$ to attributes of $h^R(\sigma)$ and $s^R(\tau)$. Notice that for any node μ and any path Π, evaluation graphs $summarygraph(\mu)$ and $summarygraph(\Pi)$ will be of constant size.

Lemma 3.6 *Let* T *be a linear attribute grammar,* $R(\mu)$ *be the graph* summarygraph(μ), $N(\mu)$ *be the bidirectional binary operator* $\odot$, *and* $P(\Pi)$ *be the directed graph* summarygraph(Π). *Then there exists a tree attribute system that supports tree attribute* $R(\mu)$ *by means of the path attribute* $P(\Pi)$.

Proof: Suppose Π is a path in T from node σ to node τ. Evaluation graph $summarygraph(\Pi)$ gives the dependencies of the attributes of $h^R(\sigma)$ and $s^R(\tau)$ on the attributes of $s^R(\sigma)$ and $h^R(\tau)$. Evaluation graph $summarygraph(\sigma)$ gives the dependencies of the attributes of $s^R(\sigma)$ on the attributes of $h^R(\sigma)$. We combine these two graphs and perform the simple evaluation scheme to get the dependencies of the attributes of $s^R(\tau)$ on the attributes of $h^R(\tau)$, which is $summarygraph(\tau)$. All this can be done in constant time, since these graphs are of bounded size.

To show that $P(\Pi)$ can be maintained in a path attribute system, consider path Π' from head σ' to τ' and path Π'' from head σ'' to τ'' with node σ'' the parent of node τ'. Suppose node μ is the sibling of node τ'. The evaluation graph $summarygraph(\mu)$ will be part of the extended node attribute set $\tilde{\mathcal{N}}(\sigma'')$.

Suppose path Π is the concatenation of paths Π' and Π''. To find $summarygraph(\Pi)$, we form an evaluation graph H with vertices for each of the attributes of $x(\sigma')$, $x(\tau')$, $x(\sigma'')$, $x(\tau'')$, and $x(\mu)$. The edges of H are the edges of $summarygraph(\Pi')$, $summarygraph(\Pi'')$, $summarygraph(\mu)$, the dependencies for calculating $s^R(\sigma'')$ from $x(\tau')$ and $x(\mu)$, and the dependencies for calculating $h^R(\tau')$ from $s^R(\tau')$, $s^R(\mu)$, and $x(\sigma'')$. We perform the simple evaluation scheme on H to determine the dependencies of the attributes of $h^R(\sigma')$ and $s^R(\tau'')$ on the attributes of $s^R(\sigma')$ and $h^R(\tau'')$, which is $summarygraph(\Pi)$ (see Fig. 3.6). All this can be done in constant time, since these graphs are of bounded size. □

For a node μ of linear attribute grammar T, the value of $Evaluate(\mu)$ is the evaluation graph $summarygraph(\mu)$. In order to return the actual values of $x(\mu)$, we present the following operation for linear attribute grammars.

- *EvaluateAttributes*(*node* μ) — Returns the values of the attributes $x(\mu)$.

We implement operation $EvaluateAttributes(\mu)$ as follows. First we call $Evaluate(\mu)$ to get evaluation graph $summarygraph(\mu)$. We then call $expose(\mu)$ to get path Π from μ to root ρ. By combining evaluation graphs $summarygraph(\Pi)$ and $summarygraph(\mu)$, and performing the simple evaluation scheme, we get the dependencies of $x(\mu)$ on $h^R(\rho)$. We then calculate the actual values of $x(\mu)$ by plugging in the default inherited value for the root ρ of T. We finish by calling $conceal(\mu)$ to restore the size invariant.

Theorem 3.6 *Let $\mathcal{S}$ be a semiring and r an integer. There is a fully dynamic data structure for maintaining a set of linear attribute grammars over $\mathcal{S}^r$ with the following performance:*

1. *an linear attribute grammar of size n uses $O(n)$ space;*
2. *operations* MakeTree *and* DeleteTree *take each $O(1)$ time;*
3. *operations* EvaluateAttributes, LocalUpdate, Link, Cut, Evert, Reflect, Cycle, LCA, PathFind, TreeFind, Parent, *and* Sibling, *take each $O(\log n)$ time, where n is the total size of the linear attribute grammars involved in the operation.*

A binary linear operator is an *unbounded linear operator* if it keeps the expansion invariant. That is, if μ is a node of linear attribute grammar T, then $summarygraph(\mu)$ is unchanged after expanding any children of μ. Note that the actual value of attributes may change as as a result of an *Expand* operation.

Consider the following example. Given a node μ in linear attribute grammar T, let $depth(\mu)$ be the depth of node μ in T. The attribute $depth(\mu)$ is inherited. The value $depth(\mu)$ is either 0, if μ is the root of T, or $depth(\nu)+1$, where node ν is the parent of

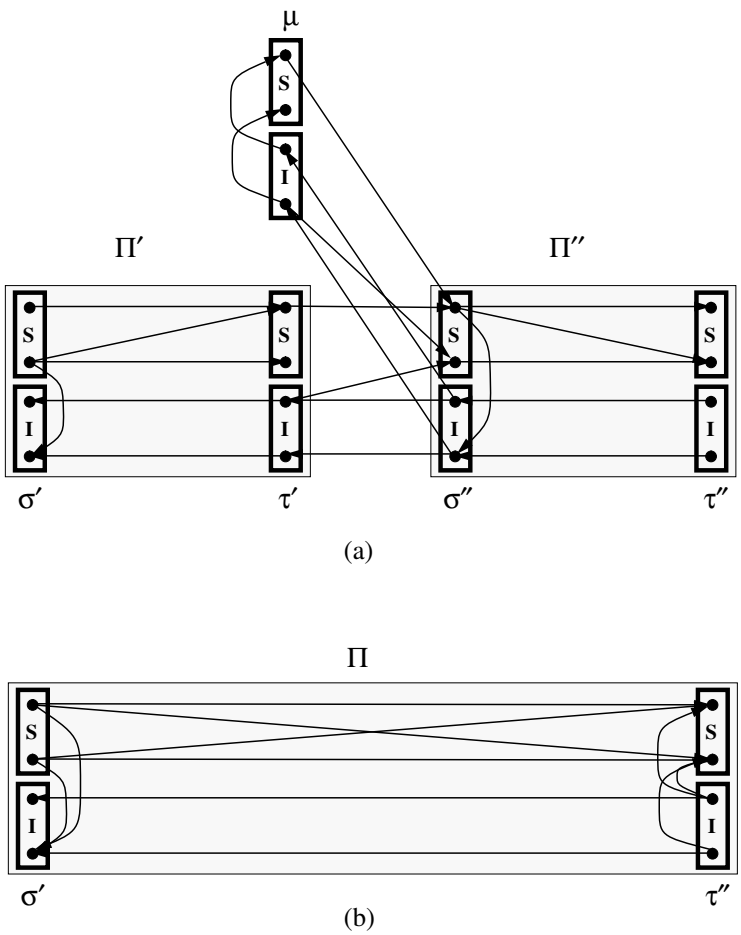

Figure 3.6: An example of the concatenation of summary graphs as described in lemma 3.6. (a) The combined evaluation graph. (b) The resulting summary graph

μ. If μ is an internal node and μ' is the leftmost child of μ. Let $synthesizeddepth(\mu) = depth(\mu') - 1$. Therefore, we get the relation $synthesizeddepth(\mu) = depth(\mu)$, which remains true after any expansion of a node of T. Yet, when expanding children of μ, we increment the depth of the descendants of the expanded nodes.

An *unbounded linear attribute grammar* is a linear attribute grammar with unbounded degree nodes such that each node calculates an unbounded linear operator.

By by lemma 3.2, lemma 3.6, and Theorem 3.2, we get:

Theorem 3.7 *Let $\mathcal{S}$ be a semiring and r an integer. There is a fully dynamic data structure for maintaining a set of unbounded linear attribute grammars over $\mathcal{S}^r$ with the following performance:*

1. *an linear attribute grammar of size n uses $O(n)$ space;*

2. *operations* MakeTree *and* DeleteTree *take each* $O(1)$ *time;*

3. *operations* Sibling *takes* $O(\log n + k)$ *time to return* k *nodes.*

4. *operations* EvaluateAttributes, LocalUpdate, Link, Cut, Evert, Reflect, Cycle, *LCA*, PathFind, TreeFind, Parent, Expand, *and* Contract *take each* $O(\log n)$ *time, where* n *is the total size of the linear attribute grammars involved in the operation.*

3.5 Applications

In this section we introduce applications of tree attribute systems. We show how a number of existing data structures can be expressed as tree attribute systems, and we present new applications using our techniques. Further applications are discussed in chapters 4 and 5.

3.5.1 Tree Structures for Graph Problems

A number of existing data structures can be expressed as tree attribute systems. This list includes dynamic trees and edge ordered trees. In this section, we review these data structures, and existing applications for tree attribute systems, linear expression trees, and linear attribute grammars.

Dynamic trees [100] are unordered, unbounded, rooted trees supporting operations *Link*, *Cut* and *Evert*. Each node μ stores a globally updatable cost associated with the edge from μ to its parent. Global updates are only performed on the path from a node to the root. A specialized *PathFind* operator is included to find the minimum cost edge on a path. Dynamic trees were first presented as an internal data structure in several (sequential) maximum flow algorithms.

A number of data structures have been presented which maintain the recursive decomposition of graphs into subgraphs with constant size overlap. SPQR-trees [28] are used to maintain the tri-connected components of an initially biconnected graph. FWRT-trees [69] used to maintain the 4-connected components of an initially tri-connected graph. These last two data structures have a specialized *PathFind* operation to find the closest ancestor of a given type.

Edge-ordered trees [38] are ordered, unbounded, rooted trees supporting operations *Link*, *Cut*, *Expand*, *Contract*, *Cycle* and *Evert*. Each node μ stores a locally updatable cost associated with the edge from μ to its parent. A specialized *PathFind* operator is included to find the minimum cost edge on a path. Edge-ordered trees were presented as the data structure for a fully dynamic algorithm to maintain the minimum or maximum spanning tree of planar graphs.

3.5.2 Summation Heaps

In this section we present a generalization of heaps called summation heaps. Recall that in the tree implementation of a heap, the structure of the tree is arbitrary (see

example 3.5 page 26). Heap elements are stored at the leaves of the tree. Internal nodes are only used to direct us when selecting the minimum value of a heap.

In summation heaps, the structure of the heap is important. We extend costs to include internal nodes. A summation heap is an expression tree over operators + and min. The cost of a node μ is the value of the subexpression rooted at μ.

We replace operation *FindMin* for heaps, with the following operation.

- *FindMinNode*(*node* ν) — return a minimum cost node μ in the subtree rooted at ν.

To implement operation *FindMinNode*, we calculate two values for each node μ:

- $cost(\mu)$ — the cost of node μ.
- $mincost(\mu)$ — the minimum cost of a node in the subtree rooted at μ.

These values are calculated as linear expressions The value of $cost(\mu)$ is defined to be the result of a linear expression. The value of $mincost(\mu)$ is the minimum of $cost(\mu)$ and *mincost* for the children of μ. Operation $FindMin(\mu)$ is performed by calling operation $TreeFind(\mu, S_4)$, where S_4 is the selection function of example 3.5.

Theorem 3.8 *There exists a fully dynamic $O(n)$ space data structure for maintaining a collection of summation heaps of total size n, such that a query or update operation takes time $O(\log n)$*

3.5.3 Blocking Heaps

In this section we present another generalization of heaps called *blocking heaps* which are motivated by the need to search for a minimum valued attribute in linear expression trees. We use blocking heaps in the implementation of the dynamic algorithms for tree-width two graphs (see section 4.2.5).

Suppose $\mathcal{S}$ is a totally-ordered set. We consider operations in $\mathcal{S}^r$, the space of r-tuples of $\mathcal{S}$ for some fixed r. Let $x = (x_1, \ldots, x_r)$ and $y = (y_1, \ldots, y_r) \in \mathcal{S}^r$. A *binary minimization operator* $\odot : \mathcal{S}^r \times \mathcal{S}^r \to \mathcal{S}^r$ is defined such that each component $(x \odot y)_i$ calculates the minimum (or maximum) of some subset of the components of x and some subset of the components of y. Given a component μ_i at node μ, a *source* of μ_i is a component λ_j at leaf λ such that reducing the value of λ_j, reduces the value of μ_i.

Consider the following operation on a tree associated with an expression of binary minimization operators:

- *FindSources*(*node* μ) — return a source for each of the components of μ.

Definition 3.20 A *blocking heap* is a linear expression tree of binary minimization operators which also supports operation *FindSources*.

Suppose T is a blocking heap. For each path Π in T from node σ to node τ, we keep the following boolean table:

- *depends*(Π, i, j) — **true** if there is a path in the dependency graph of T from σ_i to τ_j.

Operation *FindSources*(μ) is performed by calling operation *TreeFind*(μ, S_5, i), for each $1 \leq i \leq r$, where S_5 is the following tree selection function:

Selection Function S_5 *Suppose* Π *is a path, node* η *the root of the path tree* $\mathcal{B}_\Pi$*, and nodes* η' *and* η'' *the left and right children of* η*. Let* τ' *be* tail(η')*. Use the transfer pairs at* η *and* η'*, the binary minimization operator at* head(η'')*, and the* depends *table for* η'' *to determine if there is a component* τ'_j *at* τ' *such that the value of* τ'_j *is the same as* μ_i*, and* μ_i *depends on* τ_j*. If so, return* η'*, else return* η''*.*

Theorem 3.9 *There exists a fully dynamic* $O(n)$ *space data structure for maintaining a collection of blocking heaps of total size* n*, such that a query or update operation takes time* $O(\log n)$

3.5.4 Point Location in Unbalanced Binary Space Partition Trees

A binary space partition is a binary tree where each node corresponds to a connected region of the space (or plane) and to a geometric object that bipartitions such a region (see Fig. 3.7). E.g. a node corresponds to a trapezoid and to a segment that divides it into two trapezoids (for details see [82]). Each leaf λ of binary space partition T represents a region referred to as *region*(λ). If the discrimination of a point with respect to the dividing object (i.e., figuring out which of the two children regions contains the point) can be done in polylog time (usually O(1) time), then one can do point location in polylog time. Previous approaches try to keep the binary space partition balanced [86,99]. With a selection function one can use unbalanced partitions. A specialized version of this technique was introduced in [15].

We support operations on a collection of binary space partitions:

- *Region*(*node* ρ; *point* p) — Return the region containing point p in the binary space partition rooted at node ρ.
- *MakeBSP*() — Create a new elementary binary space partition with no partitions.
- *DeleteBSP*(*node* λ) — Remove the elementary binary space partition represented by node λ.
- *Divide*(*node* λ, *partition* P) — Use partition P to divide the region represented by leaf λ.
- *Compose*(*node* λ, ρ) — Partition the region represented by leaf λ using the binary space partition with root ρ.
- *Decompose*(*node* μ) — Replace the sub-partition represented by node μ with a single region.

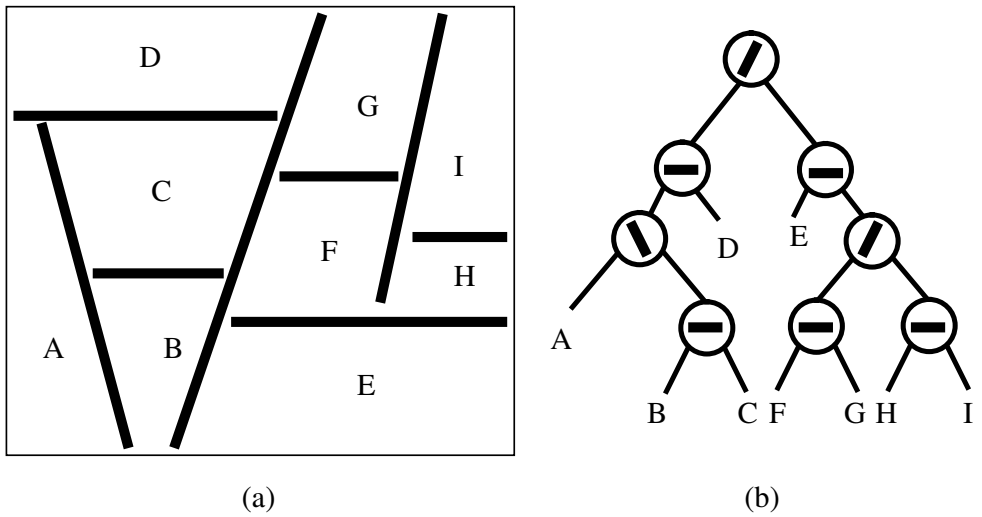

Figure 3.7: (a) A binary space partition of the plane into trapezoids. (b) The associated decomposition tree.

Operations *MakeBSP* and *delbsp* correspond to operations *MakeTree* and *DeleteTree* respectively. Operations *Compose*(λ, ρ) and *Decompose*(μ) each are implemented with a constant number of *Link* and *Cut* operations.

Operation *Divide*(λ, P) consists of replacing node λ with a node μ representing partition P. Node μ has two children representing the regions created by partition P.

We implement operation *Region*(ρ, p) using the following selection function, which takes a point p as an argument.

Selection Function S_6 *Suppose η is the root of a path tree with children η' and η''. If p is contained in* region(tail(η')), *then return η', else return η''.*

Operation *TreeFind*(ρ, S_6, p) returns a leaf λ of T such that point p is contained in *region*(λ) (see Fig. 3.8). We then return either *region*(λ), or if p is on the boundary of *region*(λ), the portion of the boundary containing p.

Theorem 3.10 *There is a fully dynamic data structure for maintaining a collection of unbalanced binary space partitions with the following performance:*

1. *operations* MakeBSP *and* DeleteBSP *take each $O(q_1(n))$ time, where $q_1(n)$ is the time required to build a single partition;*

2. *operations* Compose, Decompose, *and* Divide *take each $O(q_1(n) \cdot \log n)$ time;*

3. *operation* Region *takes $O(q_2(n) \cdot \log n)$ time, where $q_2(n)$ is the time to perform point location in a single region. The value n is the total size of the binary space partitions involved in each operation.*

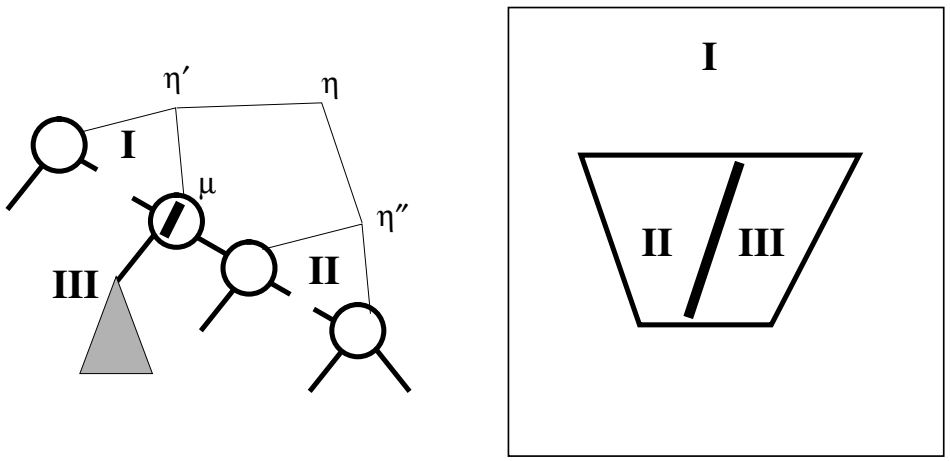

Figure 3.8: (a) A solid path in a decomposition tree; and (b) some regions of a binary space partition. For a point p, consider selection node $\zeta = S_6(\eta, p)$. If $\zeta = \eta''$ then p is in regions I or III. If $\zeta = \eta'$ then p is in regions II. If node $\mu = \mathit{find}(\Pi, S_6, p)$, then p is in region III.

The trapezoid method for planar point location described in [17] is an example of an existing method that can be expressed as a selection function.

3.5.5 Slicing Floorplan Compaction

Linear attribute grammars have immediate application to the problem of compacting *slicing floorplans*, a layout technique widely used in VLSI (see, e.g., [103]). A slicing floorplan is either a rectangle (called basic rectangle), or is the union of two slicing floorplans that share a horizontal side (called horizontal slice) or a vertical side (called vertical slice). Each basic rectangle r has a minimum width w_r and a minimum height h_r, which describe the dimensions of a circuit module to be placed inside r.

An important problem in VLSI layout is determining the location of basic rectangles while minimizing the area of the slicing floorplan, subject to the above constraints on the height and width of the basic rectangles. (For full details, see [103].) This problem can be solved using a linear attribute grammar T. Leaves of T represent basic rectangles, and internal nodes represent horizontal or vertical slices (see Fig. 3.9). The *reference point* of a sub-floorplan is its bottom-left corner.

We keep the following synthesized attributes at each node μ of linear attribute grammar T with root ρ:

- $\mathit{height}(\mu)$ — The height of the sub-floorplan represented by node μ.

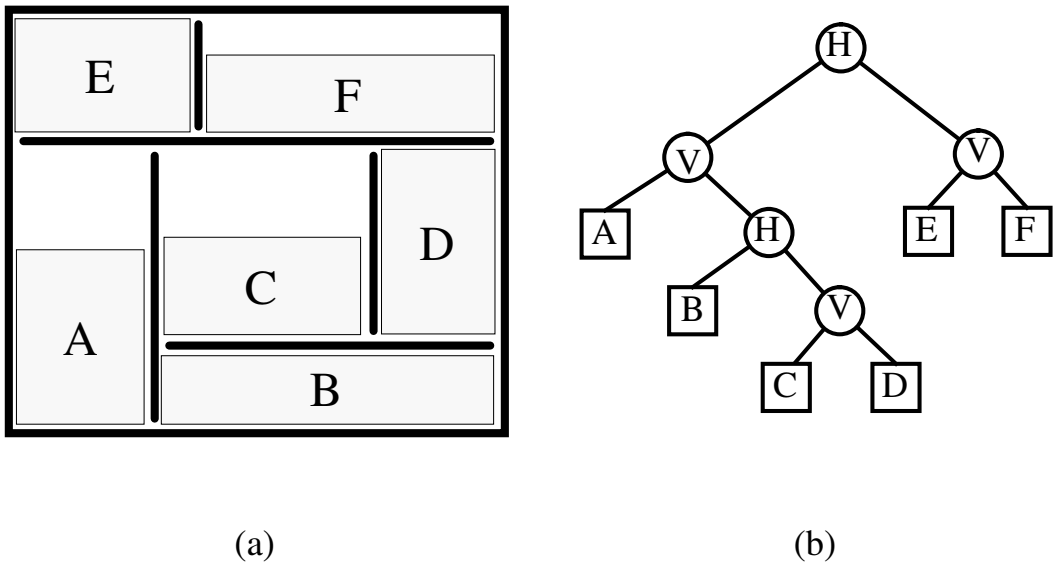

Figure 3.9: (a) A slicing floorplan; and (b) its associated parse tree.

- $width(\mu)$ — The width of the sub-floorplan represented by node μ.

We also keep the following inherited attributes:

- $xpos(\mu)$ — The x-coordinate of the reference point of the sub-floorplan represented by node μ. The value $xpos(\rho)$ is assumed to be 0.
- $ypos(\mu)$ — The y-coordinate of the reference point of the sub-floorplan represented by node μ. The value $ypos(\rho)$ is assumed to be 0.

The equations to calculate these values is shown in table 3.3. Using linear attribute grammars, we can solve a dynamic version of the compaction problem for slicing floorplans, where query operations ask for the minimum area of a floorplan (or sub-floorplan) or the location of a basic rectangle and update operations change the dimensions of a basic rectangle or join (or separate) two floorplans.

Value	*Vertical*	*Horizontal*
$height(\mu)$	$\max(height(\mu_1), height(\mu_2))$	$height(\mu_1) + height(\mu_2)$
$width(\mu)$	$width(\mu_1) + width(\mu_2)$	$\max(width(\mu_1), width(\mu_2))$
$xpos(\mu_1)$	$xpos(\mu)$	$xpos(\mu)$
$ypos(\mu_1)$	$ypos(\mu)$	$ypos(\mu)$
$xpos(\mu_2)$	$xpos(\mu) + width(\mu_1)$	$xpos(\mu)$
$ypos(\mu_2)$	$ypos(\mu)$	$ypos(\mu) + height(\mu_1)$

Table 3.3: The equations to calculate the compaction of floorplans for a node μ and its children μ_1 and μ_2. The choice of equations depend on if μ represents a vertical or horizontal slice.

Chapter 4

Bounded Tree-Width Graphs

4.1 Introduction

In this chapter, we study the dynamic maintenance of graphs of tree-width two. Graphs of tree-width one are forests. Graphs of tree-width two have been extensively studied due to their role in fault-tolerant communication networks [43,44,78,119], concurrent broadcasting in common medium networks [23], reliability evaluation in complex systems [3], and evaluation of queries in relational data base systems [4]. Note that tree-width two graphs include the important subclasses of trees and series-parallel graphs.

The notion of the tree width of a graph was introduced by Robertson and Seymour [94]. Given a graph $G = (V, E)$, a *tree decomposition* of G consists of tree T and a subset of vertices X_μ associated with each node μ of T such that:

- every vertex $v \in V$ is in some X_μ;
- for every edge e in E there is a node μ in T such that both endpoints of e are contained in X_μ; and
- if a vertex v is in both $X_{\mu'}$ and $X_{\mu''}$ for nodes μ' and μ'' in T, then v is contained in X_μ for each node μ on the path in T between μ' and μ''.

The *width* of a tree decomposition is defined to be $max_\mu(|X_\mu| - 1)$. The *tree width* of graph G is the minimum width of any tree decomposition of G. Let $TW(k)$ be the class of graphs of tree width $\leq k$. If $G \in TW(k)$ then clearly any subgraph of G is in $TW(k)$.

Rose [74] gives an alternate characterization of $TW(k)$ graphs as subgraphs of *k-trees*. The class of *k-trees* is defined recursively as follows. The complete graph with k vertices is a k tree. A k-tree with $n + 1$ vertices $n \geq k$ can be constructed from a k-tree with n vertices by adding a new vertex adjacent to all the vertices of one of its k-vertex complete subgraphs. A graph G is a *partial k-tree* if it is the subgraph of a k-tree.

Arnborg and Proskurowski [5] proved that the class of partial k-trees is the same as $TW(k)$. Therefore, we use the terms interchangeably to denote the same class of graphs.

Also, they proved that it is *NP*-Complete to determine the tree-width of a graph. Robertson and Seymour [93] showed the existence of a polynomial time algorithm to determine if a graph has tree width k for some constant k; later Bodlaender [12] and Matoušk and Thomas [76] gave *NC* algorithms to construct a tree decomposition of any partial k-tree (for some constant k), whose width is within a constant factor of k. Recently Lagergren [71] has given a linear processor *NC* algorithm to construct such a tree decomposition. They thus demonstrated that a large number of problems which are *NP*-Complete for general graphs can be solved very efficiently for graphs of constant tree width, both sequentially and in parallel. No equivalent dynamic result has been shown. (Efficient parallel algorithms for problems on series-parallel graphs are given in [59].)

Using tree attribute systems, we find the following results:

- We show that a tree decomposition for a tree-width two graph with m edges can be maintained in a fully dynamic environment so that an update operation takes worst-case time $O(\log^2 m)$.
- We provide a framework for the design of efficient dynamic algorithms on bounded tree-width graphs. Namely, we consider the classes of graph problems DECC and DLCC introduced in [21] which can be represented as tree attribute systems. These classes include the important problems of shortest path, minimum cut, minimum spanning tree, and many problems that are NP-complete for general graphs, such as *vertex cover*, *dominating set*, *chromatic number*, *partial feedback edge set*, *independent set*, *bipartite subgraph*, *max cut*, *partition into Hamiltonian subgraphs*, *partition into cliques*, *transitive subgraph*, *cubic subgraph*, *kernel*, and *chromatic index*. We show fully dynamic data structures supporting these queries for tree-width two graphs. Each query requires $O(\log m)$ time and each update requires $O(\log^2 m)$ time. Our dynamic algorithms use $O(m)$ space.
- We show that the above queries can be maintained in $O(\log m)$ query and update time for series-parallel graphs and trees.
- We show how to maintain the solutions to additional problems on tree-width two graphs including *common neighbor*, *minimum spanning tree*, and *all-connectible pairs shortest path and minimum cut*.

The remainder of the chapter is organized as follows: Section 4.2 describes the fully dynamic algorithm to maintain the tree decomposition of tree-width two graphs. A fully-dynamic algorithm is presented to maintain the solution to problems in DECC and DLCC for tree-width two graphs. While section 4.3 presents fully dynamic algorithms for some additional problems on tree-width two graphs.

4.2 Tree Width Two Graphs

In this section we show how to maintain tree decompositions of tree-width two graphs using tree attribute systems.

Given a graph G, *contracting* edge e of G consists of removing e from G and identifying the endpoints of e. Graph H is a *minor* of G if H can be obtained by a series of contractions from a subgraph of G.

Robertson and Seymour [93] gave a nonconstructive proof of the fact that for any $k \in N$ the class of graphs with tree width at most k, can be characterized by a finite set of forbidden minors. Wald and Colbourn [119] show that tree-width two graphs are exactly the graphs which exclude the complete graph with four vertices, K_4, as a minor.

4.2.1 Preliminaries

Definition 4.1 A *two-terminal graph* $G^\tau = (V, E, \tau)$ consists of a graph $G = (V, E)$ and an ordered pair of distinguished vertices $\tau = (t_1, t_2)$. Vertices t_1 and t_2 are called the *terminals* of G with t_1 being the *first terminal* and t_2 being the *second terminal*. Given ordered pair $\tau = (t_1, t_2)$, let ordered pair $\bar{\tau} = (t_2, t_1)$. The two-terminal graph $G^{\bar{\tau}}$ is called the *swap* of G^τ.

Definition 4.2 The *parallel composition* of two-terminal graphs $G_1^{(s_1,t_1)}, \ldots, G_j^{(s_j,t_j)}$ is a two-terminal graph $G^{(s,t)}$ such that vertex s is the result of identifying vertices $s_1, \ldots, s_j$, and vertex t is the result of identifying vertices $t_1, \ldots, t_j$.

Definition 4.3 The *series composition* of two-terminal graphs $G_1^{(s_1,t_1)}, \ldots, G_j^{(s_j,t_j)}$ is a two-terminal graph $G^{(s,t)}$ such that s is vertex s_1, t is vertex t_1, and for $1 \leq i < j$, we identify terminals t_i and s_{i+1}. The identified vertices form the *join vertices* of the series composition.

Definition 4.4 A *two-terminal series-parallel graph* (TTSP) is recursively defined as follows:

- The graph consisting of a single edge from t_1 to t_2 with terminals (t_1, t_2) is a TTSP.
- Given TTSPs $G_1^{\tau_1}, \ldots, G_j^{\tau_j}$, TTSP G^τ is formed by the parallel composition of $G_1^{\tau_1}, \ldots, G_j^{\tau_j}$.
- Given TTSPs $G_1^{\tau_1}, \ldots, G_j^{\tau_j}$, TTSP G^τ is formed by the series composition of $G_1^{\tau_1}, \ldots, G_j^{\tau_j}$.

Lemma 4.1 *Suppose G^τ is a TTSP. Then $G^{\bar{\tau}}$ is a TTSP.*

Proof: By induction on the compositions to create G^τ. If G^τ is a single edge, then $G^{\bar{\tau}}$ also is a single edge and hence a TTSP. If G^τ is the parallel composition of $G^{\tau'}$ and $G^{\tau''}$ then $G^{\bar{\tau}}$ is the parallel composition of $G^{\overline{\tau'}}$ and $G^{\overline{\tau''}}$. If G^τ is the series composition of $G^{\tau'}$ and $G^{\tau''}$ then $G^{\bar{\tau}}$ is the series composition of $G^{\overline{\tau''}}$ and $G^{\overline{\tau'}}$. □

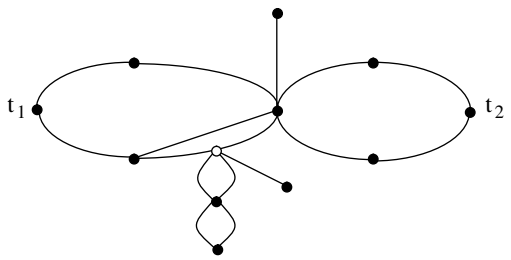

Figure 4.1: A tree-width two graph. Solid-colored vertices are candidates with respect to the first terminal of their train.

Definition 4.5 Graph G is a *series-parallel graph* if G^τ is a TTSP for some terminal pair τ. The TTSP G^τ is said to be a *two terminal representation* or simply representation of G. Graph G is said to be *represented* by G^τ. The two terminal representation is not unique (see lemma 4.4).

A TTSP G^τ is associated with a rooted, ordered, unbounded tree T, called a *SPQ-tree*. A node $\mu \in T$ represents a sub-two-terminal graph of G^τ, called the *pertinent graph* of μ, and denoted by G_μ. Pertinent graph G_μ can also be written G_1^σ, where G_1 is the subgraph of G represented by μ, and σ is the pair of terminals of G_μ. If μ is a leaf, then G_μ is a single edge. Otherwise, G_μ is obtained by the composition (series or parallel) of the pertinent graphs of the children of μ. The nodes of T are of three types: S-nodes, P-nodes, and Q-nodes. Tree T is defined recursively as follows:

- If G^τ is a single edge, then T consists a single *Q-node* μ.
- If G^τ is the parallel composition of two or more TTSPs $G_1^\tau, \ldots, G_d^\tau$ with SPQ-trees $T_1, \ldots, T_d$ with roots $\rho_1, \ldots, \rho_d$, then T consists of a *P-node* root with children $\rho_1, \ldots, \rho_d$.
- If G^τ is the series composition of two or more TTSPs $G_1^\tau, \ldots, G_d^\tau$ with SPQ-trees $T_1, \ldots, T_d$ with roots $\rho_1, \ldots, \rho_d$, then T consists of an *S-node* root with children $\rho_1, \ldots, \rho_d$.

The *type* of a node μ is either P, S, or Q. We keep the *type invariant* that each node of a SPQ-tree T not have a child of the same type. The *terminals* of node μ are the terminals of G_μ. If G^τ has m edges, then T has $O(m)$ nodes. Tree T can be constructed in $O(m)$ time using the recognition algorithm of [118].

The following corollary to lemma 4.1 shows the connection between swapping a TTSP and reflecting the associated SPQ-tree.

Corollary 4.1 *Consider a TTSP G^τ and associated SPQ-tree T. Let $\bar{T}$ be the reflection of T. Then $\bar{T}$ is the SPQ-tree associated with $G^{\bar{\tau}}$.*

Definition 4.6 Let ν be an S-node with children $\mu_1, \ldots, \mu_d$. The *first representative* of ν is node μ_1 and the *last representative* of ν is node μ_d. The *skeleton* of μ, denoted *skeleton*(μ), is the two terminal graph consisting of d edges $e_i = (v_{i-1}, v_i)$, $1 \leq i \leq d$, where v_0 and v_d are the first and second terminals of the pertinent graph of ν, and for $1 \leq i < d$, v_i is the second terminal of the pertinent graph of μ_i. The *proper node* of vertices $v_1, \ldots, v_{d-1}$ is node ν.

Note that v_i is a join-vertex used in the series composition at its proper node. Hence, if G^τ is a TTSP with associated SPQ-tree T then each vertex of G^τ, with the exceptions of its terminals, gets assigned to a proper node.

We allow a node μ to swap its pertinent graph. The *swap status* of a node μ is the exclusive-or of all swaps done by ancestors of μ.

4.2.2 Biconnected Series-Parallel Graphs

Biconnected series-parallel graphs are exactly the class of biconnected graphs that exclude the complete graph with 4 vertices, K_4, as a minor [34]. Hence, biconnected tree-width two graphs are biconnected series-parallel graphs.

Lemma 4.2 [34] *Suppose G is a tree-width two graph and B is a biconnected component of G. Then B is a biconnected series-parallel graph.*

We call the biconnected components of G *blocks*. If T is a SPQ-tree representing a biconnected series-parallel graph, then the root of T is a P-node.

Lemma 4.3 *Let G be a biconnected series-parallel graph and G^τ be a representation of G with associated SPQ-tree T. Let $\tau = (t_1, t_2)$ and let v' and v'' be two vertices of G. Then the graph $\hat{G}$ obtained by inserting an edge from v' to v'' is a biconnected series-parallel graph if and only if one of the following conditions is true:*

1. *$v' = t_1$ and $v'' = t_2$ or $v' = t_2$ and $v'' = t_1$, or*
2. *v' and v'' are vertices of the skeleton of the same S-node of T, or*
3. *v' is a non-terminal vertex of the skeleton of S-node μ_1 and v'' is a non-terminal vertex of the skeleton of S-node μ_2 and nodes μ_1 and μ_2 are the only children of the root P-node of T.*

Proof:

(If) If vertices v' and v'' meet conditions 1 or 2 of the lemma, then $\hat{G}^\tau$ clearly is a TTSP with an additional parallel composition. If vertices v' and v'' meet condition 3 of the lemma, then we can find another representation $G^{\tau'}$ of G, where $\tau' = (v', v'')$. We find $G^{\tau'}$ as follows (see Fig. 4.2).

Consider four subgraphs of G: subgraph G_1 is the subgraph separated by vertices t_1 and v'; subgraph G_2 is the subgraph separated by vertices v' and t_2; subgraph G_3 is the subgraph separated by vertices t_1 and v''; and subgraph G_4 is the subgraph

separated by vertices v'' and t_2. Clearly, each of G_1, G_2, G_3, and G_4 are series-parallel graphs. TTSP G^τ is the parallel compositions of the TTSPs formed by the serial composition of TTSPs $G_1^{(t_1,v')}$ and $G_2^{(v',t_2)}$, and the serial composition of TTSPs $G_3^{(t_1,v'')}$ and $G_4^{(v'',t_2)}$. TTSP $G^{\tau'}$ then is the parallel compositions of the TTSPs formed by the serial composition of TTSPs $G_1^{(v',t_1)}$ and $G_3^{(t_1,v'')}$, and the serial composition of TTSPs $G_2^{(v',t_2)}$ and $G_4^{(t_2,v'')}$.

(Only If) We use the property that a series-parallel graph does not have graph K_4 as a minor. Suppose that v' and v'' do not meet any of the conditions of the lemma. Let nodes μ' and μ'' be $Proper(v')$ and $Proper(v'')$ and let ν be the least common ancestor of μ' and μ'' in T. Suppose ν is a P-node. Let vertices t' and t'' be the terminals of G_ν. Since G is biconnected and condition 3 is not met, there is a path in G from t' to t'' that does not contain an edge of G_ν. Since node ν is a P-node, there will be non-intersecting paths in G_ν between t' and v', v' and t'', t' and v'', and v'' and t''. Therefore, in $\hat{G}$ the vertices v', v'', t', and t'' induce a minor of K_4.

Now suppose ν is an S-node. Since vertices v' and v'' are not vertices of the skeleton of any S-node, at least one of v' and v'' (say v') is not a vertex of $skeleton(\nu)$. Let node μ be the child of ν such that v' is contained in G_μ. By the type invariant, node μ must be a P-node. Let vertices s' and s'' be the terminals of G_μ. Since G is biconnected, there are non-intersecting paths in G between v'' and s', and v'' and s'' that do not contain an edge of G_μ. Also, since μ is a P-node, G_μ represents a biconnected graph. Hence, there are non-intersecting paths in G_μ between s' and v', and v' and s''. Therefore, in $\hat{G}$ the vertices v', v'', s', and s'' induce K_4 as a minor. □

Definition 4.7 Let G be a biconnected series-parallel graph. The vertex pair (v', v'') of G is *2spg-connectible* if the graph resulting from adding an edge between v' and v'' is a biconnected series-parallel graph. If vertices v' and v'' meet condition 3 of lemma 4.3, then we call the pair (v', v'') a *cross-pair*.

Lemma 4.4 *Suppose G is a biconnected series-parallel graph and $\tau = (u, v)$ an ordered pair of vertices. The two-terminal graph G^τ is a TTSP iff (u, v) is 2spg-connectible.*

Proof:

(If) If G^τ is a TTSP then the pair (u, v) is 2spg-connectible by lemma 4.3.

(Only If) Assume that G^σ is some representation of G and T is the associated SPQ-tree. If u and v are the terminals of G^σ, then we are done. If (u, v) is a cross-pair, then by the proof of lemma 4.3, we can find a representation G^τ of G. The final case is when u and v are both vertices of the skeleton of the same S-node μ of T. We show how we can construct an SPQ-tree T' for either G^τ or $G^{\bar{\tau}}$.

Consider the following operation:

- *MakeTerminals*(*node* ρ, *vertex* u, v) — Suppose ρ is the root of SPQ-tree T associated with TTSP G^σ. Restructure T such that the resulting tree T' is associated with G^τ, where τ is the vertex pair (u, v). Return the root of tree T'. Vertices u and v are assumed to either be the terminals of G^σ or vertices of the skeleton of the same S-node ν of T with vertex u preceding vertex v in $skeleton(\nu)$.

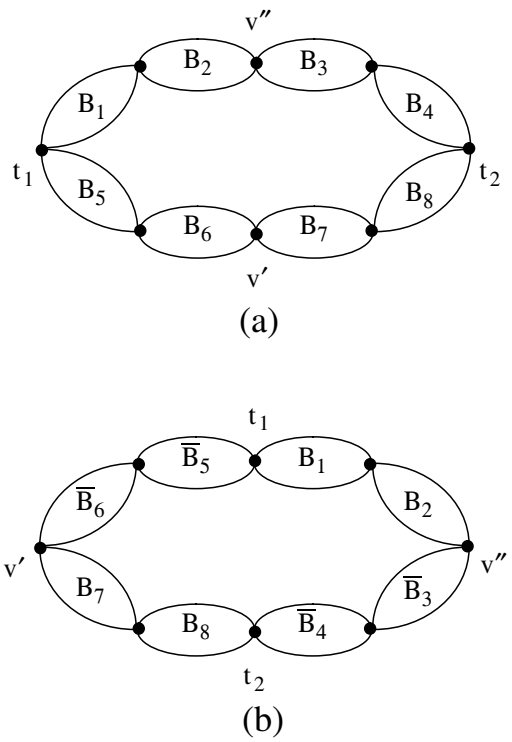

Figure 4.2: Restructuring to make a cross pair the terminals of a biconnected series-parallel graph. (a) The original TTSP. (b) The restructured graph.

We implement operation *MakeTerminals*(ρ, u, v) as follows. If vertices u and v are the terminals of G^σ, then set node μ to be node ρ. Otherwise, if it exists, let node μ be the P-node child of ν such that u and v are the terminals of G_μ. If no such node exists, then we construct it. If ν has a Q-node child λ representing an edge with endpoints u and v, then we insert P-node μ such that ν is the parent of μ and λ is the only child of μ. Otherwise, let μ' be the child of ν such that vertex u is the first terminal of $G_{\mu'}$. Similarly, Let μ'' be the child of ν such that vertex v is the second terminal of $G_{\mu''}$. We expand ν between μ' and μ'' to get node ν'. We then insert P-node μ such that ν is the parent of μ and ν' is the only child of μ.

Let Π be the path from μ to ρ. We perform operation *MakeTerminals* by everting at node μ. We use simple eversion at the P-nodes of Π and steady eversion at the S-nodes of Π (see definition 3.13, page 20). If node ρ now has a single child ρ', we cut ρ from its parent and replace it with ρ'. Finally, if node ρ' is an S-node, then we restore the type invariant by contracting ρ' into its parent.

We prove the correctness of operation *MakeTerminals* by induction on the number of S-nodes on path Π. If Π has no S-nodes, then no restructuring is done and $G^\tau = G^\sigma$. Otherwise, let ν'' be the P-node parent of ν and let s' and s'' be the terminals of $G_{\nu''}$. By induction, *MakeTerminals*(ρ, s', s'') gives us an SPQ-tree T'' associated with $G^{(s,t)}$.

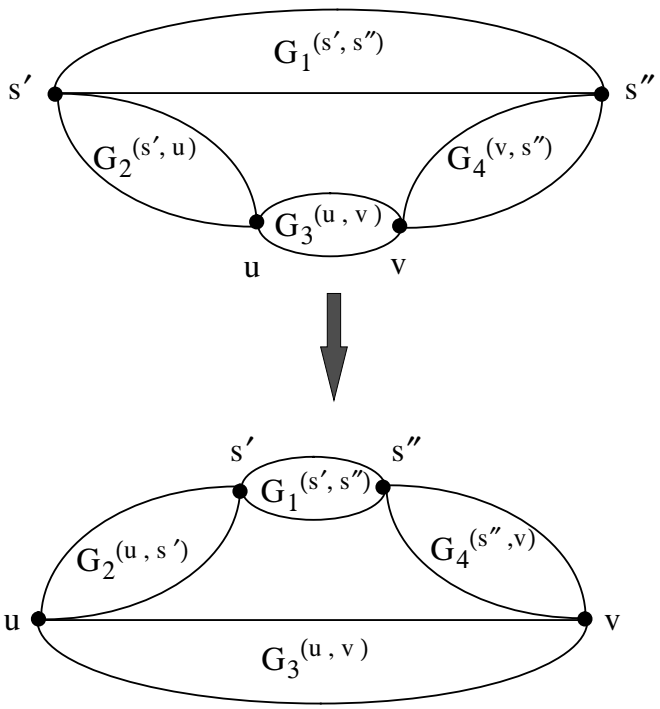

Figure 4.3: Example of the final step of the induction in the proof of lemma 4.4.

Let $G_1^{(s',s'')}$ be the TTSP $G^{(s',s'')} - G_\nu$. We get the associated SPQ-tree by cutting node ν from ν''. Notice that $G^{(s',s'')}$ is the parallel composition of G_ν and $G_1^{(s',s'')}$. Let $G_2^{(s',u)}$ be the TTSP represented by the children of ν to the left of μ, $G_3^{(u,v)}$ be the TTSP represented by μ, and $G_4^{(v,s'')}$ be the TTSP represented by the children of ν to the right of μ. TTSP $G^{(u,v)}$ is then the parallel composition of $G_3^{(u,v)}$ and the series composition of $G_2^{(u,s')}$, $G_1^{(s',s'')}$, and $G_4^{(s'',v)}$ (see Fig. 4.3). This is the graph represented by $\mathit{MakeTerminals}(\rho, u, v)$.

□

4.2.3 Representing Tree Width Two Graphs

Let G be a tree-width two graph. We define the *block-cutvertex tree* $\mathcal{C}$ of G as follows. There is a node in $\mathcal{C}$ for each vertex and block of G. There is an edge between each vertex v and each block which contains v. We arbitrarily root $\mathcal{C}$ at some block B_0. For each block B of G we assign $\mathit{level}(B)$ to be one half the depth of B in $\mathcal{C}$. (Note we do not actually build $\mathcal{C}$, but we implicitly use the level assignments.)

Since C is a tree, any block B of level $i > 0$ in tree-width two graph G shares a cutvertex c with exactly one block B' at level $i - 1$. If B also shares a distinct cut

vertex $c' \neq c$ with any other block B'' then $level(B'') = i + 1$. In this way we can view a tree-width two graph G, as a tree of biconnected series-parallel graphs (see Fig. 4.4).

The following definitions are natural extensions from trees.

Definition 4.8 The *parent* of block B at level i is the unique block B' at level $i - 1$ that shares a vertex with B. Block B is said to a *child* of block B' if B' is the parent of B. Block B is a *leaf* if it has no children. Block B is the *root* if it has no parent. An *ancestor* of block B is either B itself or a parent of an ancestor of B. A *path* is the sequence of blocks encountered from a block B to an ancestor block B'. The *least common ancestor* of blocks B' and B'' is the lowest level block that is an ancestor of both B' and B''. Block B is a *descendant* of block B'' if B'' is an ancestor of B. The root of G will be the block B_0.

Suppose G is a tree-width two graph containing vertices u and v. The *component chain* between u and v is the unique sequence $(B_1, c_1, B_2, \ldots, c_p, B_p)$ of cut vertices and blocks encountered in $\mathcal{C}$ between u and v. Notice that vertex u a member of B_0 and vertex v a member of B_p.

Definition 4.9 The vertex pair (u, v) is *tw2-connectible* if the graph resulting from adding an edge between u and v is a tree-width two graph.

Lemma 4.5 *Let G be a tree-width two graph with vertices $c_0, c_p \in G$, and $(B_1, c_1, B_2, \ldots, c_{p-1}, B_p)$ be a component chain with $c_0 \in B_1$ and $c_p \in B_p$. Then the pair (c_0, c_p) is tw2-connectible iff for each $0 \leq i < p$, (c_i, c_{i+1}) is 2spg-connectible.*

Proof:

(If) Suppose adding an edge e between c_0 and c_{p+1} creates K_4 as a minor, while for each $0 \leq i < p$, (c_i, c_{i+1}) is 2spg-connectible. Therefore, all the vertices of the K_4 minor cannot be members of the same block. Let u and v be vertices of a K_4 minor such that there is no block containing both u and v. In graph G there is only one disjoint path between u and v, since there is a cut vertex between them. Therefore, adding edge e creates at most a second disjoint path from u to v, contradicting u and v as vertices of a K_4 minor.

(Only If) If the pair (c_i, c_{i+1}) is not 2spg-connectible then adding an edge between u and v creates a path from c_i to c_{i+1} that does not share any edge with B_i. Hence a K_4 minor is created. □

Consider a tree-width two graph G containing a block B at level i. Recall that if B is not the root block, then there is exactly one cut vertex v in B that is shared with the parent block of B. We require that any representation of B has v as its first terminal. We allow any representation of the root block. We call vertex v the *linking vertex* of B.

We decompose G into a collection of series-parallel graphs, called trains such that each block of G is contained in exactly one train. Suppose B' is a block at level i with linking vertex v' and B'' is a child block of B' at level $i + 1$ with linking vertex v''. The connection between B' and B'' is said to be *coupled* if B' and B'' are contained in the same train. The connection is *open* otherwise. By lemma 4.4, we can couple

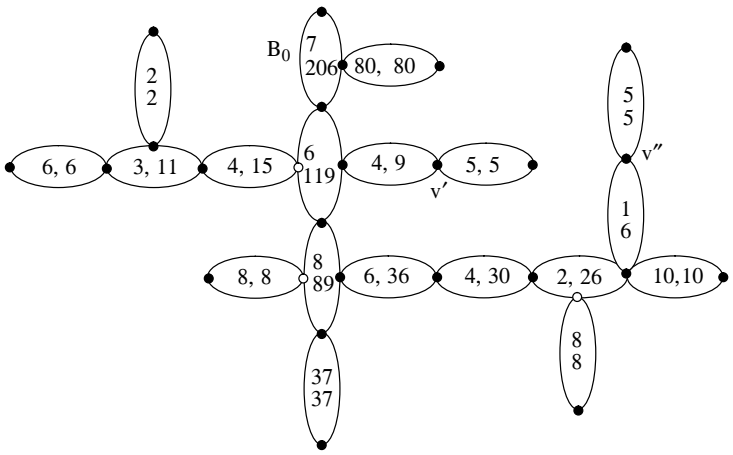

Figure 4.4: Representation of a tree-width two graph that meets the weight invariant starting at block B_0. Each block B is labeled with the number of vertices in B and $weight(B)$. Black filled vertices are candidates and white filled vertices are non-candidates.

the connection between B' and B'' only if the pair (v', v'') is 2spg-connectible in the series-parallel subgraph consisting of block B'.

We now show in a manner similar to dynamic trees that for a graph with m edges we only have to consider $O(\log m)$ trains to perform any dynamic operation.

Suppose G^τ is a TTSP with t_1 its first terminal and v some other vertex. We say that v is a *candidate* with respect to t_1 if (t_1, v) is tw2-connectible.

Let B be a block of a tree-width two graph G. A *candidate child* of B is a block of level $i+1$ that connects to B at a candidate vertex. For each block B, we define the weight of B, $weight(B)$, to be the sum of the number of edges in B and all descendant blocks of B.

We define the connection between child block B' and its parent B to be *heavy* if $weight(B')$ is greater than one-half $weight(B)$, otherwise the connection is *light*. We keep a *weight invariant* that the connection between a candidate child and its parent is coupled if and only if it is heavy. Notice that for any block there is at most one heavy connection to a candidate child. Suppose v is the connecting vertex between B and candidate child B'. We couple the connection between blocks B and B' by representing v as the second terminal of B and representing the connection between B and B' as a series composition. The final result is a decomposition of G into trains. If M is a train of G, we define $level(M)$ to be $level(B)$ where B is the block of M of minimum level. As with blocks, we use the tree relationships parent, child, ancestor, descendant, root, and leaf for trains.

We now define an extension of SPQ-trees, called *SPQC-trees* that will be used to represent tree-width two graphs. Suppose G is a tree-width two graph. For each vertex v of G we associate a unique C-node κ. Suppose M is the lowest level train containing

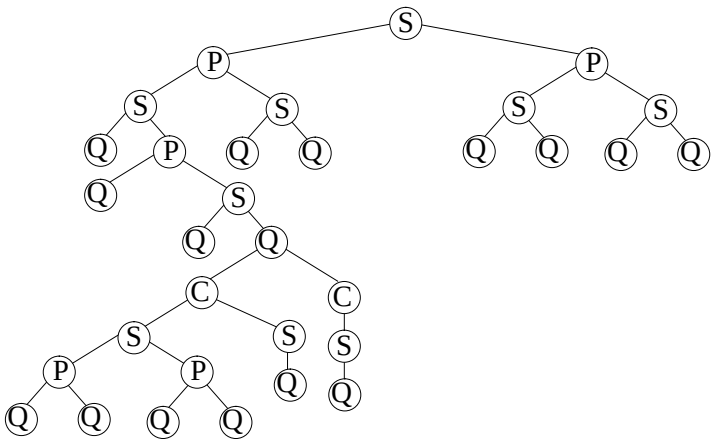

Figure 4.5: The SPQC-tree for the tree-width two graph in Fig. 4.1. Not shown are C-nodes for non-cut vertices.

v and $M_1, \ldots, M_h$ are the higher level trains containing v. Suppose nodes $\rho_1, \ldots, \rho_h$ are the roots of the SPQ-trees representing $M_1, \ldots, M_h$. We require that each ρ_i be an S-node. Therefore, if node ρ_i represents a train consisting of a single block, then ρ_i will have a single child. C-node κ has children $\rho_1, \ldots, \rho_h$. Node κ has as its parent any Q-node μ that represents an edge of M with vertex v as an endpoint. We say that κ is *Attachment*(v). It is possible for a Q-node μ to have 0, 1, or 2 children (see Fig. 4.5).

The SPQC-tree T associated with tree-width two graph G implicitly represents a tree decomposition. We can create a tree decomposition T' from T recursively as follows. We initialize by creating a root ρ' of T' and setting μ to be ρ, the root of T:

- If μ is a C-node with children $\mu_1, \ldots, \mu_d$, create children $\mu'_1, \ldots, \mu'_d$ of μ' in T'. Store the vertex v associated with μ at μ'.
- If μ is a P or Q-node with children $\mu_1, \ldots, \mu_d$, create children $\mu'_1, \ldots, \mu'_d$ of μ' in T'. Store the terminals of M_μ at μ'.
- If μ is an S-node with children $\mu_1, \ldots, \mu_d$, replace μ' in T' with a node for each node in some binary expansion of μ. Store at each node λ in the expansion of μ the union of the terminals of the pertinent trains of the children of λ (see Fig 4.6).

Definition 4.10 Given a node μ in T, the *pertinent train* of μ, M_μ, will be the subgraph induced by the edges associated with Q-nodes which are descendants of μ and can be reached from μ without passing through C-nodes. The *pertinent graph* of μ, G_μ, is the subgraph represented by the sub-tree rooted at μ. The terminals of μ are the terminals of M_μ.

Recall node μ is fixed if it does not allow the subtree rooted at μ to be everted or reflected. In a SPQC-tree T, swapping a node μ represents swapping the pertinent train of μ. Therefore, we fix C-nodes.

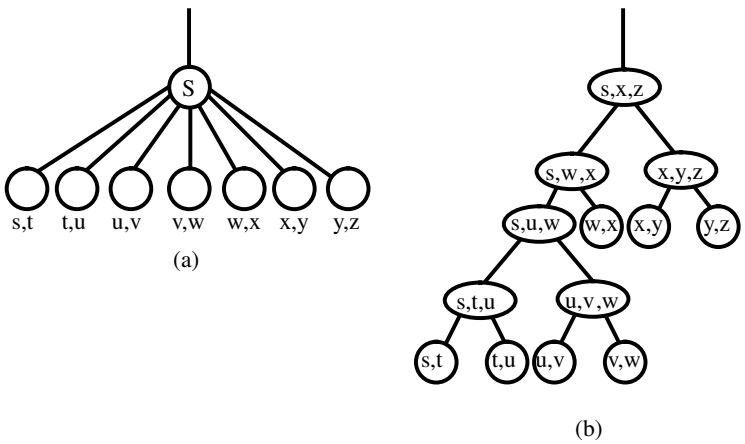

Figure 4.6: A tree decomposition associated with an S-node. (a) The S-node. Each child is labeled with terminals of its pertinent train. (b) The tree decomposition.

Lemma 4.6 *Suppose G is a tree-width two graph and T is a SPQC-tree that represents a decomposition of G that meets the weight invariant, and node μ is a descendant of node ν in T. Let Π be the path from μ to ν in T. Then there are either $O(\log m)$ C-nodes encountered on Π or there is a non-candidate node on Π.*

Proof: If there is a non-candidate node on Π, then we are done. Otherwise, suppose κ is a C-node on Π and node μ' is the child of κ on Π. Let ν' be the node representing the parent train of $M_{\mu'}$. Suppose node ν' is on path Π. There is no C-node on Π between the parent of κ and ν. Therefore, by the weight invariant

$$2 \cdot \mathit{weight}(\mu) < \mathit{weight}(\nu)$$

and the lemma follows. □

4.2.4 Dynamic Operations

Each SPQC-tree is stored as a tree attribute system. We show how to maintain a collection $\mathcal{G}$ of connected tree-width two graphs under a series of dynamic operations. The operations we support are:

- *Report*(*node* μ) — Report the values stored for the pertinent graph of node μ.
- *Update*(*node* μ; *value* x) — Update the value stored at node μ to x.
- *MakeGraph* — Create a new elementary tree-width two graph G, represented by a single Q-node, and add G to $\mathcal{G}$.
- *DeleteGraph*(*node* λ) — Remove from $\mathcal{G}$ the elementary tree-width two graph represented by the single Q-node λ.

- *Attach*(*node* ρ, *vertex* v) — Attach the tree-width two graph represented by node ρ to vertex v by identifying v with the first terminal of the root train of G. Node ρ must be the root P-node of the SPQC-tree representing G.

- *Detach*(*node* μ) — Detach the block represented by node μ, and all its descendant blocks, from G by separating the first terminal of M_μ. The biconnected series-parallel graph G_μ is added to $\mathcal{G}$.

- *Compose*(*nodetype* X; *node* ρ', ρ'') — Perform a composition on the root trains of the tree-width two graphs $G_{\rho'}$ and $G_{\rho''}$. The composition is series or parallel according to whether $X = S$ or $X = P$. The resulting tree-width two graph is added to $\mathcal{G}$ while $G_{\rho'}$ and $G_{\rho''}$ are removed from $\mathcal{G}$.

- *Inject*(*node* ρ, λ) — Replace the edge e represented by Q-node λ with the tree-width two graph G_ρ. The terminals of M_ρ are identified with the endpoints of e. Node ρ must be the root of its SPQC-tree. The resulting TTSP is added to $\mathcal{G}$ while G is removed from $\mathcal{G}$.

- *Extract*(*node* μ) — Let graph G' be the tree-width two graph consisting of G_μ minus the attachments at the terminals of M_μ. Remove G' from G and replace it with a single edge e. The endpoints of e are the terminals of M_μ. The tree-width two graph G' is added to $\mathcal{G}$.

- *InsertEdge*(*vertex* v', v''; *edge* e) — Insert a new edge e from v' to v'' . The operation is performed only if the resulting graph is a tree-width two graph.

- *DeleteEdge*(*edge* e) — Delete edge e.

- *InsertVertex*(*vertex* v; *edge* e, e', e'') — Replace edge e with two edges e' and e'' by inserting vertex v.

- *DeleteVertex*(*node* ρ, *vertex* v; *edge* e, e', e'') — Replace vertex v and its incident edges e' and e'' with a single edge e. The operation is performed only if e' and e'' are the only incident edges of v.

- *Swap*(*node* ρ) — Swaps the pertinent train of node ρ. Node ρ is assumed to be the root of its SPQC-tree.

In order to implement these operations we assume the existence of the following operations, which are described in section 4.2.5:

- *GetBlock*(*node* μ) — Returns the node ν that represents the block containing M_μ as a subgraph. If μ is a C-node representing vertex v, then return the lowest level block containing v. Operation *GetBlock* runs in $O(\log m)$ time.

- *LCABlock*(*node* μ', μ'') — Returns the node μ that represents the least common ancestor block of the blocks represented by μ' and μ''. Operation *LCABlock* runs in $O(\log m)$ time.

- *GetTrain*(*node* μ) — Returns the node ν that represents the train containing M_μ as a subgraph. If μ is a C-node representing vertex v, then return the lowest level train containing v. Operation *GetTrain* runs in $O(\log m)$ time.

- *Exposegraph*(*node* μ) — For the block B represented by node μ, open the coupled attachment to a higher level block, if it exists. Let B' be the closest ancestor block of B that is not a candidate child of its parent. If no such block exists, then let B' be the root block. Couple all open connections on the path between B and B'. Operation *Exposegraph* returns a node which represents the constructed train and runs in $O(\log^2 m)$ time.

- *ConcealGraph*(*node* ρ) — Restores the weight invariant for the train represented by node ρ. Operation *ConcealGraph* runs in $O(\log^2 m)$ time.

- *MakeTerminal*(*node* μ; *vertex* v) — Restructure G_μ such that vertex v becomes the second terminal. This operation assumes that μ represents a block containing v and that vertex v is 2spg-connectible with the first terminal of M_μ. Operation *MakeTerminal* runs in $O(\log m)$ time.

- *Proper*(*vertex* v) — Returns the node triple (ν, μ', μ''), where ν is the proper node of v, node μ' is the child of ν such that v is the first terminal of μ', and μ'' is the child of ν such that v is the second terminal of μ''. If v is a terminal of the root chain, then *Proper*(v) returns $(\rho, -, -)$, where ρ is the root of T. Operation *Proper* runs in $O(\log m)$ time.

- *GetTerminals*(*node* μ) — Returns the terminal set (t_1, t_2) of the pertinent train of μ. Operation *GetTerminals* runs in $O(\log m)$ time.

- *FindEdge*(*node* μ, *vertex* v) — Returns the Q-node associated with an edge in the pertinent train of μ connected to vertex v. Vertex v must be a terminal of G_μ. Operation *FindEdge* runs in $O(\log m)$ time.

- *Candidate*?(*vertex* v) — Returns true if v is a candidate vertex of *GetTrain*(*Attachment*(v)). Operation *Candidate*? runs in $O(\log m)$ time.

Operations *Report* and *Update* correspond to *Evaluate* and *LocalUpdate* respectively, and hence take $O(\log m)$ time. Operations *MakeGraph* and *DeleteGraph* can be trivially implemented in $O(1)$ time.

Recall that we fix C-nodes. By corollary 4.1, operation *Swap*(ρ) is implemented in $O(\log m)$ time by issuing *Reflect*(ρ).

Operation *Attach*(ρ, v) is implemented as follows. Let vertex v' be the first terminal of M_ρ and node μ' be *Attachment*(v'). Additionally, let node μ be *Attachment*(v). Operation *Attach*(ρ, v) identifies vertices v and v'. Therefore, we merge nodes μ and μ' into a single C-node. We do this by first letting node ν be *Exposegraph*(*GetBlock*(μ)). We then restructure to make both node ρ and the children of node μ' children of node μ. This can be implemented using a constant number of *Cut*, *Link*, and *Contract*

operations. Note that cutting node μ' does not modify the weight invariant for any connection in G_ρ since vertex v' is the first terminal. We conclude by restoring the weight invariant by calling operation *ConcealGraph*(ν). All this can be implemented in $O(\log^2 m)$ time.

The inverse operation, *Detach*(μ), is implemented similarly. If μ is not the first block of *GetTrain*(μ), we first call *Exposegraph* at the left sibling of μ. Then, let ν be *GetTrain*(μ), and ν' be *Exposegraph* of *GetBlock* for the C-node parent of ν. We complete operation *Detach* by cutting ν from its parent then calling *ConcealGraph*(ν'). Therefore, operation *Detach* can be implemented in $O(\log^2 m)$ time.

We perform operation *Compose*(X, ρ', ρ''), with $X = S$ or $X = P$, as follows. Let vertices s', t', s'', and t'' be the first and second terminals of $M_{\rho'}$ and $M_{\rho''}$ found using operation *GetTerminals*. First, consider the case where $X = S$. As with operation *Attach*, we move the children of *Attachment*(s'') to become children of *Attachment*(t'). We then create a new S-node ρ with children ρ' and ρ''. If either or both of nodes ρ' and ρ'' are S-nodes, we then use operation *Contract* to restore the type invariant. We then restore the weight invariant by performing *ConcealGraph*(ρ).

When $X = P$ the implementation is similar. We move the children of *Attachment*(s'') and *Attachment*(t'') to become children of *Attachment*(s') and *Attachment*(t'). We then create a new P-node ρ with children ρ' and ρ'' and restore the type invariant using operation *Contract* and the weight invariant using operation *ConcealGraph*. Both cases can be implemented in $O(\log^2 m)$ time.

Operation *Inject*(ρ, λ) is implemented as follows. We begin by setting node ν to be *Exposegraph*(*GetBlock*(λ)). Suppose vertices s', t', s'', and t'' are the first and second terminals of $M_{\rho'}$ and M_λ found using operation *GetTerminals* (vertices s'' and t'' are the endpoints of the edge represented by λ). As described in the implementation of operation *Attach*, we move the children of *Attachment*(s'') and *Attachment*(t'') to become children of *Attachment*(s') and *Attachment*(t'). Let node μ be the parent of λ. We *Cut* node λ from μ, and *Link* node ρ. If nodes μ and ρ are the same type, we perform *Contract*(ρ) to restore the type invariant. We conclude by calling calling *ConcealGraph*(ν). Therefore, operation *Inject* can be implemented in $O(\log^2 m)$ time.

The inverse operation *Extract*(μ) is performed similarly. We begin by setting node ν to be *Exposegraph*(*GetBlock*(μ)). We then create a new Q-node λ, and use a constant number of *Cut* and *Link* operations to make children of λ the *Attachment* nodes of the terminals of μ. We then *Cut* μ from its parent. We restore the weight invariant by calling *ConcealGraph*(ν) and *ConcealGraph*(μ). Therefore, we perform operation *Extract* in $O(\log^2 m)$ time.

Operations *InsertVertex*(v, e, e', e'') and *DeleteVertex*(v, e, e', e'') are special cases of operations *Inject* and *Extract*. We implement operation *InsertVertex* by first building a SPQC-tree T' with root ρ representing a graph consisting of the series composition of edges e' and e''. This can be done with a constant number of *MakeGraph* and *Compose* operations. Let λ be the Q-node representing e. Operation *InsertVertex* is then performed by calling *Inject*(ρ, λ). All this is performed in $O(\log^2 m)$ time.

There are two cases for operation *DeleteVertex*(v, e, e', e''). Let λ' and λ'' be the Q-nodes representing e' and e''. If λ' and λ'' share the same parent μ, we simply

Extract(μ) then use a constant number of *Cut* and *DeleteGraph* operations to destroy μ, λ', and λ''. Otherwise, edges e' and e'' are in different blocks B' and B''. Assume B' is the parent block of B'' (the other case is symmetric). Since edges e' and e'' are the only edges incident to vertex v, blocks B' and B'' consist only of edges e' and e'', vertex v is the first terminal of $M_{\lambda'}$, and B' is the root block. We restructure such that B'' is the root block and edges e' and e'' are composed serially. We perform this with a single *Detach* and *Compose* operation. We then proceed as in the first case. Therefore, operation *DeleteVertex* is implemented in $O(\log^2 m)$ time.

Our implementation of operation *InsertEdge*(v', v'', e) consists of two steps. First determine if the pair (v', v'') is tw2-connectible. If so, we then restructure the associated SPQC-tree to represent the edge addition. Let nodes ν' and ν'' be *GetBlock*(*Attachment*(v')) and *GetBlock*(*Attachment*(v'')) and let B' and B'' be the associated blocks. Let nodes μ' and μ'' be *Proper*(v') and *Proper*(v''). Let vertices s', t', s'', and t'' be the first and second terminals of $M_{\nu'}$ and $M_{\nu''}$ found using operation *GetTerminals*.

Consider the case where vertices v' and v'' are both contained in the same block (either $\nu' = \nu''$, $v'' = s'$, or $v' = s''$). Suppose, without loss of generality, that node ν' represents the common block. Then, to test if the pair (v', v'') is tw2-connectible, we check the conditions of lemma 4.3. The first condition is where vertices v' and v'' are the terminals of $M_{\nu'}$. This condition is true if $v' = s'$ and $v'' = t'$, or $v' = t'$ and $v'' = s'$. The second condition is where v' and v'' are vertices of the skeleton of the same S-node. This condition is true if either $\mu' = \mu''$, v'' is a terminal of $M_{\mu'}$, or v' is a terminal of $M_{\mu''}$. The final condition is where (v', v'') form a cross-pair. This is true if μ' and μ'' are the only children of ν'. To restructure, we first set node ν to be *Exposegraph*(ν'). We then perform the restructuring described in the proof of lemma 4.3. This can be done with a constant number of *Link*, *Cut*, *Expand*, and *Contract* operations.

We now consider the general case where there does not exist a block containing both vertices v' and v''. Let node ν be $LCABlock(\nu', \nu'')$ and let B be the block represented by ν. Perform the following sequence of assignments. Let ν_1 = *Exposegraph*(ν'), ν_2 = *Exposegraph*(ν''), and ν_3 = *Exposegraph*(ν). If *GetTrain*(μ') returns ν_3, then let vertex $u' = v'$, otherwise let vertex u' be the first terminal of $M_{\nu'}$. Similarly, if *GetTrain*(μ'') returns ν_3, then let vertex $u'' = v''$, otherwise let vertex u'' be the first terminal of $M_{\nu''}$. Then, by lemma 4.5, the pair (v', v'') is tw2-connectible iff each of the following are true (see Fig. 4.7):

- either $u' = u''$ or the pair (u', u'') is 2spg-connectible in the block B;
- if $u' \neq v'$ then *Candidate*?(v') is **true**;
- if $u'' \neq v''$ then *Candidate*?(v'') is **true**.

If the pair (v', v'') is not tw2-connectible, then the weight invariant is restored by calling *ConcealGraph*(ν_3), letting ν' = *Exposegraph*(ν_1) and then calling *ConcealGraph*(ν'), and letting ν'' = *Exposegraph*(ν_2) and then calling *ConcealGraph*(ν'').

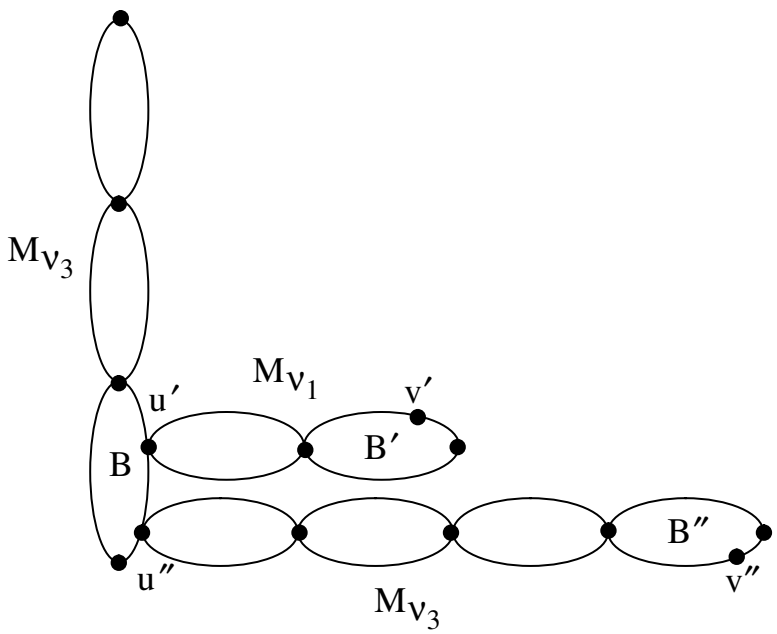

Figure 4.7: The items considered in operation *InsertEdge*. The pair (v', v'') is tw2-connectible if v' and v'' are both candidates and the pair (u', u'') is tw2-connectible.

To insert edge e, we then proceed as follows. If $u' \neq v'$ then $Detach(\nu_1)$ and $MakeTerminal(\nu', v')$, otherwise set $\nu_1 = nil$. Similarly, If $u'' \neq v''$ then $Detach(\nu_2)$ and $MakeTerminal(\nu'', v'')$, otherwise set $\nu_2 = nil$.

Consider the case with $u' \neq v'$. Vertices u' and v' are in the same block. We perform $InsertEdge(u', u'', \hat{e})$, as described above. Let node $\hat{\lambda}$ be the Q-node representing $\hat{e}$. Suppose, without loss of generality, that u' is the first terminal of $G_{\hat{\lambda}}$. Let node κ be the root of the SPQC-tree representing the series composition of graph G'_ν, edge e, and the swap of graph G''_ν. We do not restore the weight invariant for G_κ. We then perform $Inject(\kappa, \hat{\lambda})$. We conclude by restoring the weight invariant by $ConcealGraph(\nu_3)$). All this can be implemented in $O(\log^2 m)$ time. Figures 4.8, 4.9, and 4.10 demonstrate these steps.

Now, suppose $u' = v'$. Let node κ be the root of the SPQC-tree resulting from the parallel composition of $M_{\nu'}$ and the series composition of $M_{\nu''}$ and edge e. Vertices u and v are connected by edge e in G_κ. We complete the operation by performing $Attach(\kappa, u')$. Therefore, this case also can be implemented in $O(\log^2 m)$ time.

Operation $DeleteEdge(e)$ is implemented as follows. Let node λ be the Q-node representing e, and node μ be the parent of λ, if it exists. Let vertices u and v be the endpoints of e. We consider four cases: node λ is the root of its SPQC-tree, node μ is a C node, node μ is a P-node, and node μ is an S-node.

Suppose node λ is the root of its SPQC-tree. By the weight invariant, G_λ has no attachments to its second terminal. If there are no attachments to the first terminal

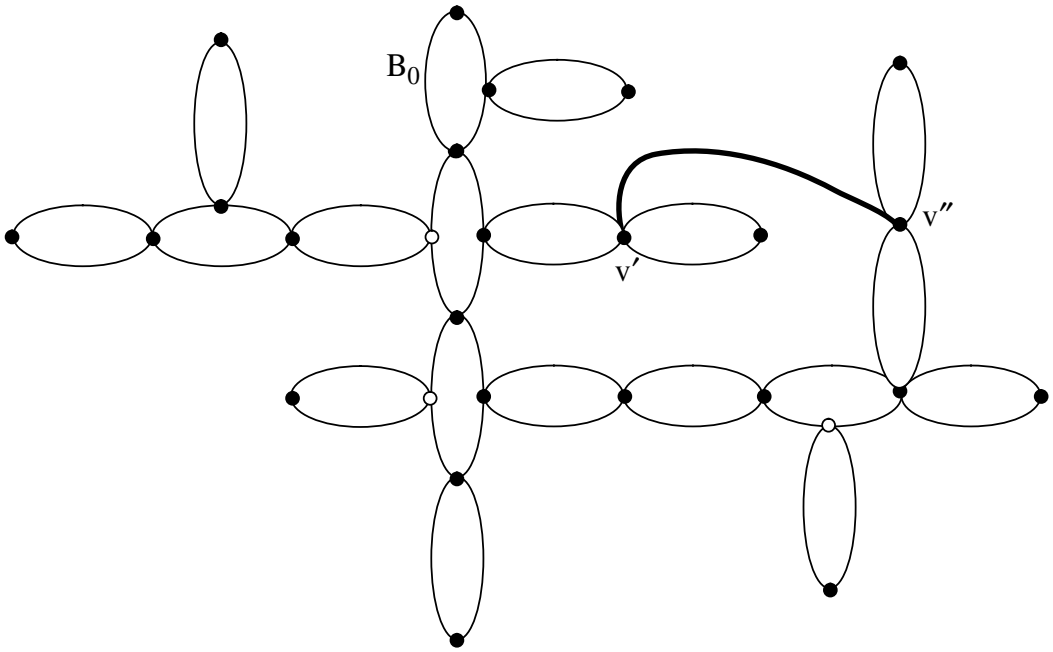

Figure 4.8: Inserting an edge from v' to v''.

v of G_λ, then we simply *DeleteGraph*(λ). Otherwise, let ν be the C-node child of λ representing vertex v. *Cut* ν from λ. Arbitrarily, *Cut* a child μ from ν. Node μ will now be the root of its tree. We conclude by linking ν to the Q-node found by *FindEdge*(μ, v).

When μ is a C-node, we also have that G_λ has no attachments to its second terminal. Let ν be *Exposegraph*(*GetBlock*(μ)). We then cut λ from μ and *DeleteGraph*(λ). We restore the weight invariant by calling *ConcealGraph*(ν).

Now, suppose node μ is a P-node. Let ν be *Exposegraph*(*GetBlock*(μ)). If λ has either *Attachment*(u) or *Attachment*(v) as children, then we relocate the attachments to Q-nodes found by calling *FindEdge*(μ', u) and *FindEdge*(μ', v) for any sibling μ' of μ. We then cut λ from μ. Let ν' be the parent of μ. If node μ now has only one child μ', we replace μ with μ' as a child of ν. This can be done with two *Link* operations and one *Cut* operation. If nodes μ' and ν' are of the same type, we restore the type invariant by performing *Contract*(μ'). Finally, if $\mu \neq \nu$ we restore the weight invariant by performing *ConcealGraph*(ν).

Now, consider the case where μ is an S-node. Let s and t be the terminals of G_μ. We first *Extract*(μ) without restoring the weight invariant for μ. We then *DeleteEdge*($\hat{e}$), where $\hat{e}$ is the edge created by *Extract*. The component chain Γ represented by μ is then divided into three subchains: the subchain Γ' preceding e, edge e, and the subchain Γ'' succeeding e. This can be accomplished with a constant number of *Expand* and *Cut* operations. Let ρ' and ρ'' be the nodes representing Γ' and Γ''. Note that one of Γ' and Γ'' may be empty. We restore the weight invariant by calling *ConcealGraph*(ρ') and

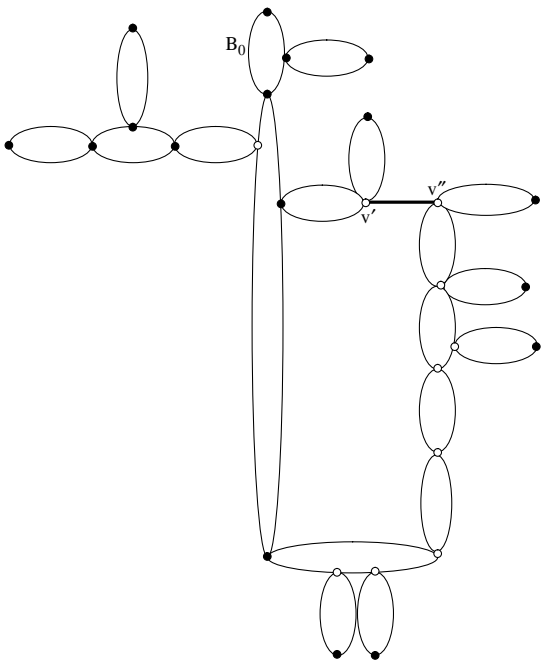

Figure 4.9: Restructuring the graph.

ConcealGraph(ρ''). We then conclude by calling *Attach*(ρ', s) and *Attach*(*Swap*(ρ''), t). Therefore, operation *DeleteEdge* can be implemented in $O(\log^2 m)$ time.

Theorem 4.1 *There is a fully dynamic data structure for maintaining the decomposition of a collection $\mathcal{G}$ of tree-width two graphs using linear expression trees with the following performance:*

1. *a tree-width two graph with m edge uses $O(m)$ space;*
2. *operations* MakeGraph *and* DeleteGraph *take each $O(1)$ time;*
3. *operations* Report, Update, *and* Swap *take each $O(\log m)$ time;*
4. *operations* Attach, Detach, InsertEdge, DeleteEdge, Inject, Extract, InsertVertex, *and* DeleteVertex *take each $O(\log^2 m)$ time, where m is the total number of edges of the tree-width two graphs involved.*

A framework for dynamic algorithms in bounded tree-width graphs is presented in [21]. They present two classes of problems on graphs, DECC and DLCC. The solution to problems in DECC can be represented as a tree attribute system for graphs of a bounded tree-width. The solution to problems in DLCC can be represented as a tree attribute system for bounded-degree graphs of a bounded tree-width.

The following problems are shown to be in DECC:

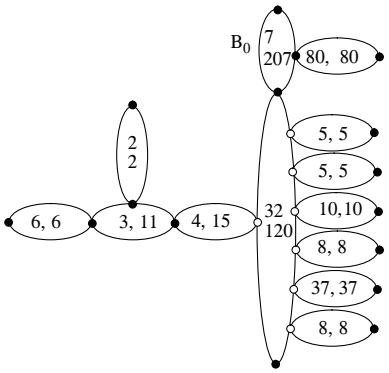

Figure 4.10: Graph obtained by inserting edge (v', v'') and restoring the weight invariant.

- Vertex Cover
- Dominating Set
- Chromatic Number
- Partial Feedback Edge Set
- Independent set
- Bipartite Subgraph
- Maximum Cut

The following problems are shown to be in DLCC:

- Minimum Maximal Matching
- Partition into Triangles
- Partition into Hamiltonian Subgraphs
- Partition into Cliques
- Monochromatic Triangle
- Transitive Subgraph
- Cubic Subgraph
- Kernel
- Chromatic Index

Lemma 4.7 [20] *Suppose the tree decomposition of a collection of graphs $\mathcal{G}$ of some bounded tree-width can be maintained such that each update or query operation requires $O(T(n))$ dynamic operations of theorem 3.3. Consider a DECC problem π_1. Then the solution to π_1 can be maintained in $O(T(n) \cdot \log n)$ time over $\mathcal{G}$ using a tree attribute system. Now consider a DLCC problem π_2. If the degree of any vertex of any graph in $\mathcal{G}$ is bounded to some constant, then the solution to π_2 can be maintained in $O(T(n) \cdot \log n)$ time over $\mathcal{G}$.*

Lemma 4.8 [20] *Suppose the SPQC-trees of a collection of tree-width two graphs $\mathcal{G}$ can be maintained such that each update or query operation requires $O(T(n))$ dynamic operations of theorem 3.3. Consider a DECC problem π_1. Then the solution to π_1 can be maintained in $O(T(n) \cdot \log n)$ time over $\mathcal{G}$ using a tree attribute system. Now consider a DLCC problem π_2. If the degree of any vertex of any graph in $\mathcal{G}$ is bounded to some constant, then the solution to π_2 can be maintained in $O(T(n) \cdot \log n)$ time over $\mathcal{G}$.*

Lemma 4.8 and Theorem 4.1 give the following:

Theorem 4.2 *Any graph decision problem π in DLCC can be solved dynamically in $O(\log^2 m)$ time per operation if the underlying graph is a bounded-degree tree-width two graph with m edges. If π is in DECC, then π can be solved dynamically for all tree-width two graphs, with the same time bounds.*

Example 4.1 Given a graph G, an *independent set* is a subset I of the vertices of G such that there is no edge in G between any pair of vertices of I. A *maximum independent set* is an independent set of maximum cardinality.

We maintain the solution to the following query on a SPQC-tree T representing tree-width two graph G:

- *MaxIndependantSet*(*node* μ) — returns the size of a maximum independent set of G_μ.

In order to solve this query, we keep the following attributes at each node μ:

- $mis(\mu, ab)$ — the size of a maximum independent set of G_μ subject to the condition indicated by ab. These conditions are:
  - 00 – neither terminal of M_μ is in the independent set.
  - 01 – the second terminal of M_μ is in the independent set, while the first terminal is not.
  - 10 – the first terminal of M_μ is in the independent set, while the second terminal is not.
  - 11 – both terminals of M_μ are in the independent set.

The equations to calculate these values are shown in table 4.1. The value of *MaxIndependantSet*(μ) is then $\max(mis(\mu, ab))$, where a and b can be either 0 or 1. It is easy to see that we can keep the equations to calculate the *mis* values in an unbounded linear expression tree.

□

Type	Condition	mis
S	ab	$\max(mis(\mu', ac) + mis(\mu'', cb))$
P	ab	$mis(\mu', ab) + mis(\mu'', ab)$
Q	ab	$mis(\mu', a0) + mis(\mu'', b0)$
C	a0	$\max(mis(\mu', ac) + mis(\mu'', bd))$

Table 4.1: The equations to calculate the values of $mis(\mu, ab)$. The values a, b, c, and d can be either 0 or 1. If there is an edge between the terminals of M_μ, then $mis(\mu, 11) = 0$. If μ is a Q-node leaf, then $mis(\mu, 00) = mis(\mu, 11) = 0$ and $mis(\mu, 01) = mis(\mu, 10) = 1$ If μ is a C-node leaf, then $mis(\mu, 00) = 0$, $mis(\mu, 10) = 1$, and the values $mis(\mu, 01)$ and $mis(\mu, 11)$ are undefined.

4.2.5 Auxiliary Operations

The proof of Theorem 4.1 concludes with the description of the auxiliary functions. In order to implement the auxiliary functions, we need to keep additional node, path, and tree attributes. We include in path attribute set $\mathcal{P}$, the transfer (A, b) pairs to represent the following values, which we shall see can be calculated by linear expressions:

- $weight(\mu)$ — The number of edges in the pertinent graph of μ.
- $maxchildwt(\mu)$ — The maximum weight among all children blocks connected to the pertinent train M_μ.
- $cnode(\mu, y)$ — True if the subtree rooted at μ contains the C-node associated with the terminal of M_μ indicated by y. The value $y = 1$ indicates the first terminal and the value $y = 2$ indicates the second terminal.

We calculate these values as follows. Consider the attribute $weight(\mu)$. If node μ is a Q-node, then the value of $weight(\mu)$ is 1. If μ is an S, P, or Q-node, then $weight(\mu)$ is the sum of the weights of the children of μ. Therefore, we can calculate $weight(\mu)$ as a linear expression.

Now consider attribute $maxchildwt(\mu)$. If μ is a C-node then $maxchildwt(\mu)$ is the maximum value of $weight$ of the children of μ. If μ is a S, P, or Q-node then $maxchildwt(\mu)$ is the maximum value of $maxchildwt$ of the children of μ. Therefore, we can calculate $maxchildwt(\mu)$ as a linear expression.

Finally, consider $cnode(\mu, y)$. If μ is a C-node then $cnode(\mu, y)$ is **false** for $y = 1$ or 2. If μ is a Q-node then $cnode(\mu, 1)$ is **true** when μ has a C-node child associated with its first terminal. Similarly, $cnode(\mu, 2)$ is **true** when μ has a C-node child associated with its second terminal. If μ is an S-node then $cnode(\mu, 1)$ takes the value of $cnode(\mu', 1)$, where μ' is the leftmost child of μ, and $cnode(\mu, 2)$ takes the value of $cnode(\mu'', 2)$, where μ'' is the rightmost child of μ. If μ is a P-node, then the values of $cnode(\mu, 1)$ and $cnode(\mu, 2)$ is the logical-or of the values at the children of μ. Therefore, we can calculate $cnode(\mu, y)$ as a linear expression where our semiring includes the logical-or operator.

We also keep the following entries in path attribute set $\mathcal{P}$ at each path tree node η.

- *headfirst*(η) (*headsecond*(η)) — indicates whether the vertex which is the first (second) terminal of the pertinent graph of *head*(η) is a terminal of the pertinent graph of *tail*(η). Possible values are 0, 1, or 2, indicating if at *tail*(η) the desired vertex is not a terminal, is the first terminal, or the second terminal.

- *tailfirst*(η) (*tailsecond*(η)) — indicates whether the vertex which is the first (second) terminal of the pertinent graph of *tail*(η) is a terminal of the pertinent graph of *head*(η).

- *contains*(η, x) — True if $\Pi(\eta)$ contains a node of type x.

We implement operation *Proper* using the following operation:

- *ProperTerm*(*node* μ, *terminal* x) — Let v be the terminal of M_μ identified by $x = 1$ or $x = 2$. Return the node triple (ν, μ', μ''), where ν is the proper node of v, node μ' is the child of ν such that v is the second terminal of μ', and μ'' is the child of ν such that v is the second terminal of μ''. If v is a terminal of the root, then *ProperTerm*(v) returns $(\rho, -, -)$, where ρ is the root of T.

We use the following path selection function which takes a terminal identifier x as an argument ($x = 1$ indicates the first terminal and $x = 2$ indicates the second terminal).

Selection Function S_7 *Suppose η is the root of a path tree with children η' and η''. Let Π' be the concatenation of the subpath represented by η' with the node* head(η''). *If $x = 1$ and* headfirst(Π') $\neq 0$, *or if $x = 2$ and* headsecond(Π') $\neq 0$ *then return η''. Else return η'.*

When used with operation *PathFind*, path selection function S_7 returns the tail of the longest subpath beginning at *head*(η) such that the terminal of *head*(η) indicated by x is a terminal of the returned node. Therefore, we implement *ProperTerm*(μ,) as follows. If node ν is the root of its SPQC-tree, then return $(\nu, -, -)$. Otherwise, let Π be *expose*(*Parent*(μ)). Let node ν be *PathFind*(*head*(Π), *tail*(Π), S_7). Let node μ be the child of ν on Π. If v is the second terminal μ, then let node μ' be the right sibling of μ and return (ν, μ, μ'). Otherwise, let node μ'' be the left sibling of μ and return (ν, μ'', μ).

Operation *Proper*(v) is then performed as *ProperTerm*(*Attachment*(v), 1).

Operation *GetTerminals*(μ) also is implemented using operation *ProperTerm*. We will describe how to find the first terminal of M_μ. Finding the second terminal is performed similarly.

Suppose v is the first terminal of M_μ. We find v by finding *Attachment*(v). This is performed with the following two steps:

1. Find a node ν such that v is a terminal of M_ν and *Attachment*(v) is a descendant of ν. Additionally, find integer x such that if v is the first terminal of M_ν, then $x = 1$, or if v is the second terminal of M_ν, then $x = 2$.

2. Find $Attachment(v)$.

For step 1, let (ν', μ', μ'') be the triple returned by $ProperTerm(\mu, 1)$. If ν' is the root, then let $\nu = \nu'$. Otherwise, let ν be either μ' or μ'', depending if $cnode(\mu', 2) =$ **true** or $cnode(\mu'', 1) =$ **true**. Let $q = headfirst(\Pi)$, where Π is the path from μ to ν'.

Step 2 uses the following tree selection function: S_8 takes an argument q which has values either 1 or 2:

Selection Function S_8 *Suppose η is the root of a path tree with children η' and η''. Let Π' be the concatenation of* tail(η') *with the subpath represented by η''. If $q = 1$, then let $q' = tailfirst(\Pi')$. If $q = 2$, then let $q' = tailsecond(\Pi')$. If $q' = 0$ or if* cnode(tail$(\eta'), q') =$ **false**, *then return η''. Otherwise, return η'.*

Step 2 is then implemented as $TreeFind(\mu, S_8, x)$.

Operations *GetTrain* and *GetBlock* use the following path selection function.

Selection Function S_9 *Suppose η is the root of a path tree with children η' and η''. If* contains(η', C) *is* **true**, *or if* head(η'') *is a C-node, then return η', else return η''.*

When used with operation *PathFind*, path selection function S_9 returns the tail of the longest subpath beginning at $head(\eta)$ that does not contain a C-node. Operation $GetTrain(\mu)$ is then implemented as follows. If μ is not a C-node, then let path Π be $expose(\mu)$. Otherwise let Π be $expose(Parent(\mu))$. Return $PathFind(head(\Pi), tail(\Pi), S_9)$.

Operation $GetBlock(\mu)$ is implemented similarly. Let node ν be $GetTrain(\mu)$, and let node ν' be the child of ν on Π. If μ is a C-node representing vertex v and v is the first terminal of $M_{\nu'}$, then v will also be the first terminal of $M_{\nu''}$, where ν'' is the left sibling of ν'. M''_ν is a lower level block than M'_ν, Therefore return ν''. For any other case, return ν'.

Operation $LCABlock(\mu', \mu'')$ is performed simply as $GetBlock(LCA(\mu', \mu''))$.

Operation $MakeTerminal(\mu, v)$ is implemented as follows. If v is the second terminal of G_μ do nothing. Otherwise, let ν be the parent of μ. We first cut μ from ν. Let s be the first terminal of G_μ. Since the pair (s, v) is 2spg-connectible, vertex s must be the first terminal of $Proper(v)$. We perform the restructuring described in the proof of lemma 4.4. This can be done with a constant number of *Link*, *Cut*, *Expand*, *Contract*, and *Evert* operations. We then link the root of the restructured tree to ν in place of μ.

Operation $FindEdge(\mu, v)$ is similar to operation *Proper*. We use the following tree selection function S_{10} to implement *FindEdge*. Selection function S_{10} takes an argument q which has values either 1 or 2, indicating if v is the first or second terminal of G_μ.

Selection Function S_{10} *Suppose η is the root of a path tree with children η' and η''. Let Π' be the concatenation of* tail(η') *with the subpath represented by η''. If $q = 1$, then let $q' = tailfirst(\Pi')$. If $q = 2$, then let $q' = tailsecond(\Pi')$. When $q' \neq 0$ return η', otherwise return η''.*

When used with operation *TreeFind*, tree selection function S_{10} returns a node that does not have a child that has v as a terminal. Operation $FindEdge(\mu, v)$ is then

implemented as follows. Call *GetTerminals*(μ) in order to determine if vertex v is the first or second terminal of G_μ. Then return *TreeFind*(μ, S_{10}, q), where q is set to 1 or 2 when v is the first or second terminal of G_μ.

We implement operation *Candidate?*(v, μ) as follows. If v is the second terminal of M_μ, then return **true**. Otherwise, let node triple (ν, ν', ν'') = *Proper*(v) and let s be the first terminal of M_μ. If s is also the first terminal of M_ν then return **true**, else return **false**. The correctness of this process follows from lemma 4.5.

Operations *Exposegraph* and *ConcealGraph* are implemented with the following operations.

- *open*(*node* μ) — If it exists, open the coupled connection to the second terminal of pertinent train M_μ. Node μ is assumed to represent a block.
- *couple*(*node* ν) — Couple the connection between the first block B of the train represented by node μ and its parent block B'. Block B is assumed to be a candidate child, and block B' is assumed to have no coupled connection.
- *findbest*(*node* μ) — Returns the node that represents the maximum weight child of M_μ.
- *splicegraph*(*node* ν) — Suppose B is the first block of the train represented by node ν and and B is a candidate child of its parent block B'. If it exists, open the coupled connection to the second terminal of B'. Couple the connection between B and B'.
- *slicegraph*(*node* μ) — Suppose B is the block represented by node μ. If it exists, open the coupled connection to the second terminal of B. Let node ν be *findbest*(μ). Couple the connection between B and M_ν if M_ν is a heavy candidate child.
- *lightgraph*(*node* ν) — Suppose node ν represents a train M. Return the child μ of ν such that the connection between the blocks represented by node μ and its right sibling is light. If no such connection exists, then return *nil*.

Operation *open*(μ) is implemented as follows. Let vertex v be the second terminal of v and node ν be the parent of μ. If node μ is the rightmost child of ν, then do no action. Otherwise, expand the children of ν succeeding node μ. Let ν' be the new S-node. Cut ν' from ν and link it to *Attachment*(v). All this can be done in $O(\log m)$ time with a constant number of *Link*, *Cut*, and *Expand* operations.

Operation *couple*(ν) is implemented as follows. Let κ be the C-node parent of ν and v be the vertex represented by κ. Let nodes ν' and μ' be *GetTrain*(κ) and *GetBlock*(κ). We perform operation *couple* by first cutting ν from κ. We then complete the operation by making v the second terminal of $M_{\mu'}$, linking ν to ν', and restoring the type invariant by contracting ν into ν'. All this can be done in $O(\log m)$ time with a constant number of *Link*, *Cut*, *MakeTerminal*, and *Contract* operations.

Operation *findbest*(μ) is implemented using a blocking heap (see section 3.5.3, page 41). Operation *findbest*(μ) is implemented by calling operation *FindSources*(μ) to find a source of *maxchildwt*(μ).

Operation *splicegraph*(ν) is implemented as follows. Let node μ be *GetBlock*(*Parent*(ν)). Node μ is the node representing the parent block of the first block of M_ν. Perform *open*(μ) then *couple*(ν).

Operation *slicegraph*(μ) is implemented similarly. If μ is not the rightmost child of its parent, we call *open* for the right sibling of μ. Let node ν be *findbest*(μ). Let κ be the C-node parent of ν and let v be the associated vertex. If *Candidate*?(v) = **true** and $2 \cdot$ *weight*(ν) > *weight*(μ) then we *couple*(*findbest*(μ)).

Operation *lightgraph*(ν) is implemented as follows. Let $\Pi_\ell(\nu)$ be the edge path of ν containing all the children of ν. Path $\Pi_\ell(\nu)$ is the left edge path of ν after we issue *expose*(ν). The *local weight* (*localwt*(μ)) of a child μ of ν is the candidate weight of μ without the candidate weight of any coupled connection to M_μ. Suppose node μ' is the right sibling of node μ. The *graph tilt* of μ, written *graphtilt*(μ), is the difference between *weight*(μ') and *localwt*(μ). Then the coupled connection between M_μ and $M_{\mu'}$ is light if *graphtilt*(μ) < 0. Therefore, operation *lightgraph*(ν) returns the rightmost child μ of ν with *graphtilt*(μ) < 0. Note every path Π has a light connection since *graphtilt*(*head*(Π)) < 0.

We augment path attribute set $\mathcal{P}$ with the following for each edge path $\Pi_X(\nu)$ ($X = \ell$ or r), where ν is an S-node:

- *mingraphtilt*($\Pi_X(\nu)$) — the minimum value of *graphtilt* over the nodes of $\Pi_X(\nu)$.
- *totalwt*($\Pi_X(\nu)$) — the total of *localwt* over the nodes of $\Pi_X(\nu)$.

These values are not defined for any other type of path.

The value of *mingraphtilt* and *totalwt* can be maintained in a path attribute system. Suppose path $\Pi_X(\nu)$ is the concatenation of paths $\Pi'_X(\nu)$ and $\Pi''_X(\nu)$. We have,

$$\begin{aligned} \mathit{totalwt}(\Pi_X(\nu)) &= \mathit{totalwt}(\Pi'_X(\nu)) + \mathit{totalwt}(\Pi''_X(\nu)) \\ \mathit{mingraphtilt}(\Pi_X(\nu)) &= \min(\mathit{mingraphtilt}(\Pi''_X(\nu)), \mathit{totalpathwt}(\Pi''_X(\nu)) - \mathit{localwt}(\mathit{tail}(\Pi'_X(\nu))), \\ &\quad \mathit{totalwt}(\Pi''_X(\nu)) + \mathit{mingraphtilt}(\Pi'_X(\nu))) \end{aligned}$$

We use the following path selection function to find *lightgraph*(v):

Selection Function S_{11} *Suppose η is the root of a path tree with children η' and η''. If* mingraphtilt(η'') < 0 *then return η'', else return η'.*

Operation *lightgraph*(ν) is then implemented as follows. Call *expose*(ν) to get left edge-path $\Pi_\ell(\nu)$. Let ζ be the root of the path tree for $\Pi_\ell(\nu)$. Let $\nu(\mu)$ be the edge-path node returned by *PathFind*(ζ, S_{11}). If $\nu(\mu)$ = *head*(ζ) then return *nil*, else return μ.

Operation *Exposegraph*(μ) is then implemented by first calling *open*(μ). Let node ν be *GetTrain*(μ) and vertex v be the first terminal of pertinent train M_ν. Perform the following while ν is not the root and *Candidate*?(v) = **true**: Let node ν be *splicegraph*(ν) and vertex v be the first terminal of M_ν. When the loop terminates, return node ν.

Operation *ConcealGraph*(ν) is implemented as follows. Let node μ = *lightgraph*(ν). Then perform the following while $\mu \neq$ *nil*: Call *slicegraph*(μ), then reset μ = *lightgraph*(ν).

4.2.6 Dynamic Operations for Series-Parallel Graphs and Trees

If we restrict our attention to series-parallel graphs or trees, we can improve our time bounds to $O(\log m)$ per dynamic operation.

Series-Parallel Graphs

The following lemmas indicate when an edge can be inserted or deleted in a series-parallel graph:

Lemma 4.9 *Let G be a series-parallel graph represented by TTSP G^τ with associated SPQ-tree T with root ρ. Let $(B_0, c_1, B_1, \ldots, c_p, B_p)$ be the component chain that comprises the blocks and join vertices of G. Suppose v' and v'' are vertices of G. Then the graph $\hat{G}$ resulting from adding an edge e from v' to v'' in G is a series-parallel graph iff the pair (v', v'') is tw2-connectible and one of the following is true:*

1. *Vertices v' and v'' are the terminals of G_ρ.*
2. *Both v' and v'' are vertices of the skeleton of the same S-node in T.*
3. *At least one of v' or v'' is a vertex of block B_0 other than c_1.*
4. *At least one of v' or v'' is a vertex of block B_p other than c_p.*

Proof:

(If:) If vertices v' and v'' meet conditions 1 or 2 of the lemma, then $\hat{G}^\tau$ clearly is a TTSP with an additional parallel composition. Suppose vertices v' and v'' meet either condition 3 or condition 4. Then either (v', v'') is a cross-pair of block B_0 or B_p, or there is not a block containing both vertex v' and vertex v''.

Suppose (v', v'') is a cross-pair of block B_0. Then, as in the proof of lemma 4.3, we can restructure the representation of B_0 in G_τ such that vertices u and c_1 are terminals. We get a representation $G^{(u,t_2)}$ of G and either condition 1 or 2 hold. We perform the similar restructuring if (v', v'') is a cross-pair of block B_p.

Finally, if vertices v' and v'' meet condition 3 and there is not a block containing both vertices v' and v'', we find a representation of $\hat{G}$ as follows. Assume, without loss of generality, v' is the vertex of block B_0, $v' \neq c_1$. Let i be the minimum such that $v'' \in B_i$. We show that $\hat{G}^{(c_i,t_2)}$ is a representation of $\hat{G}$. By lemma 4.5, the pair (v', c_1) and (c_i, v'') are 2spg-connectible. Therefore, $G^{(v',t_2)}$ is a representation of G. We separate the join at vertex c_i to get TTSPs $G_1^{(v',c_i)}$ and $G_2^{(c_i,t_2)}$. Perform a serial composition of edge e with $G_1^{(v',c_i)}$ to get $\overline{G_1}^{(v',c_i)}$. Insert an edge $\hat{e}$ from c_i to v'' to get $\overline{G_2}^{(c_i,t_2)}$. Finally, we get $\overline{G}^{(c_i,t_2)}$ by injecting $\overline{G_1}^{(c_i,v')}$ for $\hat{e}$. The restructuring is similar for condition 4.

(Only If) The pair (v', v'') must be tw2-connectible, since a series-parallel graph is also a tree-width two graph. Suppose (v', v'') is tw2-connectible, but none of conditions $1-4$ hold. Therefore, either (v', v'') is a cross-pair of block B_i, $i \neq 0, p$, or there is not a block containing both vertex v' and v'' and neither vertex is contained in B_0 or B_p.

Suppose (v', v'') is a cross-pair of block B_i, $i \neq 0, p$. In order for $\hat{G}$ to be a series-parallel graph, we must find a representation of block B_i with terminals c_i and c_{i+1}. However, if we insert an edge in $\hat{G}$ between vertices c_i and c_{i+1}, then we can find K_4 as a minor considering vertices v', v'', c_i, and c_{i+1}. This is a contradiction by lemma 4.4.

Finally, suppose the vertex pair (v', v'') is tw2-connectible, there is not a block containing both v' and v'', and neither of v' or v'' is contained in block B_0 or block B_p. Assume, without loss of generality, that if $v' \in B_i$ and $v'' \in B_j$ then $i < j$. Let i be the maximum such that $v' \in B_i$ and j be the minimum such that $v'' \in B_j$. Either vertex $v' \neq c_i$ or $v'' \neq c_{j+1}$, since otherwise v' and v'' would be vertices of $skeleton(\rho)$. Assume $v' \neq c_i$. Let B be the block containing edge e in $\hat{G}$. In order for $\hat{G}$ to be a series-parallel graph, we must find a representation of block B with terminals c_i and c_{j+1}. However, if we insert an edge in $\hat{G}$ between vertices c_i and c_{j+1}, then we can find K_4 as a minor considering vertices v', c_i, c_{i+1} and c_{j+1}. This is a contradiction by lemma 4.4. A similar argument holds if $v'' \neq c_{j+1}$. □

Lemma 4.10 *Let G be a series-parallel graph represented by TTSP G^τ with associated SPQ-tree T with root ρ. Let $(B_0, c_1, B_1, \ldots, c_p, B_p)$ be the component chain that comprises the blocks and join vertices of G. Suppose e is an edge of G represented by Q-node λ in T. If $\lambda \neq \rho$ then let node μ be the parent of λ and let t' and t'' be the terminals of μ. Then the graph $\hat{G}$ resulting from deleting edge e G is a series-parallel graph iff node $\lambda \neq \rho$ and one of the following is true:*

1. *Node μ is a P-node.*
2. *Node μ is an S-node and one of the following is true.*
   (a) *Graph G is biconnected.*
   (b) *Node $\mu = \rho$ and node λ is the first or last representative of μ.*
   (c) *Edge e is in block B_0, node λ is the first representative of μ, and the pair (t_2, c_1) is tw2-connectible.*
   (d) *Edge e is in block B_0, node λ is the last representative of μ, and the pair (t_1, c_1) is tw2-connectible.*
   (e) *Edge e is in block B_p, node λ is the first representative of μ, and the pair (t_2, c_p) is tw2-connectible.*
   (f) *Edge e is in block B_p, node λ is the last representative of μ, and the pair (t_1, c_p) is tw2-connectible.*

Proof:

(If:) If condition 1 is true, then $\hat{G}^\tau$ is G^τ with one fewer parallel compositions.

Suppose condition 2a is true. Then by lemma 4.4, $G^{(t',t'')}$ is a representation of G. We find a representation $\hat{G}^{(v',v'')}$ as follows. We perform $Extract(\mu)$, which replaces M_μ by an edge e'. We then delete e'. Let G_1 be the resulting series-parallel graph. We then delete edge e from G_μ to create (in general) two TTSPs, $G_2^{(t',v')}$ and $G_3^{(v'',t'')}$. Finally, $\hat{G}^{(v',v'')}$ is the serial composition of TTSPs $G_2^{(v',t')}$, $G_1^{(t',t'')}$, and $G_3^{(t'',v'')}$.

For condition 2b, we find a representation $\hat{G}^{(s'',t_2)}$ as follows. We perform $Extract(\mu)$, which replaces M_μ by an edge e'. We then delete e'. Let G_1 be the resulting series-parallel graph. Since (s'', c_1) is tw2-connectible, we then make s'' the first terminal. Finally, $\hat{G}^{(v'',t_2)}$ is the serial composition of the graph represented by $Swap(\mu)$ with $G_1^{(s'',t_2)}$. We handle conditions 2b, 2c, and 2d similarly.

(Only If) Suppose $\lambda = \rho$. Then $\hat{G}$ is the empty graph, which is not a series-parallel graph. Now, suppose neither condition 1 or 2 hold. Then graph G is not biconnected, node μ must be an S-node, and one of the following must be true.

The first case is if node λ is neither the first nor last representative of μ. Then, if $\mu = \rho$ $\hat{G}$ is disconnected and consists of two series-parallel graphs. Otherwise, there exists a block of $\hat{G}$ with three cutvertices, contradicting $\hat{G}$ being a series-parallel graph. Similarly, if edge e is not in B_0 or B_p, then also there exists a block of $\hat{G}$ with three cutvertices.

The next case is if node λ is the first representative of μ and edge e is in block B_0, but the pair (v'', c_1) is not tw2-connectible. Then, under any representation of $\hat{G}$, there exists a block B with a cutvertex that is not a terminal of B, contradicting $\hat{G}$ being a series-parallel graph. A similar argument holds for the remaining cases. □

We do not maintain the weight invariant for series-parallel graphs. Each series-parallel graph G is represented as a single train and is associated with an SPQ-tree T. We support operations , *Report*, *Change*, *Swap*, *MakeGraph*, *DeleteGraph*, *Detach*, *Compose*, *InsertEdge*, *DeleteEdge*, *Inject*, *Extract*, *InsertVertex*, and *DeleteVertex*. These operations are implemented as described in section 4.2.4, without maintaining the *Attachment* nodes, or maintaining the weight invariant.

Using the techniques we have developed, we can test the conditions and perform the updates described in lemma 4.9 and lemma 4.10 in $O(\log m)$ for a graph with m edges. For operation $DeleteEdge(\lambda)$ we allow two conditions where the resulting graph is not a series-parallel graph. The first is where λ is the root of its SPQ-tree. In this case, performing $DeleteEdge(\lambda)$ is equivalent to performing $DeleteGraph(\lambda)$. The second case is where the parent μ of λ is an S-node, μ is the root of its tree, and λ is not the first or last child of μ. Then $DeleteEdge(\lambda)$ results in two series-parallel graphs.

We summarize with the following:

Theorem 4.3 *There is a fully dynamic data structure for maintaining the decomposition of series-parallel graphs using linear expression trees with the following performance:*

1. *a series-parallel graph with m edge uses $O(m)$ space;*

2. *operations* MakeGraph *and* DeleteGraph *take each $O(1)$ time;*

3. *operations* , Report, Update, Swap, Detach, InsertEdge, DeleteEdge, Inject, Extract, InsertVertex, *and* DeleteVertex *take each* $O(\log m)$ *time, where* m *is the total number of edges of the series-parallel graphs involved.*

Then, by lemma 4.8 and Theorem 4.3 we get the following:

Theorem 4.4 *Any graph decision problem* π *in* $DLCC$ *can be solved dynamically in* $O(\log m)$ *time per operation if the underlying graph is a bounded-degree series-parallel graph with* m *edges. If* π *is in* $DECC$*, then* π *can be solved dynamically for any series-parallel graph, with the same time bounds.*

Trees

Recall that tree-width one graphs are forests. Therefore, each connected component of a tree-width one graph is a tree.

Given a tree T, we construct a tree decomposition of T by storing the pair (μ, ν) at a node μ with parent ν. The root stores only itself. Clearly, we can maintain this decomposition implicitly under operations *Link*, *Cut*, *Expand* and *Contract*. Then, by lemma 4.7 and Theorem 3.3 we get the following:

Theorem 4.5 *Any graph decision problem* π *in* $DLCC$ *can be solved dynamically using a tree attribute system in* $O(\log n)$ *time per operation if the underlying graph is a bounded-degree tree with* n *nodes. If* π *is in* $DECC$*, then* π *can be solved dynamically for any tree, with the same time bounds.*

4.3 Other Dynamic Problems

In this section we study other problems which have dynamic algorithms on tree-width two graphs. Namely, given a tree-width two graph G with real-valued edge weights, we consider the problem of answering the following queries in a fully dynamic environment:

4.3.1 All Connectible-Pairs Shortest Path / Minimum Cut

Suppose we are given a graph G with positive valued weighted edges. Let u and v be distinct vertices of G. The *shortest-*(u, v) *path problem* considers the edge weights to be lengths. The problem consists of finding a path between u and v of minimum total length. The *minimum-*(u, v) *cut problem* considers the edge weights to be capacities. A (u, v)*-cut* in G is a set C of edges of G such that every path between vertices u and v in G contains an edge of C. The problem consists of finding a (u, v)-cut of minimum total capacity. It is well known that the value of the minimum (u, v)-cut is equivalent to the value of the maximum (u, v)-flow.

For a tree-width two graph G, we wish to answer the following queries:

- *MinimumCut*(*vertex* v', v'') — Find the size of a minimum edge cut separating the tw2-connectible pair of vertices v' and v''.

- *ShortestPath*(*vertex* v', v'') — Find the length of a shortest path between the tw2-connectible pair of vertices v' and v''.
- *ReportPath*(*vertex* v', v'') — returns a shortest path between the tw2-connectible pair of vertices v' and v''.
- *ReportCut*(*vertex* v', v'') — returns a minimum cut between the tw2-connectible pair of vertices v' and v''.

As with our general technique, we begin by considering series-parallel graphs, then extend our technique to tree-width two graphs. Suppose we are given a TTSP G^τ with positive valued weighted edges. Let τ be the terminal pair (t_1, t_t). The *shortest terminal path problem* and the *minimum terminal cut problem* consists of finding the shortest (t_1, t_2)-path and minimum (t_1, t_2)-cut in G.

The problems of determining the value of a minimum terminal cut and the length of a shortest terminal path are closely related for a TTSP G^τ . Let T be the SPQ-tree of G. We associate with each node μ of T the values $mincut(\mu)$ and $shortest(\mu)$, where $mincut(\mu)$ is the minimum terminal cut value of the pertinent graph G_μ of μ, and $shortest(\mu)$ is the length of a shortest terminal-path in G_μ. (For a Q-node λ, $mincut(\lambda) = capacity(e)$ and $shortest(\mu) = length(e)$, where e is the edge associated with λ.) Let $children(\mu)$ denote the set of children of node $\mu \in T$. We have:

$$mincut(\mu) = \begin{cases} \sum_{\lambda \in children(\mu)} mincut(\lambda) & \text{if } \mu \text{ is a P-node} \\ \min_{\lambda \in children(\mu)} mincut(\lambda) & \text{if } \mu \text{ is an S-node} \end{cases}$$

$$shortest(\mu) = \begin{cases} \min_{\lambda \in children(\mu)} shortest(\lambda) & \text{if } \mu \text{ is a P-node} \\ \sum_{\lambda \in children(\mu)} shortest(\lambda) & \text{if } \mu \text{ is an S-node} \end{cases}$$

Therefore, the SPQ-tree associated with a TTSP can be viewed as an expression tree with operators min and + over the reals (note that + distributes over min). For the minimum terminal cut (resp. shortest terminal path) problem, P-nodes store operator + (resp. min) and S-nodes store operator min (resp. +). The Q-nodes store the variables of the expression, i.e., the edge weights.

Now suppose G is a tree-width two graph represented by SPQ-tree T and (v', v'') is a tw2-connectible pair of vertices. We implement queries $MinimumCut(v', v'')$ and $ShortestPath(v', v'')$ as follows. We proceed as in the implementation of operation *InsertEdge*. We find the lowest level blocks B' and B'' which contain vertices v' and v''. Let block B be the least common ancestor block of B' and B''. Let train M' be between blocks B' and B. Similarly, let train M'' be from B'' to the child of B on the path to B''. Note that train M'' may be empty. We then construct train M with terminals v' and v'' by combining trains M' and M'' and changing terminals. All

this can be done with a constant number of calls to operations *GetBlock*, *GetTrain*, *MakeTerminals*, *Cut* and *couple*.

Let ν be the node such that $M_\nu = M$. Clearly, any edge on a shortest (v', v'')-path or a minimum (v', v'')-cut will be contained in train M. Therefore, we simply return the value of $shortest(\nu)$ and $mincut(\nu)$. We then restore our structure by calling the appropriate *Link*, *open*, and *ConcealGraph* operations.

We now show how to implement operation *ReportPath*. Operation *ReportCut* is implemented similarly, replacing the roll of S and P-nodes.

Let ν be the node created as above such that v' and v'' are the terminals of M_ν. We implement $ReportPath(v', v'')$ by calling the following recursive operation:

- *ReportTerminalPath*(*node* ν) — returns a shortest path between the terminals of M_ν.

Operation $ReportTerminalPath(\nu)$ is implemented as follows:

1. If ν is a Q-node, then return ν.
2. If ν is an S-node with children $\mu_1, \ldots, \mu_d$, then call $ReportTerminalPath(\mu_i)$, where i ranges from 1 to d.
3. If ν is an P-node with children $\mu_1, \ldots, \mu_d$, then call $ReportTerminalPath(\mu_j)$, where μ_j is a child of ν such that $shortest(\mu_j) = shortest(\nu)$.

We implement step 3 by calling $expose(\nu)$ then keeping a heap on the left edge-path of ν (see example 3.5, page 26). Therefore, steps 2 and 3 take $O(\log m)$ time plus $O(1)$ time for each recursive call to *ReportTerminalPath*. Suppose k edges are returned. Consider the subtree T' of visited nodes. There are k Q-nodes and $O(k)$ P-nodes in T'. Additionally, each S-node has more than 1 child in T'. Hence, subtree T' has $O(k)$ nodes. Therefore, operation *ReportPath* takes $O(\log^2 m + k \log m)$ time to return k edges.

Theorem 4.6 *Given a tree-width two graph G with m weighted edges, there exists a fully dynamic data structure that uses $O(m)$ space and supports update operations and query operations* MaxFlow *and* ShortestPath *in $O(\log^2 m)$ time and query operations* ReportPath *and* ReportCut *in $O(\log^2 m + k \log m)$ time to return k edges.*

If we restrict G to be a series-parallel graph, then we perform update operations and query operations MaxFlow *and* ShortestPath *in $O(\log m)$ time and query operations* ReportPath *and* ReportCut *in $O(k \log m)$ time to return k edges.*

4.3.2 Shortest st-Path / Minimum st-Cut in Series-Parallel Digraphs

The queries of the previous section also can be maintained dynamically for the directed version of TTSPs known as *series-parallel digraphs*.

A *source* of a digraph G is a vertex without incoming edges. A *sink* is a vertex without outgoing edges. A *pole* of G is either a source or a sink. A series-parallel digraph is a digraph G with exactly one source s and one sink t, recursively defined as follows:

- A digraph G consisting of a single edge from s to t is a series-parallel digraph.
- Given series-parallel digraphs $G_1, \ldots, G_j$, the digraph G obtained by identifying the sources of each graph into a single vertex s and identifying the sinks into a single vertex t is a series-parallel digraph. This is called *parallel composition.*
- Given series-parallel digraphs $G_1, \ldots, G_k$ with sources $s_1, \ldots, s_k$ and sinks $t_1, \ldots, t_k$, the digraph G obtained by identifying sink t_i with source s_{i+1} for $1 \leq i < k$ is a series-parallel digraph. The source and sink of G are vertices s_1 and t_j. Vertices $v_i = t_i = s_{i+1}$, $1 \leq i < k$ are called the *join-vertices* of such a composition. This is called *parallel composition.*

A *series-parallel digraph* is an acyclic digraph with one source and one sink recursively defined as: a single directed edge, or the *series composition* of series-parallel digraphs, or the *parallel composition* of series-parallel digraphs. Clearly, we can also use a SPQ-tree T to represent a series-parallel digraph G.

The minimum st-cut of series-parallel digraph G is the minimum cut separating the poles of G. The shortest st-path of series-parallel digraph G is the shortest path between the poles of G. The equations presented in the previous section for the calculation of shortest terminal path and minimum terminal cut are identical for the directed version of these problems.

The following lemmas show when edges can be added and deleted from series-parallel digraph G with associated SPQ-tree T:

Lemma 4.11 *Let G be a series-parallel graph with SPQ-tree $\mathcal{T}$, and v' and v'' be two vertices of G. Then the digraph $\bar{G}$ obtained by inserting an edge from v' to v'' is a series-parallel graph if and only if one of the following conditions is true:*

1. *$v' = s$ and $v'' = t$, or*
2. *v' and v'' are vertices of the skeleton of the same S-node of $\mathcal{T}$, with v' preceding v''.*

Proof:

(If) If v' and v'' meet either condition in the lemma, then $\bar{G}$ is clearly a series-parallel graph with an additional parallel composition.

(Only If) We use the property that a series-parallel graph has no subgraph homeomorphic to the digraph W shown in Fig. 4.11 [118]. Suppose that v' and v'' do not meet either condition of the lemma. Let μ be the least common ancestor in $\mathcal{T}$ of the proper nodes of v' and v''. Let e be the edge from v' to v''. If μ is a P-node then we consider the subgraph of $\bar{G}$ consisting of edge e and two directed paths from the poles of G_μ through the pertinent graphs of the children of μ. Such a digraph is homeomorphic to W, which implies that $\bar{G}$ is not a series-parallel graph. Now, suppose that μ is an S-node; G has a directed path either from v' to v'' or from v'' to v'. If G has a path from v'' to v', then $\bar{G}$ has a cycle and hence is not a series-parallel graph. Thus, G has a path from v' to v''. Since Condition 2 is not met, at least one of v' and v'', say v'', is not a vertex of $skeleton(\mu)$, so that the pertinent digraph of a child of μ contains v''

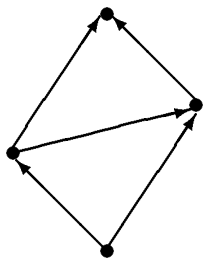

Figure 4.11: Digraph W that is the forbidden homeomorphic subgraph for a series-parallel digraph.

and has a directed path p between its poles that avoids v''. Thus, we can identify in $\bar{G}$ a subgraph homeomorphic to W that consists of edge e, path p, and a path from v' to the sink of G_μ through v'', so that $\bar{G}$ is not a series-parallel graph. □

The following lemma determines when we can delete an edge.

Lemma 4.12 *Let G be a series-parallel graph G with SPQ-tree $\mathcal{T}$, and e be an edge of G represented by Q-node λ. The digraph obtained by removing edge e is a series-parallel graph if and only if one of the following conditions is true:*

1. *the parent of λ in $\mathcal{T}$ is a P-node,*
2. *e is the only outgoing edge of s, or*
3. *e is the only incoming edge of t.*

Proof: If the parent of λ in $\mathcal{T}$ is a P-node, then G' is the series-parallel graph that results by omitting the parallel composition step on e in the construction of G. If the parent of λ is an S-node, then removing e adds a second source or disconnects G' unless e is the only edge originating at s or e is the only edge terminating at t. □

Using the techniques we have developed, we can test the conditions and perform the updates described in lemma 4.9 and lemma 4.10 in $O(\log m)$ for a graph with m edges.

Therefore, using the methods of theorem 4.3, we get:

Theorem 4.7 *Given a series-parallel digraph G with m weighted directed edges, there exists a fully dynamic data structure that uses $O(m)$ space and supports update operations and query operations* MaxFlow *and* ShortestPath *in $O(\log m)$ time and query operations* ReportPath *and* ReportCut *in $O(k \log m)$ time to return k edges.*

4.3.3 Common Neighbor

The *common neighbor problem* consists of answering the following queries on a graph G:

- *CommonNeighbor*(*vertex* v', v'') — Return a common neighbor of v' and v'' in G, or *nil* if no neighbor exists.

Suppose we give each edge unit length. We maintain shortest connectible pair information as in section 4.3.1. Notice that if $\mathit{CommonNeighbor}(v', v'') \neq \mathit{nil}$, then the length of the shortest (v', v'')-path is either 1 or 2.

In order to find $\mathit{CommonNeighbor}(v', v'')$, we keep the following values in path attribute set $\mathcal{P}$ for each path Π.

- $\mathit{numberC}(\Pi)$ — the number of C-nodes on path Π.
- $\mathit{contains2}(\Pi)$ — **true** if Π contains a node ν with a child μ not on Π such that $\mathit{shortest}(\mu) = 2$.

It is clear that these values can be maintained as part of a tree attribute system.

We implement $\mathit{CommonNeighbor}(v', v'')$ as follows. Let μ' be $\mathit{Proper}(v')$, μ'' be $\mathit{Proper}(v'')$ and μ their least common ancestor. Let Π' and Π'' be the paths from μ' to μ and from μ'' to μ. Vertices v' and v'' have a common neighbor only if they are in the same block or adjacent blocks, since otherwise the shortest path between them is of length greater than 2. Therefore, if there is more than one total C-node encountered on the paths from μ_u to λ and μ_v to λ, then return *nil*.

Suppose there is exactly one C-node κ total on Π' and Π''. Node κ will be the parent of either $\mathit{GetTrain}(\mu')$ or $\mathit{GetTrain}(\mu'')$. Let v be the vertex represented by κ. Then v' and v'' have a common neighbor if and only if there are edges in G between vertices v and v' and between vertices v and v''. If so, return v, else return *nil*. A similar technique is used if v' and v'' are in the same train, but not the same block.

The final case is where v' and v'' are in the same block. We consider two subcases, when the pair (v', v'') is tw2-connectible, and when the pair (v', v'') is not tw2-connectible. Let node ν represent the common block. First, suppose (v', v'') is tw2-connectible. We cut node ν from its parent. We then let node ν' be $\mathit{MakeTerminals}(\nu, u, v)$. If $\mathit{ShortestPath}(\nu') > 2$ then return *nil*. Otherwise, call $\mathit{Exposegraph}(\nu')$ so that the left edge-path $\Pi_\ell(\nu')$ of ν' has all the children of ν. We use the following path selection function to determine if v' and v'' share a common neighbor:

Selection Function S_{12} *Suppose η is the root of a path tree with children η' and η''. If* contains2(η') = **true** *then return η', else return η''.*

We then find a common neighbor as follows. If $\mathit{contains2}(\Pi_\ell(\nu'))$ = **false** then return *nil*. Otherwise, let node ν'' be the child of ν' associated with $\mathit{PathFind}(\Pi_\ell(\nu'), S_{12})$. Return the second terminal of the left child of ν''.

Finally, suppose the pair (v', v'') is not tw2-connectible. Let nodes ν' and ν'' be the P-node parent of μ' and μ''. Let (s', t') and (s'', t'') be the terminal pairs of $M_{\nu'}$ and

$M_{\nu''}$. At least one of (s', t') and (s'', t'') is a pair of vertices which separate v' and v''. Therefore, if v' and v'' have a common neighbor, then it will be one of vertices s', t', s'', or t''. We can determine this by checking for the existence of a constant number of edges.

Theorem 4.8 *Given a tree-width two graph G with m weighted edges, there exists a dynamic data structure that uses $O(m)$ space and supports query operation* CommonNeighbor *in $O(\log m)$ time and update operations in $O(\log^2 m)$ time.*

If we restrict G to be a series-parallel graph, then we perform update operations in $O(\log m)$ time.

4.3.4 Minimum (Maximum) Spanning Tree

Given a graph G with weighted edges, a spanning tree U of G is a subgraph of G that is a tree and contains every vertex of G. A minimum spanning tree is a spanning tree of minimum weight. [39] show a fully dynamic algorithm to maintain a minimum (or maximum) spanning tree for embedded planar graphs. Edge insertions are supported only across faces of a given embedding. In this section we extend their technique to support edge insertions between any tw2-connectible pair of vertices of tree-width two graphs, regardless of choice of embedding.

Definition 4.11 Consider a planar graph $G = (V, E)$ with an embedding $\mathcal{E}$. Let F be the set of faces of G under embedding $\mathcal{E}$. We define the *dual-graph* $G^* = (F, E^*)$. There is a dual-vertex f in G^* for every face in F. Dual-vertices f_1 and f_2 are connected by *dual-edge* e^* if edge e is adjacent to both faces f_1 and f_2.

We use the following lemmas from [38]:

Lemma 4.13 [38] *Suppose U is a spanning tree of planar graph G. Then the set of dual-edges U^* defined by $\{e^* | e \text{ not in } S\}$ form a spanning tree of G^*.*

Lemma 4.14 [38] *U is a minimum spanning tree of planar graph G if and only if U^* is a maximum spanning tree of dual-graph G^*.*

Consider the graph $\hat{G}$ formed by the following process. Let G be a series-parallel graph represented by TTSP G^τ with associated SPQ-tree T. Suppose G is embedded such that the terminals of G^τ are adjacent to the external face. Form series-parallel graph G' by adding an edge between the terminals of G^τ. Graph $\hat{G}$ is the dual-graph of graph G'. It is easy to see that graph $\hat{G}$ is a series-parallel graph, and a that a representation of $\hat{G}$ is SPQ-tree $\hat{T}$, where $\hat{T}$ is constructed by making S-nodes into P-nodes, and P-nodes into S-nodes in tree T.

Dual graph $\hat{G}$ has two dual-vertices f_1 and f_2 representing the external face of dual graph G^*. Vertices f_1 and f_2 will be the terminals of the root of $\hat{T}$. In order that a spanning tree of $\hat{G}$ contains the same edges as a spanning tree of G^*, we maintain an additional edge $\hat{e}$ between dual vertices f_1 and f_2. Dual-edge $\hat{e}$ has zero-weight and is always assumed to be in a spanning tree of $\hat{G}$.

Suppose U is a spanning tree of series-parallel graph G, represented by SPQ-tree T. We mark each Q-node associated with edges of U. An internal node μ of T is considered marked if there is a path of marked edges between the terminals of G_μ. That is, if μ is a P-node, μ is marked if and only if at least one of its children is marked. If μ is an S-node, then μ is marked if and only if all of its children are marked.

Lemma 4.15 *Suppose G is a series-parallel graph represented by SPQ-tree T. Suppose E' is a subset of edges of G. Subset E' contains a cycle if and only if marking only the Q-nodes associated with edges in E', results in some P-node μ of T having two or more marked children.*

Proof:

(If:) If μ is a P-node with two marked children, then there are two paths connecting the terminals of μ that only meet at the terminals, and hence a cycle.

(Only If:) Suppose E' are the edges of a simple cycle of G. Let node μ be the least common ancestor in T of the Q-nodes associated with edges in E'. If μ is an S-node, there can only be one distinct path between the terminals of μ, since G_μ contains a cutvertex. Therefore, E' does not form a cycle. If μ is a P-node, then there must be two distinct paths between the terminals of μ. Since each child of μ is either a Q-node or an S-node, each child can contribute at most one path between the terminals of μ. Hence, there are two marked children of μ.

□

Now, suppose G is a tree-width two graph represented by SPQC-tree T. Consider a train M of G. Let T_M be the sub-tree of T representing M. We associate with M the the dual SPQ-tree $\hat{T}_M$. It is easy to see that we can maintain the structure of the dual SPQ-trees associated with SPQC-tree T in the same time complexity as maintaining tree T under update operations. The rest of this section is concerned with maintaining the marked nodes.

Changing Edge Weights

Consider the following restricted version of operation *LocalUpdate*:

- *ChangeWeight*(*edge* e, *value* δ) — Reset $weight(e)$ to be $weight(e) + \delta$

In this section we present the algorithm to implement this operation. Suppose G is a tree-width two graph and M is the train containing edge e. Let T_M be the SPQ-tree representing M and $\hat{T}_M$ be the SPQ-tree representing $\hat{M}$. Let S be the minimum spanning tree of M and S^* be the maximum spanning tree of M^*. If edge e is in S and $\delta < 0$, or if e^* is in S^* and $\delta > 0$, then the edges of S and S^* remain fixed. Next consider the case where edge e^* is in S^* and $\delta < 0$. It is well known (e.g. [48]) that for edge e to move to minimum spanning tree S, the weight of e must be less than the weight of the maximum weight edge e' on the cycle induced by adding e to S. If $weight(e) < weight(e')$ then edge e' is called the *replacement edge* and is moved to S^*,

while edge e is moved to S. The case where e is an edge of S and $\delta > 0$ is handled similarly, exchanging the roles of S and S^*.

We use the following fact to find replacement edge e'. Suppose μ is a P-node representing a marked cycle as in Lemma 4.15. If λ is a Q-node representing an edge of the cycle, then every node on the path from λ to μ is marked.

We keep the following tree attribute for each node μ:

- *maxmarked*(μ) — the maximum weight of a leaf λ in the subtree rooted at μ such that every node on the path from λ to μ is marked.

We keep the following path attributes on each path of T:

- *markedP*(Π) — True if path Π contains a P-node that has a marked child not on Π.
- *maxifmarked*(Π) — The value of *maxmarked*(*tail*(Π)) if *head*(Π) is marked.
- *maxifnotmarked*(Π) — The value of *maxmarked*(*tail*(Π)) if *head*(Π) is not marked.
- *allmarked*(Π) — **true** if all the nodes of Π are marked if *head*(Π) is marked.

It is easy to see that these values can be maintained in a tree attribute system.
We use the following subfunctions to find a replacement edge.

- *findcycle*(*node*λ) — Find the P-node ancestor that represents the cycle formed by adding edge e represented by node λ to minimum spanning tree$S$.
- *findreplacement*(*node*μ) — Return the maximum weight edge on the cycle represented by P-node μ.

Operation *findcycle* is implemented using the following path selection function.

Selection Function S_{13} *Suppose η is the root of a path tree with children η' and η''. If* markedP(η') = **true** *then return η', else return η''.*

Operation *findcycle*(λ) is then implemented as *PathFind*(λ, S_{13}).

Operation *findreplacement* is implemented using the following tree selection function.

Selection Function S_{14} *Suppose η is the root of a path tree with children η' and η''. Let Π''' be the path consisting of the concatenation of* tail(η') *with the path represented by η''. If* allmarked(Π''') = **true**, tail(η')) *is marked, and* maxmarked(tail(η')) = maxmarked(tail(η'')) *then return η', else return η''.*

Operation *findreplacement*(λ) is then implemented as *TreeFind*(μ, S_{14}).

We perform operation *ChangeWeight*(e, δ) by first updating *weight*(e) by δ, unmarking the Q-node representing e^* in $\hat{T}_M$, and marking the Q-node λ representing e in T_M. We then let node μ = *findcycle*(λ) and edge e' = $findreplacement(\mu)$. We conclude by unmarking the Q-node representing e' in T_M and making the corresponding node in $\hat{T}_M$. Therefore, operation *ChangeWeight* can be implemented in $O(\log n)$ time.

Modifying the Structure

As already mentioned, we can maintain the structure of the dual trees under update operations. In this section we show how to maintain the marked nodes under operations *MakeGraph*, *DeleteGraph*, *Attach*, *Detach*, *Compose*, *Inject*, *Extract*, *InsertVertex*, *DeleteVertex*, *InsertEdge*, and *DeleteEdge*. Note that the new edges added by operations *InsertEdge*, *InsertVertex*, and *DeleteVertex* are all assumed to have zero-weight.

Operation *MakeGraph* creates a graph consisting of a single edge e, which is marked.

Since each train maintains its own minimum spanning tree and maximum spanning tree there is no change to marked nodes for operations *Attach* and *Detach*. In the same manner there is no change for serial composition. However, suppose we perform operation $Compose(\rho', \rho'', P)$. Both nodes ρ' and ρ'' will be marked since they are the roots of their respective trees. Therefore, we need to find a replacement edge to move to the dual spanning tree. Let node ρ be the root of the resultant tree. The replacement edge is found by calling $findreplacement(\rho)$.

Consider operation $InsertVertex(v, e, e', e'')$. Sine edges e' and e'' will have zero-weight. We first reduce the weight of e to zero. If e is marked then we mark both edges e' and e''. If e is not marked then we do not mark both edges e' and e''. We perform operation $DeleteVertex(\rho, v, e, e', e'')$ similarly. We first reduce the weight of edges e' and e'' to zero. If both edges e' and e'' are marked, we mark edge e. If at least one of edges e' and e'' are not marked, then we do not mark edge e.

Operation $InsertEdge(v', v'', e)$ add zero weight edge e. After restructuring, we mark e then find a replacement edge as in the implementation of *ChangeWeight*.

Finally, consider operation $DeleteEdge(e)$. If e is in the maximum spanning tree S^*, then we simply delete the edge. Tree S^* will remain connected since the two faces adjacent to e will be merged into a single face. If e is in minimum spanning tree S, then we first call $ChangeWeight(e, \infty)$, so e moves into S^*.

Theorem 4.9 *Given a tree-width two graph G with m weighted edges, there exists a fully dynamic data structure that maintains a minimum or maximum spanning tree of G using $O(m)$ space and supporting update operations in $O(\log^2 m)$ time and query operations on $O(\log m)$ time.*

Chapter 5

Dynamic Graph Drawing

5.1 Introduction

Drawing graphs is an important problem that combines flavors of computational geometry and graph theory. Applications can be found in a variety of areas including circuit layout, network management, software engineering, and graphics. For a survey on graph drawing, see [36]. While this area has recently received increasing attention (see, e.g., [10,30,45,46,62,75,98]), the study of drawing graphs in a dynamic setting has been an open problem. Previous work [77] only considers trees and presents a technique that restructures the drawing of a tree in time proportional to its height, and hence linear in the worst case.

The motivation for investigating dynamic graph drawing algorithms arises when very large graphs need to be visualized in a dynamic environment, where vertices and edges are inserted and deleted, and subgraphs are displayed. Several graph manipulation systems allow the user to interactively modify a graph; hence, techniques that support fast restructuring of the drawing would be very useful. Also, it is important that the dynamic drawing algorithm does not alter drastically the stucture of the drawing after a local modification of the graph. In fact, human interaction requires a "smooth" evolution of the drawing.

In this chapter we present dynamic algorithms for drawing planar graphs under a variety of drawing standards. We consider straight-line, polyline, grid, upward, and visibility drawings together with aesthetic criteria that are important for readability, such as the display of planarity, symmetry, and reachability. Also, we provide techniques that are especially tailored for important subclasses of planar graphs such as trees and series-parallel digraphs. Our dynamic drawing algorithms have the important property of performing "smooth updates" of the drawing.

5.1.1 Definitions

A *drawing* Γ of a graph G maps each vertex of G to a distinct point of the plane and each edge (u, v) of G to a simple Jordan curve with endpoints u and v. We say that Γ is a *straight-line* drawing if each edge is a straight-line segment; Γ is a

polyline drawing if each edge is a polygonal chain; Γ is an *orthogonal* drawing if each edge is a chain of alternating horizontal and vertical segments. A *grid* drawing is such that the vertices and bends along the edges have integer coordinates. *Planar* drawings, where edges do not intersect, are especially important because they improve the readability of the drawing. A *planar embedding* specifies the circular order of the edges around a vertex in a planar drawing. Hence, different drawings may have the same planar embedding. Note that a planar graph may have an exponential number of planar embeddings (see, e.g. [79]). An *upward* drawing of an acyclic digraph has all the edges flowing from bottom to top. Planar upward drawings are attracting increasing theoretical and practical interest [10,25,26,29,35,62,104,117]. A *visibility representation* maps vertices to horizontal segments and edges to vertical segments that intersect only the two corresponding vertex segments.

We assume the existence of a *resolution rule* that implies a finite minimum area for the drawing of a graph. Two typical resolution rules are integer coordinates for the vertices, or a minimum distance δ between any two vertices. When a resolution rule is given, it is meaningful to consider the problem of finding drawings with minimum area. Planar drawings require $\Omega(n^2)$ area in the worst-case [32]. Further results on the area of planar drawings appear in [9,30,46,98].

5.1.2 Model

Here we describe a framework for dynamic graph drawing algorithms. At a first glance, it appears that updating a drawing may require $\Omega(n+m)$ time in the worst case, since we may have to change the coordinates of all vertices and edges. Our approach is to consider graph drawing problems in a "query/update" setting. Namely, we aim at maintaining an *implicit* representation of the drawing of a graph G such that the following operations can be efficiently performed:

- *Drawing queries* that return the drawing of a subgraph S of G consistent with the overall drawing of G. We aim at an output-sensitive time complexity for this operation, i.e., a polynomial in $\log n$ and k, where k is the size of S. Ideally, the time complexity should be $O(k + \log n)$. A special case of this query ($S = \{v\}$) returns the coordinates of a single vertex v.

- *Window queries* that return the portion of the drawing inside a query rectangle.

- *Point-location queries* in the subdivision of the plane induced by the drawing of G. Such queries are defined when the drawing of G is planar.

- *Update operations*, e.g., insertion and deletion of vertices and edges or replacement of an edge by a graph, which modify the (implicit) representation of the drawing accordingly.

There are two types of quality measures in dynamic graph drawing: the "aesthetic" properties of the drawing being maintained, and the space-time complexity of queries and updates. There is an inherent tradeoff between the two. For example, it is very easy

to maintain the drawing of a graph where the vertices are randomly placed on the plane and the edges are drawn as straight-line segments. However, the aesthetic quality of the drawings produced by this simple strategy is typically not satisfactory. On the other hand, if we want to guarantee optimal drawings with respect to some aesthetic criteria, e.g., planarity, symmetry, etc., the update/query operations may require high time complexity. In the following we present techniques with polylogarithmic query/update time that maintain drawings that are optimal with respect to a set of aesthetic criteria.

More formally, a dynamic graph drawing problem consists of

- A class of graphs $\mathcal{G}$ to be drawn.
- A repertory $\mathcal{O}$ of operations to be performed, subdivided into
  - A set $\mathcal{Q}$ of *query operations* (such as drawing, window, and point location queries) that ask questions on the drawing of the current graph.
  - A set $\mathcal{U}$ of *update operations* that modify the current graph and restructure its drawing, such as insertion and deletion of vertices and edges.

  The drawing is modified only by update operations and is not changed by queries.
- A *static drawing predicate* $\mathcal{P}_S$ that expresses "aesthetic" properties to be satisfied by the drawing of the current graph. An example of a static drawing predicate for planar graphs is "The drawing is planar, polyline, grid, with $O(n^2)$ area, and at most $2n + 4$ bends along the edges."
- A *dynamic drawing predicate* $\mathcal{P}_D$ that expresses "similarity" properties to be satisfied by the drawings before and after an update operation. An example of a dynamic drawing predicate for trees is "The drawing of a subtree not affected by the update stays the same up to a translation." Such predicates can be used to guarantee a "smooth" evolution of the drawing.

A solution to a dynamic graph drawing problem is an algorithm that dynamically maintains a drawing of a graph of class $\mathcal{G}$ satisfying predicates $\mathcal{P}_S$ and $\mathcal{P}_D$, under a sequence of operations of repertory $\mathcal{O}$. Performance measures for the algorithm are the space requirement and the time complexity of the various operations. Typically, there is a tradeoff between the efficiency of the algorithm and the tightness of the requirements expressed by the drawing predicates $\mathcal{P}_S$ and $\mathcal{P}_D$.

5.1.3 Overview

The rest of this chapter is organized as follows. In Section 5.2 we describe a dynamic technique for upward drawings of rooted trees. The data structure uses $O(n)$ space and supports updates and point-location queries in $O(\log n)$ time. Drawing queries take time $O(k + \log n)$ for a subtree, and $O(k \log n)$ for an arbitrary subgraph. Window

queries take time $O(k \log n)$. The drawings follow the usual convention of horizontally aligning vertices of the same level. Symmetries and isomorphisms of subtrees are displayed, and the area is $O(n^2)$.

In Section 5.3 we present an algorithm for dynamically drawing series-parallel digraphs. It uses $O(n)$ space and supports updates in $O(\log n)$ time. Drawing queries take time $O(k + \log n)$ for a series-parallel subgraph, and $O(k \log n)$ for an arbitrary subgraph. Point location queries take $O(\log n)$ time. Window queries take $O(k \log^2 n)$ time. The algorithm constructs upward straight-line drawings with $O(n^2)$ area, which is optimal in the worst case.

In Section 5.4 we present a family of algorithms that maintain various types of drawings for planar *st*-digraphs, including polyline upward drawings, and visibility representations. The drawings occupy $O(n^2)$ area, which is optimal in the worst case. All of the algorithms use $O(n)$ space and support updates in $O(\log n)$ time. Also, we consider (undirected) biconnected planar graphs. We present semi-dynamic algorithms for maintaining polyline drawings and visibility representations. The data structure uses $O(n)$ space and supports insertions in $O(\log n)$ amortized time (worst-case for insertions that preserve the embedding). Drawing queries take $O(k \log n)$ time.

5.2 Dynamic Tree Drawing

In this section, we investigate the dynamic drawing of a rooted ordered tree T. Assume that edges are directed from the child to the parent, and denote with T_μ the subtree rooted at μ.

5.2.1 $\square$-drawings

We consider the following static drawing predicate $\mathcal{P}_S$:

Upward: The drawing is upward.

Planar: The drawing is planar.

Grid: Vertices are placed at integer coordinates.

Straight-Line: Edges are drawn as straight line segments.

Layered: Nodes of the same depth (distance from the root) are drawn on the same horizontal line.

Order-Preserving: The left-to-right ordering of the children of a node is preserved in the drawing.

Centered: A node μ is "centered" over its children $\mu_1, \cdots, \mu_k$. Examples of variations of this rule are:

- $x(\mu) = \frac{1}{k} \cdot \sum_{i=1}^{k} x(\mu_i)$;

- $x(\mu) = \frac{1}{2} \cdot (x(\mu_1) + x(\mu_k))$.

Isomorphic: Isomorphic subtrees have drawings that are congruent up to a translation.

Symmetric: Symmetric subtrees have drawings that are congruent up to a translation and a reflection.

Quadratic-Area: The drawing has $O(n^2)$ area.

Reingold and Tilford [90] argue that drawings satisfying $\mathcal{P}_S$ are aesthetically pleasing and show how to construct them in $O(n)$ time. We give a fully dynamic algorithm for constructing such drawings. However, in general the drawings produced by the algorithm of [90] are less wide than those produced by our algorithm. Note that finding drawings of minimum width that satisfy the above properties is NP-hard [105].

The $\Box$-drawing of T and the *bounding box* $\Box(\mu)$ of a node μ of T are recursively defined as follows (see Fig. 5.1):

- μ is a leaf: $\Box(\mu)$ is a 2×1 rectangle.
- μ has children $\mu_1, \ldots, \mu_k$: The width of $\Box(\mu)$ is the sum of the widths of $\Box(\mu_i)$, $1 \leq i \leq k$. The height of $\Box(\mu)$ is one plus the maximum of the heights of $\Box(\mu_i)$, $1 \leq i \leq k$. The bounding boxes of the children of μ are placed inside $\Box(\mu)$ such that they do not overlap, their top sides are placed one unit below the top side of $\Box(\mu)$, and their left-to-right order preserves the ordering of the tree.
- If μ is a leaf, it is drawn in the middle of the top side of $\Box(\mu)$. Else, μ is drawn along the top side of $\Box(\mu)$ according to the centering rule of predicate $\mathcal{P}_S$. We call *reference point* of $\Box(\mu)$ the top left corner of $\Box(\mu)$.

To fully specify the $\Box$-drawing, we assume that the reference point of the bounding box of the root of T is placed at $(0, 0)$.

Lemma 5.1 *Given an n-node tree T, the $\Box$-drawing of T satisfies the static drawing predicate $\mathcal{P}_S$ and can be constructed in $O(n)$ time.*

Proof: It is immediate to verify that $\Box$-drawings satisfy $\mathcal{P}_S$. Concerning the area, let g be the number of leaves of T, and let h be the height of T. The area of the drawing is $g \cdot (h + 1)$, which is $O(n^2)$. To construct the $\Box$-drawing of T we use two traversals. The first traversal computes in post-order the sizes of the bounding boxes of the subtrees of T. The second traversal computes in pre-order the positions of the vertices of T. Each traversal can be performed in linear time. $\Box$

5.2.2 Dynamic Environment

We consider a fully dynamic environment for the maintenance of $\Box$-drawings on a collection of trees. Namely, we introduce the following set $\mathcal{O} = \mathcal{Q} \cup \mathcal{U}$ of operations:

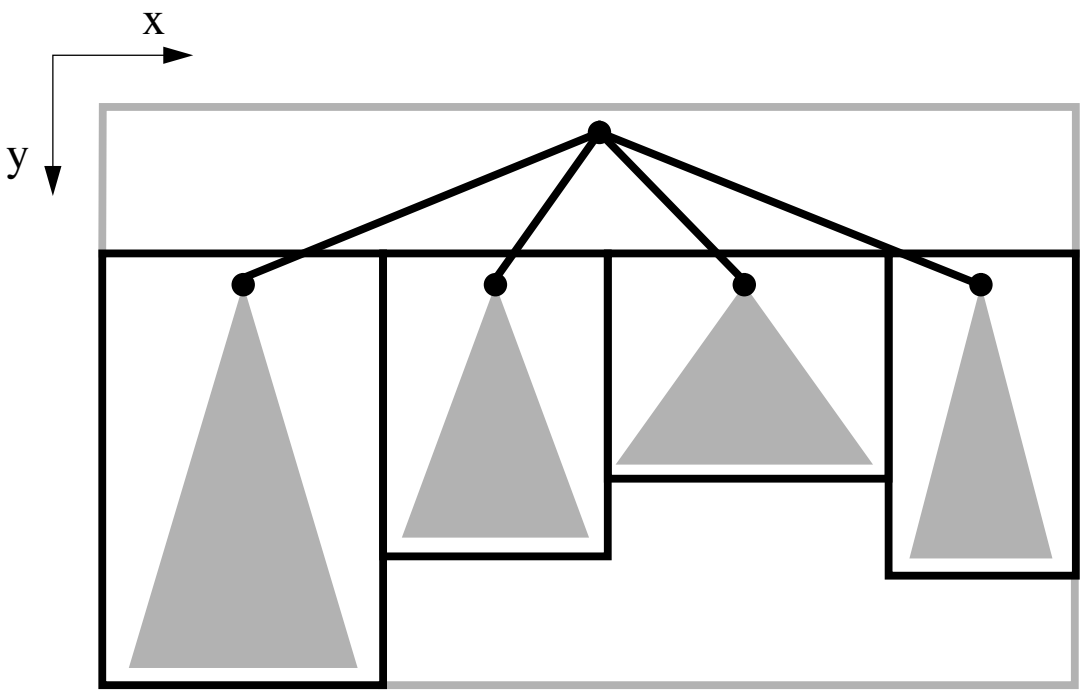

Figure 5.1: Geometric constructions in the $\square$-algorithm

- Query operations ($\mathcal{Q}$):
  - *Draw*(*node* ν) — Return the (x, y) position of node ν.
  - *Offset*(*node* ν) — Return the (x, y) position of the reference point of $\square(\nu)$.
  - *DrawSubtree*(*node* ν) — Return the subdrawing of the subtree rooted at node ν.
  - *Window*(*node* ν, *point* p, q) — Draw the portion of subtree T_ν contained in the query window defined by lower-left corner p and upper-right corner q.
- Update operations ($\mathcal{U}$):
  - *MakeGraph*(*node* λ) — Create a new elementary tree T, consisting of a single node.
  - *DeleteGraph*(*node* λ) — Remove the elementary tree consisting of the single node λ.
  - *Link*(*node* ρ, ν, μ_1, μ_2) — Let ρ be the root of a tree, and let ν be a node of another tree. Also, let μ_1 and μ_2 be consecutive children of ν. Add an edge from ρ to ν and insert ρ between μ_1 and μ_2. If μ_1 and μ_2 are not given, then ρ becomes the only child of ν. If μ_1 (μ_2) is not given then ρ is inserted as the first (last) child of ν.

- *Cut*(*node* μ) — This operation assumes that μ is not the root of a tree. Remove the edge from μ to its parent, thus separating the subtree rooted at μ.
- *Evert*(*node* μ) — Change the parent/child relationship for all nodes on the path from μ to the root of its tree, making μ the root. This operation maintains the clockwise order of neighbors of every node of T.
- *Reflect*(*node* μ) — Reflect the subtree T_μ, i.e., reverse the order of the children of all the nodes of T_μ.
- *Expand*(*node* ν, μ', μ'') — Let $\mu_1, \cdots, \mu_k$ be the children of node ν, and let $\mu' = \mu_i$ and $\mu'' = \mu_j$ $(1 \leq i < j \leq k)$. Replace nodes $\mu_i, \ldots \mu_j$ in the ordering of the children of ν with a new node μ. Node μ has children $\mu_i, \cdots, \mu_j$.
- *Contract*(*node* μ) — This is the inverse operation of *Expand*. It merges node μ with its parent.

In the rest of this section we prove the following theorem:

Theorem 5.1 *Consider the following dynamic graph drawing problem:*

- *Class of graphs $\mathcal{G}$: forest of rooted ordered trees.*
- *Static drawing predicate $\mathcal{P}_S$: upward, planar, grid, straight-line, layered, order-preserving, centered, isomorphic, symmetric, quadratic-area drawing.*
- *Repertory of operations $\mathcal{O}$:* Draw, Offset, DrawSubtree, Window, MakeGraph, DeleteGraph, Link, Cut, Evert, Reflect, Expand, *and* Contract.
- *Dynamic drawing predicate $\mathcal{P}_D$: the drawing of a subtree not affected by an update operation changes only by a translation.*

There exists a fully dynamic algorithm for the above problem with the following performance:

- *A tree with n nodes uses $O(n)$ space;*
- *Operations* MakeGraph *and* DeleteGraph *take each $O(1)$ time;*
- *Operation* DrawSubtree *takes $O(\log n + k)$ time to return the position of k nodes and edges;*
- *Operation* Window *takes $O(k \cdot \log n)$ time to return the position of k nodes and edges;*
- *Operations* Draw, Offset, Link, Cut, Evert, Reflect, Expand, *and* Contract *take each $O(\log n)$ time.*

5.2.3 Data Structure

To simplify formulas, in the rest of this section we assume that the y-axis is directed downward. Note that edges are still directed upward from each child to its parent. We use the following centering rule:

- $x(\mu) = \frac{1}{2}(x(\mu_\ell) + x(\mu_r))$,

  where x_ℓ and x_r are the leftmost and rightmost descendants of μ, respectively.

In order to maintain the drawing of a tree T we keep the following values for a node μ:

- $width(\mu)$ — The width of $\square(\mu)$.
- $level(\mu)$ — The level (distance from the root) of μ.
- $reference(\mu)$ — The x-coordinate of the reference point of $\square(\mu)$. (The y-coordinate of the reference point of $\square(\mu)$ is $level(\mu)$.)

Table 5.1 shows the equations to calculate these values. Note that if μ is the root of T, then $reference(\mu) = 0$. From these values we can easily compute the coordinates of a node μ. Namely, the x-coordinate of μ is $reference(\mu) + width(\mu)/2$, and the y-coordinate of μ is $level(\mu)$.

$$\begin{aligned} width(\mu) &= \sum_{i=1}^{d} width(\mu_i) \\ level(\mu_i) &= level(\mu) + 1 \\ reference(\mu_i) &= reference(\mu) + \sum_{j=1}^{i-1} width(\mu_j) \end{aligned}$$

Table 5.1: The equations to calculate the values of *width* for a node μ and the values of *reference* and *level* for the children $\mu_1, \ldots, \mu_d$ of μ.

We show that the calculations of these attributes can be maintained by storing T as a linear attribute grammar (see section 3.4).

Lemma 5.2 *Using the equations in table 5.1, the attributes* width, level, *and* reference *for the* $\square$*-drawing of a tree* T *can be maintained as a linear attribute grammar.*

Proof: Consider a node μ of T. The only synthesized attribute of μ is $width(\mu)$. The inherited attributes are $level(\mu)$ and $reference(\mu)$. By definition 3.19, we need to show that the precedence graph G of T is acyclic and all dependencies are linear.

All dependencies are linear since attributes are only added. Suppose there is a cycle in G. The cycle is either between attributes of nodes all at the same level of T, or different levels of T. Suppose C is a cycle between attributes of nodes all at the same level of T. However, the only relationships between attributes of nodes at the same level of T are in the calculation of *reference*, and that relationship is only from a node to its right sibling, which contradicts C being a cycle. Now, suppose C is a cycle including attributes of nodes at different levels of T. Therefore, there must be an edge in C from an attribute of a node μ to its parent. However, the only synthesized attribute of any node ν is $width(\nu)$, which only is dependent on *width* at the children of ν.

Since T is unbounded, we also must show the expansion invariant at nodes of T. Recall that for a linear attribute grammar, the tree property at each node μ is a summary graph of the dependencies of the synthesized attributes of μ on the inherited attributes of μ. As we already have noticed, there is no dependency of the the synthesized attributes (*width*) of μ on the inherited attributes of μ. Therefore, the expansion invariant holds since the calculation of *width* is associative. □

5.2.4 Dynamic Operations

We keep the following additional values for each solid path Π in T:

- *rightmost*(Π) — **true** if each node of Π other than *tail*(Π) is the rightmost child of its parent.
- *leftmost*(Π) — **true** if each node of Π other than *tail*(Π) is the leftmost child of its parent.

Clearly we can maintain the values of *rightmost* and *leftmost* in a path attribute system on the paths of T.

Update operations are implemented as for general linear attribute grammars (Theorem 3.7). Therefore, we only describe query operations. Operations $Draw(\nu)$ and $Offset(\nu)$ are implemented by first calling $EvaluateAttributes(\nu)$. Operation $Draw(\nu)$ returns the pair $(reference(\nu) + width(\nu)/2, level(\nu))$. Operation $Offset(\nu)$ returns the pair $(reference(\nu), level(\nu))$. Operation $DrawSubgraph(\nu)$ is performed by calling $Offset(\nu)$ in order to determine the position of $\Box(\nu)$. We then use the sequential algorithm to draw T_ν.

Suppose $p = (x_p, y_p)$ and $q = (x_q, y_q)$ are points with $x_p < x_q$ and $y_p < y_q$, and let W be the window defined by p and q. Recall that the y-coordinate of the drawing of a node μ of tree T corresponds to the depth of μ in T.

We use the following fact in our algorithm. If x_ℓ is the x-coordinate of the reference point of the bounding box of some node μ, then the line $x = x_\ell + 0.25$ does not intersect any edge or vertex in the drawing of T_μ.

We keep the following additional value for each solid path Π in T:

- *rightmost*(Π) — **true** if and only if each node of Π other that *tail*(Π) has no right sibling.

- *leftmost*(Π) — **true** if and only if each node of Π other that *tail*(Π) has no left sibling.

Clearly we can maintain the value *rightmost* during join operations.

Operation *Window* is implemented using the following operations.

- *locatepoint*(*node* ν; *point* p) — Returns the node μ of T_ν such that contains p is contained in $\Box(\mu)$, but p is not contained in the bounding rectangle of any of the children of μ. If p is outside $\Box(\nu)$, then *locatepoint*(ν, p) returns *nil*.
- *drawproper*(*node* ν; *point* p, q) — Draw node ν and the edges from ν to its children, clipping to W.
- *findrightint*(*node* ν; *point* p, q) (*findleftint*(*node* ν; *point* p, q)) — Let Π be the path following right (left) children from ν. Return the closest descendant μ of ν on Π such that *drawproper*(μ, p, q) draws (at least) a portion of an edge.

Operation *locatepoint*(ν, p) first calls *expose*(ν) in order to find the location and width of $\Box(\nu)$. If point p is outside $\Box(\nu)$, then return *nil*. Otherwise, we use the following tree selection function, which takes query point p as an argument.

Selection Function S_{15} *Suppose η is the root of a path tree with children η' and η''. If point p is contained in* $\Box(\text{tail}(\eta'))$ *then return η', else return η''.*

Operation *TreeFind*(ν, S_1, p) returns a node μ such that p is contained in $\Box(\mu)$, but p is not contained in the bounding box for any of the children of μ. We perform operation *locatepoint* by setting node μ = *TreeFind*(ν, S_1, p). If p is on the right boundary of $\Box(\mu)$, then let $x_p = x_p + 0.25$, and repeat. We then return node μ.

We perform operation *drawproper*(μ, p, q) by first calling *expose*(μ). The following path selection function on edge-path $\Pi_\ell(\mu)$ takes as an argument the pair of points (p, q) that define window W.

Selection Function S_{16} *Suppose η is the root of a path tree with children η' and η''. Let μ' be the child of μ associated with* tail(η'). *If the edge from node μ' to node μ intersects W, then return η', else return η''.*

We find the leftmost edge of the proper region of μ intersecting W, by letting node μ_ℓ = *PathFind*$(\Pi_\ell(\mu), S_2, (p, q))$. We find the rightmost intersecting edge connected to node μ_r similarly. We then draw μ, and the edges between μ_ℓ and μ_r, clipping to W.

Operation *findrightint* is performed using the following tree selection function which takes the pair $(\ell, (x, y))$, where ℓ is the x-coordinate of the left side of the query window, and x as an argument, and pair (x, y) is the coordinates of the reference point of the tail of the currently considered path.

Selection Function S_{17} *Suppose η is the root of a path tree with children η' and η''. Construct node η''' representing the path from* tail(η') *to* tail(η''). *If rightmost*(η''') = **false** *or if the x-coordinate of* tail(η') *is greater than x, then return η''. Else return η'.*

Operation *findrightint* is performed using the following tree selection function which takes an argument x_ℓ, the the x-coordinate of the left side of the query window.

Selection Function S_{18} *Suppose η is the root of a path tree with children η' and η''. Construct node η''' representing the path from* tail(η') *to* tail(η''). *If $rightmost(\eta''') =$* **false** *or if the x-coordinate of* tail(η') *is greater than x_ℓ, then return η''. Else return η'.*

When used with operation *TreeFind*, selection function S_{18} returns the first descendant μ' of μ reached through only rightmost children such that there is an edge from μ' which intersects the line $x = x_\ell$. Operation $findrightint(\nu, p, q)$ is then implemented as follows. Let node
$\mu = TreeFind(\nu, S_{18}, x_p)$. If μ is a leaf then return *nil*. Otherwise, let node μ' be the rightmost child of μ. If the edge from node μ' to node μ intersects W, return μ. Else return *nil*. Operation $findleftint(\nu, p, q)$ is implemented in a similar manner, substituting the value *leftmost* for *rightmost*.

Operation *Window* is implemented using the following function that is mutually recursive with operation *Window*.

- *drawtree*(*node* ν, *point* p, q) — Draw the portion of subgraph G_ν contained in the query window W defined by lower-left corner p and upper-right corner q. The proper region of ν is assumed to intersect W.

We implement operation $Window(\nu, p, q)$ by performing the following steps.

1. If W does not intersect $\square(\nu)$ then stop.
2. Clip W to $\square(\nu)$, update points p and q accordingly.
3. Initialize scan point $s = (x_s, p_s)$ to be point p. Let node $\mu = locatepoint(\nu, s)$. We call $drawtree(\mu, s, q)$.
4. Move scan point s by resetting x_s to be 0.25 plus the x-coordinate of the right edge of $\square(\mu)$. Repeat steps 3 and 4 until s is no longer contained in $\square(\nu)$ or W.

Operation $drawtree(\nu, p, q)$ is implemented by performing the following steps.

1. Perform $drawproper(\nu, p, q)$.
2. If node ν is a leaf, or if $y_q < depth(\nu) + 1$ then stop.
3. If step 1 did not draw any edges, then suppose $Draw(\nu)$ is to the left (right) of W. Let $\mu = findrightint(\nu, p, q)$ ($\mu = findleftint(\nu, p, q)$). If $\mu = nil$ then stop. Otherwise call $drawtree(\mu, p, q)$
4. If step 3 did draw edges then let $y_p = depth(\nu) + 1$ and call $Window(\mu, p, q)$.

Each step is implemented in $O(\log n)$ time for each edge and vertex drawn. After drawing an edge, we take $O(\log n)$ time to determine if we stop. Therefore, to draw k nodes and edges takes $O(k \cdot \log n)$ time.

Operation *findrightint*(ν, p, q) is then implemented as follows. Let node $\mu = TreeFind(\nu, S_3, x_p)$. If μ is a leaf then return *nil*. Otherwise, let node μ' be the rightmost child of μ. If the edge from node μ' to node μ intersects W, return μ. Else return *nil*.

Operation *findleftint*(ν, p, q) is implemented in a similar manner, substituting the value *leftmost* for *rightmost*.

Operation *Window* is implemented using the following function that is mutually recursive with itself.

- *drawtree*(*node* ν, *point* p, q) — Draw the portion of subgraph G_ν contained in the query window W defined by lower-left corner p and upper-right corner q. The proper region of ν is assumed to intersect W.

We implement operation $Window(\nu, p, q)$ by performing the following steps.

1. If W does not intersect $\square(\nu)$ then stop.
2. Clip W to $\square(\nu)$, update points p and q accordingly.
3. Initialize scan point $s = (x_s, p_s)$ to be point p. Let node $\mu = locatepoint(\nu, s)$. We call $drawtree(\mu, s, q)$.
4. Move scan point s by resetting x_s to be 0.25 plus the x-coordinate of the right edge of $\square(\mu)$. Repeat steps 3 and 4 until s is no longer contained in $\square(\nu)$ or W.

Operation $drawtree(\nu, p, q)$ is implemented by performing the following steps.

1. Perform $drawproper(\nu, p, q)$.
2. If node ν is a leaf, or if $y_q < depth(\nu) + 1$ then stop.
3. If step 1 did not draw any edges, then suppose $Draw(\nu)$ is to the left (right) of W. Let $\mu = findrightint(\nu, p, q)$ ($\mu = findleftint(\nu, p, q)$). If $\mu = nil$ then stop. Otherwise call $drawtree(\mu, p, q)$
4. If step 3 did draw edges then let $y_p = depth(\nu) + 1$ and call $Window(\mu, p, q)$.

Each step is implemented in $O(\log n)$ time for each edge and vertex drawn. After drawing an edge, we take $O(\log n)$ time to determine if we stop. Therefore, to draw k nodes and edges takes $O(k \cdot \log n)$ time.

5.3 Series Parallel Digraphs

In this section we investigate the maintenance of upward drawings of series-parallel digraphs. For definitions, and description of the algorithm to maintain a tree attribute system kept on the SPQ-trees associated with a collection of series-parallel digraphs, see sections 4.2.4, 4.2.6, and 4.3.2.

Suppose G is a series-parallel digraph, and T is its associated SPQ-tree. Let μ be a node of T and $\mu_1, \cdots, \mu_k$ be the children of μ. A *closed component* of G is either G or the union of the pertinent digraphs of a subsequence $\mu_i, \cdots, \mu_j$, where $1 < i \leq j < k$ and μ is an S-node. An *open component* of G is the union of the pertinent digraphs of a subsequence $\mu_i, \cdots, \mu_j$, minus its poles, where $1 \leq i \leq j \leq k$. A *component* is either an open or a closed component.

5.3.1 Δ-drawings

We consider the following static drawing predicate $\mathcal{P}_S$:

Upward: The drawing is upward.

Planar: The drawing is planar.

Grid: Vertices are placed at integer coordinates.

Straight-Line: Edges are drawn as straight line segments.

Quasi-Embedding-Preserving: The drawing preserves the embedding, except, possibly, for the transitive edges.

Isomorphic: Isomorphic components have drawings that are congruent up to a translation.

Vertically-Symmetric: The drawings of a series-parallel digraph and its reverse have drawings that are congruent up to a translation and a reflection.

Quadratic-Area: The drawing has $O(n^2)$ area.

It is important to note that in order to get polynomial area the embedding cannot be completely preserved. Namely, it is shown in [9] that there exists a class of embedded series-parallel digraphs for which any upward straight-line drawing that preserves the embedding requires exponential area under any resolution rule.

Δ-drawings of series-parallel digraphs are introduced in [9] and satisfy the above static drawing predicate. In Δ-drawings the embedding is modified so that all the transitive edges are embedded on one side, say, the right side. We call such embedding *right-pushed*. The Δ-drawing Γ of a series-parallel digraph G is inductively defined inside a *bounding triangle* $\Delta(\Gamma)$ that is isosceles and right-angled. The hypotenuse of $\Delta(\Gamma)$, from now on called the *right side* of $\Delta(\Gamma)$, is a vertical segment, and the other two sides are on its left. The height of $\Delta(\Gamma)$ is the length of the right side; the width

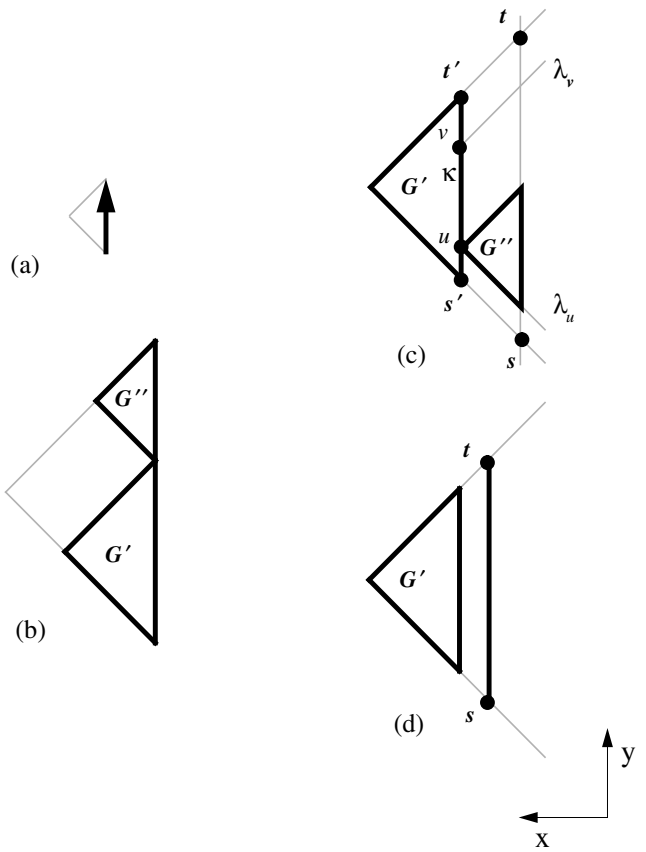

Figure 5.2: Geometric construction of a Δ-drawing: (a) base case; (b) series composition; (c) parallel composition (general case); (d) parallel composition with the "right-pushed" transitive edge.

of $\Delta(\Gamma)$ one half of the height. In a series composition, the subdrawings are placed one above the other. In a parallel composition, the subdrawings are placed one to the right of the other and are deformed in order to identify the poles, guaranteeing that their edges do not cross. The algorithm is outlined below. More details can be found in [9].

- Modify the embedding of G into a right-pushed embedding.
- If G consists of a single edge, it is drawn as a vertical segment of height 2, with bounding triangle having width 1 (see Fig. 5.2.a).
- If G is the series composition of G' and G'', the drawings Γ' and Γ'' of G' and G'' are recursively constructed and combined by translating Γ'' so that its source is identified with the sink of G' (see Fig. 5.2.b). The bounding triangle $\Delta(\Gamma)$ is obtained by extending the bottom side of $\Delta(\Gamma')$ and the top side of $\Delta(\Gamma'')$.

- If G is the parallel composition of G' and G'', the drawings Γ' and Γ'' of G' and G'' are recursively constructed. We consider the the rightmost outgoing edge (s', u) of the source s' of G' and the rightmost incoming edge (v, t') of the sink t' of G' (see Fig. 5.2.c and d). Let λ_u be the line through u that is parallel to the bottom side of $\Delta(\Gamma')$, and λ_v be the line through v that is parallel to the top side of $\Delta(\Gamma')$. Also, let κ be the vertical line extending the right side of $\Delta(\Gamma')$. We call *prescribed region* of Γ'' the (infinite) region to the right of κ, λ_u, and λ_v. First, we translate Γ'' anywhere inside its prescribed region. Then we identify the sources and sinks of G' and G'' by moving them to the intersections s and t of the right side of Γ'' with the lines extending the bottom and top sides of Γ'.

If the series or the parallel compositions involve more than two graphs (say ℓ graphs), the above steps are applied $\ell - 1$ times.

In a Δ-drawing Γ the source (resp. sink) of G is placed at the bottom (resp. top) vertex of $\Delta(\Gamma)$, and the other vertex of $\Delta(\Gamma)$ is not occupied by any vertex of G. Also, the rightmost outgoing edge of the source and the rightmost incoming edge of the sink lie on the right side of $\Delta(\Gamma)$.

We obtain an $O(n^2)$-area Δ-drawing by specializing the placement of Γ'' in a parallel composition so that $\Delta(\Gamma'')$ touches $\Delta(\Gamma')$. Let p be the point on the right side of $\Delta(\Gamma')$ and half-way between u and v. We translate Γ'' so that the left vertex of $\Delta(\Gamma'')$ coincides with p. In this way, Γ'' is in its prescribed region.

Notice that in both series and parallel compositions the height of $\Delta(\Gamma)$ is equal to the sum of the heights of $\Delta(\Gamma')$ and $\Delta(\Gamma'')$. Hence, the height of $\Delta(\Gamma)$ is exactly $2m$, and the area of the drawing Γ is m^2.

Lemma 5.3 [9] *Given an n-node series-parallel digraph G, the Δ-drawing of G satisfies the static drawing predicate $\mathcal{P}_S$, and can be constructed in $O(n)$ time.*

5.3.2 Dynamic Environment

We create an addition node type, the *P_Q-node*, to handle right-pushed embeddings. A P_Q-node represents the following case. If G is the parallel composition of series-parallel digraphs G_1 and G_2 with SPQ-trees T_1 and T_2 with roots ρ_1 and ρ_2, and G_2 is a single edge, then T consists of a P_Q-node root with children ρ_1 and ρ_2. We extend the type invariant (page 50) such that a P-node cannot have P_Q-node as a child.

For a node μ of SPQ-tree T, we define $\Delta(\mu)$ the *bounding triangle* to be the triangle enclosing the drawing of G_μ without considering the translation of the poles of G_μ performed at ancestors of μ in T. The *reference point* of $\Delta(\mu)$ is the intersection of the right and bottom sides.

We consider a fully dynamic environment for the maintenance of Δ-drawings on a collection $\mathcal{G}$ of series-parallel digraphs. Namely, we introduce the following set $\mathcal{O} = \mathcal{Q} \cup \mathcal{U}$ of operations:

- Query operations ($\mathcal{Q}$):

- *Draw*(*vertex* v) — Return the (x, y) position of the drawing of vertex v of series-parallel digraph G. The reference point of the drawing of G is considered to be at $(0, 0)$.
- *Offset*(*node* ν) — Return the (x, y) position of the reference point of $\Delta(\nu)$, where ν is a node in the SPQ-tree representing series-parallel digraph G. The reference point of the drawing of G is considered to be at $(0, 0)$.
- *DrawSubgraph*(*node* ν) — Draw the subgraph G_ν as it appears in the drawing of G.
- *Window*(*node* ν, *point* p, q) — Draw the portion of subgraph G_ν contained in the query window defined by lower-left corner p and upper-right corner q.
- *Locate*(*node* ν, *point* p) — Returns the vertex, edge, or face of the digraph G_ν containing point p.

- Update operations ($\mathcal{U}$):
  - *MakeDigraph* — Create a new elementary series-parallel digraph G, represented by a single Q-node, and add G to $\mathcal{G}$.
  - *DeleteDigraph*(*node* λ) — Remove from $\mathcal{G}$ the elementary series-parallel digraph represented by the single Q-node λ.
  - *Compose*(*nodetype* X; *node* ρ', ρ'') — Perform a composition on the series-parallel digraphs $G_{\rho'}$ and $G_{\rho''}$. The composition is series or parallel according to whether $X = S$ or $X = P$. The resulting series-parallel digraph is added to $\mathcal{G}$ while $G_{\rho'}$ and $G'_{\rho'}$ are removed from $\mathcal{G}$.
  - *Attach*(*node* ρ, λ) — Replace the edge represented by Q-node λ with the series-parallel digraph G_ρ. The resulting series-parallel digraph is added to $\mathcal{G}$ while G_ρ is removed from $\mathcal{G}$.
  - *Detach*(*node* μ) — Remove the pertinent digraph G_μ of node μ from G and replace it with a single edge. The series-parallel digraph G_μ is added to $\mathcal{G}$.
  - *InsertEdge*(*vertex* v', v''; *edge* e) — Insert a new edge e from v' to v'' . The operation is performed only if the resulting digraph is a series-parallel digraph.
  - *DeleteEdge*(*edge* e) — Delete edge e. The operation is performed only if the resulting digraph is a series-parallel digraph.
  - *InsertVertex*(*vertex* v; *edge* e, e', e'') — Replace edge e with two edges e' and e'' by inserting vertex v.
  - *DeleteVertex*(*node* ρ, *vertex* v; *edge* e, e', e'') — Replace vertex v and its incident edges e' (incoming) and e'' (outgoing) with a single edge e. The operation is performed only if e' and e'' are the only incident edges of v.

In the rest of this section we prove the following theorem:

Theorem 5.2 *Consider the following dynamic graph drawing problem:*

- *Class of graphs $\mathcal{G}$: embedded series-parallel digraphs.*
- *Static drawing predicate $\mathcal{P}_S$: upward, planar, grid, Straight-line, quasi-embedding-preserving, isomorphic, vertically-symmetric, quadratic-area drawing.*
- *Repertory of operations $\mathcal{O}$:* Draw, Offset, DrawSubgraph, Window, Locate, MakeDigraph, DeleteDigraph, Compose, Attach, Detach, InsertEdge, DeleteEdge, InsertVertex, *and* DeleteVertex.
- *Dynamic drawing predicate $\mathcal{P}_D$: the drawing of a component not affected by an update operation changes only by a translation.*

There exists a fully dynamic algorithm for the above problem with the following performance:

- *a series-parallel digraph uses $O(n)$ space;*
- *Operations* MakeDigraph *and* DeleteDigraph *take each $O(1)$ time;*
- *Operation* DrawSubgraph *takes $O(\log n + k)$ time to return the position of k nodes and edges;*
- *Operation* Window *takes $O(k \cdot \log^2 n)$ time to return the position of k nodes and edges;*
- *Operations* Draw, Offset, Compose, Attach, Detach, InsertEdge, DeleteEdge, InsertVertex, DeleteVertex, *and* Locate *take each $O(\log n)$ time.*

5.3.3 Data Structure

To simplify formulas, in the rest of this section we assume that the x-axis is directed from right to left. If Z is a (x, y)-pair, then $x(Z)$ returns the x-value and $y(Z)$ returns the y-value. In order to maintain the drawing of a series-parallel digraph G represented by SPQ-tree T, we keep the following values for a node μ:

- *width*(μ) — The width of $\Delta(\mu)$. Note that the height of $\Delta(\mu)$ will be $2 \cdot$ *width*(μ).
- *position*(μ) — The offset of the position of the reference point of $\Delta(\mu)$ from the position of the reference point of $\Delta(\Gamma)$.

The following values are relative positions in $\Delta(\mu)$ considering the reference point of $\Delta(\mu)$ to be at $(0, 0)$:

- *sourceright*(μ) — The location of the drawing of the vertex connected to the rightmost outgoing edge from the source of G_μ.
- *sourceleft*(μ) — The location of the drawing of the vertex connected to the leftmost outgoing edge from the source of G_μ.

$$
\begin{aligned}
width(\mu) &= \sum_{i=1}^{k} width(\mu_i) \\
x(position(\mu_j)) &= x(position(\mu)) + \sum_{i=j+1}^{k} width(\mu_i) \\
y(position(\mu_j)) &= y(position(\mu)) + \sum_{i=j+1}^{k} width(\mu_i) + \sum_{i=1}^{j-1} y(sourceright(\mu_i)) \\
x(sourceright(\mu)) &= 0 \\
y(sourceright(\mu)) &= \sum_{i=1}^{k} y(sourceright(\mu_i)) \\
x(sourceleft(\mu)) &= \sum_{i=2}^{k} width(\mu_i) + x(sourceleft(\mu_1)) \\
y(sourceleft(\mu)) &= \sum_{i=2}^{k} width(\mu_i) + y(sourceleft(\mu_1)) \\
x(sinkright(\mu)) &= 0 \\
y(sinkright(\mu)) &= \sum_{i=1}^{k-1} y(sourceright(\mu_i)) + y(sinkright(\mu_k)) \\
x(sinkleft(\mu)) &= \sum_{i=2}^{k} width(\mu_i) + x(sinkleft(\mu_1)) \\
y(sinkleft(\mu)) &= \sum_{i=2}^{k} width(\mu_i) + y(sinkleft(\mu_1))
\end{aligned}
$$

Table 5.2: The equations to calculate the values of *width*, *sourceright*, *sourceleft*, *sinkright*, and *sinkleft* for a P-node μ and the value of *position* for the children $\mu_j, 1 \leq j \leq k$ of μ.

- *sinkright*(μ) — The location of the drawing of the vertex connected to the rightmost incoming edge to the sink of G_μ.
- *sinkleft*(μ) — The location of the drawing of the vertex connected to the leftmost incoming edge to the sink of G_μ.

The equations to calculate these values are linear expressions, and are shown in Tables 5.2, 5.3, 5.4, and 5.5. As an example, Fig. 5.3 shows pictorally how to calculate the value of *position* for the bounding triangle of a graph involved in a parallel composition. Note that if μ is the root of T, then $position(\mu) = (0, 0)$.

We show that the calculations of these attributes can be maintained on SPQ-tree T as a linear attribute grammar (see section 3.4).

$$
\begin{aligned}
sourceright(\mu) &= (0, 2 \cdot width(\mu)) \\
sinkright(\mu) &= (0, 0)
\end{aligned}
$$

Table 5.3: The equations to calculate the values of *sourceright* and *sinkright* for a P_Q-node μ. All other values are calculated as for a P-node (see Table 5.2).

$$
\begin{aligned}
width(\mu) &= \sum_{i=1}^{k} width(\mu_i) \\
x(position(\mu_j)) &= x(position(\mu)) \\
y(position(\mu_j)) &= y(position(\mu)) + 2 \cdot \sum_{i=1}^{j-1} width(\mu_i) \\
sourceright(\mu) &= sourceright(\mu_1) \\
sourceleft(\mu) &= sourceleft(\mu_1) \\
x(sinkright(\mu)) &= 0 \\
y(sinkright(\mu)) &= 2 \cdot \sum_{i=1}^{k-1} width(\mu_i) + y(sinkright(\mu_k)) \\
x(sinkleft(\mu)) &= x(sinkleft(\mu_k)) \\
y(sinkleft(\mu)) &= 2 \cdot \sum_{i=1}^{k-1} width(\mu_i) + y(sinkleft(\mu_k))
\end{aligned}
$$

Table 5.4: The equations to calculate the values of *width*, *sourceright*, *sourceleft*, *sinkright*, and *sinkleft* for an S-node μ and the value of *position* for the children $\mu_j, 1 \leq j \leq k$ of μ.

Lemma 5.4 *Using the equations in tables 5.2, 5.4, and 5.5, the attributes* width, position, sourceright, sourceleft, sinkright, *and* sinkleft *for the Δ-drawing of a series-parallel digraph G with associated SPQ-tree T can be maintained as a linear attribute grammar.*

Proof: Consider a μ of T. The synthesized attributes of μ are $width(\mu)$, $sourceright(\mu)$, $sourceleft(\mu)$, $sinkright(\mu)$, and $sinkleft(\mu)$. The only inherited attribute is $position(\mu)$. As in the proof of lemma 5.2, we need to show that the precedence graph G_p of T is acyclic and all dependencies are linear.

All dependencies are linear since attributes are either added or multiplied by a constant. Suppose there is a cycle in G_p. The cycle is either between attributes of nodes all at the same level of T, or different levels of T. Suppose C is a cycle between attributes of nodes all at the same level of T. However, the only relationships between attributes of nodes at the same level of T are in the calculation of *position* among the

$$
\begin{aligned}
\mathit{width}(\mu) &= 1 \\
\mathit{sourceright}(\mu) &= (0,2) \\
\mathit{sourceleft}(\mu) &= (0,2) \\
\mathit{sinkright}(\mu) &= (0,0) \\
\mathit{sinkleft}(\mu) &= (0,0)
\end{aligned}
$$

Table 5.5: The values of *width*, *sourceright*, *sourceleft*, *sinkright*, and *sinkleft* for a Q-node μ.

children of an S or P-node μ. These relationships are only from a node to its right sibling (if μ an S-node), or from a node to its left sibling (if μ an P-node), which contradicts C being a cycle. Now, suppose C is a cycle including attributes of nodes at different levels of T. Therefore, there must be an edge in C from an attribute of a node μ to its parent ν. However, the synthesized attributes of any node ν are only dependent on the synthesized attributes at the children of ν.

Since T is unbounded, we also must show the expansion invariant at nodes of T. Recall that for a linear attribute grammar, the tree property at each node μ is a summary graph of the dependencies of the synthesized attributes of μ on the inherited attributes of μ. As we already have noticed, there is no dependency of the the synthesized attributes of μ on the inherited attributes of μ. Therefore, the expansion invariant holds since the calculations of the synthesized attributes are associative. □

5.3.4 Dynamic Operations

We keep the following additional values for each solid path Π from node σ to node τ.

- *samesource*(Π) — **true** if and only the source of G_σ is the same vertex as the source of G_τ.
- *samesink*(Π) — **true** if and only the sink of G_σ is the same vertex as the sink of G_τ.
- *noleftp*(Π) — **true** if and only if path Π does not contain both a P-node and its leftmost child.

Recall for a vertex v in series-parallel digraph G, recall that operation *Proper*(v) Returns the node triple (ν, μ', μ''), where ν is the proper node of v, node μ' is the child of ν such that v is the first terminal of μ', and μ'' is the child of ν such that v is the second terminal of μ''. If v is a terminal of the root chain, then *Proper*(v) returns $(\rho, -, -)$, where ρ is the root of T. Operation *Proper* runs in $O(\log m)$ time.

Operation *Offset*(ν) is implemented as for trees. We call *EvaluateAttributes*(ν) and return the value of *position*(ν).

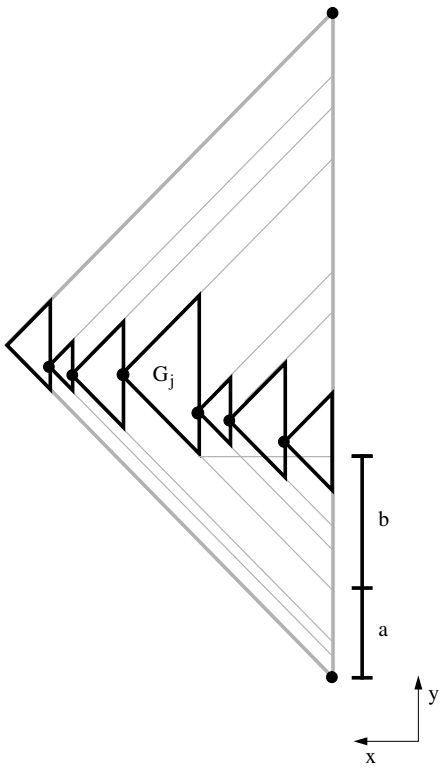

Figure 5.3: A pictoral representation of the calculations of $position(\mu_j)$, where μ_j is a child of a P-node μ. Graph G_j is the pertinent graph of μ_j. Notice that lengths $a = \sum_{i=1}^{j-1} y(sourceright(\mu_i))$ and $b = \sum_{i=j+1}^{k} width(\mu_i)$.

Operation $Draw(v)$ is performed as follows. Find $Proper(v)$. If v is the source of G, then return $(0,0)$. If v is the sink of G then return $(0, 2 \cdot width(\rho))$, where ρ is the root of T. Otherwise, suppose $Proper(v) = (\mu, \mu', \mu'')$. Return $Offset(\mu'')$. This can all be done in $O(\log m)$ time.

Operation $DrawSubgraph(\nu)$ is performed by calling $Draw$ on the poles of G_ν and then calling $Offset(\nu)$ in order to determine the position of $\Delta(\nu)$. We then visit G_ν and compute the positions in $O(1)$ amortized time per vertex and edge using the sequential Δ-algorithm (see Lemma 5.3).

Each face in the drawing Γ of series-parallel digraph G is formed by the parallel composition of two subgraphs. Therefore, we identify each face f of Γ by the node triplet (ν, μ_ℓ, μ_r), where ν is a P-node or a P_Q-node, and μ_ℓ and μ_r are the consecutive children of ν such that f is defined by the parallel composition of G_{μ_ℓ} and G_{μ_r}. Operation $Locate(\nu, p)$ consists of the following steps for a digraph G represented by SPQ-tree T:

1. If point p is not contained in $\Delta(\nu)$, then return the external face.

2. Find the deepest descendant μ of ν such that p is contained in $\Delta(\mu)$, but not the bounding triangle of any of the children of μ.

3. Let s and t be the source and sink of G_μ. If p is at $Draw(s)$ or $Draw(t)$ then return the found vertex. If p is on the rightmost edge from s or the right most edge entering t, then return the edge.

4. If node μ is an S or Q-node, then p is to the left of the drawing of any of the edges of G_μ. Let node ν' be the closest ancestor of μ such that node μ is not a descendant of the leftmost child of ν', let node μ'' be the child of ν' on the path from μ to ν', and let node μ' be the left sibling of μ''. Therefore, face containing p will be (ν', μ', μ'').

5. The final case is if node μ is a P-node or a P_Q-node. Let $R_s(\mu)$ be the region bounded by the triangle formed by the drawing of vertices s, $sourceleft(\mu)$, and $sourceright(\mu)$. Similarly, let $R_t(\mu)$ be the region formed by the drawing of vertices t, $sinkleft(\mu)$, and $sinkright(\mu)$.

   (a) If p is not contained in $R_s(\mu)$ or $R_t(\mu)$, then, as above, p is to the left of the drawing of any of the edges of G_μ, and we continue as in step 2.

   (b) Otherwise, suppose p is in $R_s(\mu)$ (the case where p is in $R_t(\mu)$ is handled similarly). Let node ν', be the deepest descendant of μ such that point p is contained in $R_s(\nu')$. Node ν' will be a P-node, since for S-node κ with left-child κ', $R_s(\kappa) = R_s(\kappa')$. If p is on an edge of $R_s(\mu)$, then return the edge. Otherwise, the face containing p is (ν', μ', μ''), where μ' is the child of ν' such that point p is to the right of $R_s(\mu')$ but to the left of $R_s(\mu'')$ where node μ'' is the immediate left sibling of μ' (see Fig. 5.4).

Steps 2, 3, and 5b are implemented using selection functions. Consider the following tree selection function, which takes query point p as an argument.

Selection Function S_{19} *Suppose η is the root of a path tree with children η' and η''. If point p is contained in $\Delta(\text{tail}(\eta'))$ then return η', else return η''.*

We then perform step 2 by letting node $\kappa = TreeFind(\nu, S_{19}, p)$. If κ is a node on an edge path of node κ', then return κ'. Otherwise, return node κ.

Consider the following path selection function.

Selection Function S_{20} *Suppose η is the root of a path tree with children η' and η''. If* noleftp(η') *is* **true**, *or if* head(η'') *is a P-node and* tail(η') *is not its leftmost child, then return η', else return η''.*

Step 4 is implemented by first calling operation *expose* to get path Π from μ to ν. Let node $\kappa'' = PathFind(\Pi, S_{20})$. If $\kappa'' = \nu$ then return the external face. Otherwise, return the node-triple $(\kappa, \kappa', \kappa'')$, where nodes κ and κ' are the parent and left-sibling of node κ''.

The following tree selection function takes query point p as an argument.

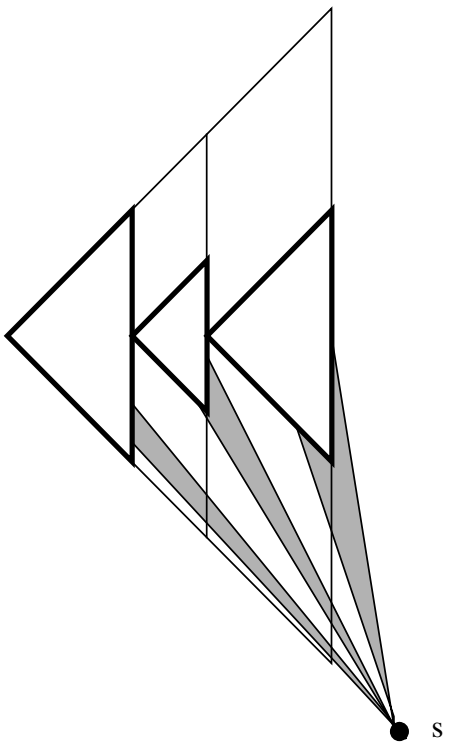

Figure 5.4: Finding the face containing a query point p. If p is in a shaded region then its face is found at a descendant P-node.

Selection Function S_{21} *Suppose η is the root of a path tree with children η' and η''. If point p is contained in $R_s(\text{tail}(\eta'))$ then return η', else return η''.*

We then perform step 5b by letting node $\kappa = \mathit{TreeFind}(\mu, S_{21}, p)$. If κ is node $\nu'(\mu'')$ on an edge path of node ν', then point p is on the face (ν', μ', μ''), where node μ' is the left-sibling of μ''. Otherwise, node κ is on a solid path Π. Let node μ' be the child of κ on Π. If p is on an edge of $R_s(\kappa)$, then return the edge. If p is to the right of the edge from the source of G_κ to $\mathit{sourceright}(\mu')$. Then p is contained in face (κ, μ', μ''), where μ'' is the right-sibling of μ'. Otherwise, point p is to the left of the edge from the source of G_κ to $\mathit{sourceleft}(\mu')$. Then p is contained in face (κ, μ''', μ'), where μ''' is the left-sibling of μ'.

Each of these steps can be implemented in $O(\log n)$ time. Therefore, operation *Locate* is performed in $O(\log n)$ time.

To implement operation *Window* we keep the data structure of [107] to to maintain the planar embedding of G. In particular given a face f of G in an upward embedding of series-parallel digraph G, we can find two lists of edges and vertices that comprise the left and right boundary of f.

Suppose $p = (x_p, y_p)$ and $q = (x_q, y_q)$ are points with $x_p < x_q$ and $y_p < y_q$, and let W be the window defined by p and q. Operation $\mathit{Window}(\nu, p, q)$ is realized as follows:

If W does not intersect $\Delta(\nu)$ then return an empty drawing. Otherwise, let $s = (x_s, y_s)$ be a scan point. Initialize s to p, and do the following, clipping to W:

- Perform *Locate*(ν, s). Let f be the returned face. Find the edge e that is to the right of s on the boundary of f. Draw and mark all unmarked edges and vertices that are in W and are reachable by forward and reverse edges from edge e.
- Repeat step 1, continuing around the boundary of W, finding unmarked edges of G that intersect the boundary of W.

We implement the search of the list for face f in step 1 by performing binary search on the location of the vertices on the boundary. Each *Draw* call takes $O(\log m)$ time, so finding an edge that intersects the boundary of W takes $O(\log^2 m)$ time. We traverse the edges reachable from e by using a modified breadth-first search that cuts at a marked node, or at the boundary of W. Hence, the internal j edges and vertices found by step 1 can be drawn in $O(j \cdot \log m)$ time.

Therefore, operation *Window* can be implemented in $O(k \cdot \log^2 m)$ time to draw k vertices and edges.

Operations *MakeDigraph* and *DeleteDigraph* can be trivially implemented in $O(1)$ time. Operations *Compose*, *Attach*, and *Detach* can each be implemented with a constant number of calls to *MakeDigraph*, *DeleteDigraph* and variations of the tree operations *Link* and *Cut*.

Consider now operation *InsertVertex*(v, e, e', e''), and let λ be the Q-node of e, and ν be the parent of λ, if it exists. We replace λ with an S-node μ having Q-node children for e' and e''. This can be done with two *MakeDigraph* operations, followed by a *Compose* operation and one each *Cut* and *Link* operation. Then, if μ is an S-node, we contract μ into ν. All this takes $O(\log m)$ time.

The inverse operation *DeleteVertex*(e, v, e', e'') is implemented similarly with a constant number of *MakeDigraph*, *DeleteDigraph*, *Link*, *Cut*, and *Expand* operations.

The restructuring of the SPQ-tree caused by *InsertEdge* and *DeleteEdge* is more complex. It is known [22] that if G is a series-parallel digraph with SPQ-tree T, and v' and v'' be two vertices of G. Then the digraph obtained by inserting an edge from v' to v'' is a series-parallel digraph if and only if either $v' = s$ and $v'' = t$, or v' and v'' are vertices of the skeleton of the same S-node of T, with v' preceding v''. If is e an edge of G represented by Q-node λ, then the digraph obtained by removing edge e is a series-parallel digraph if and only if one of the following conditions is true: the parent of λ in T is a P_Q-node, e is the only outgoing edge of s, or e is the only incoming edge of t. These conditions can each be tested in $O(\log m)$ time, where G has m edges. The restructuring to T required for *InsertEdge* and *DeleteEdge* also can be performed in $O(\log m)$ time.

The transformations to T that result from *InsertEdge* and *DeleteEdge* are demonstrated in figures 5.5 and 5.6. Notice that transitive edges are always added as the right child of a P_Q-node.

Each face of as series-parallel digraph G, with the exception of the external face, is created by a parallel composition. Therefore, we name each internal face f of G by a

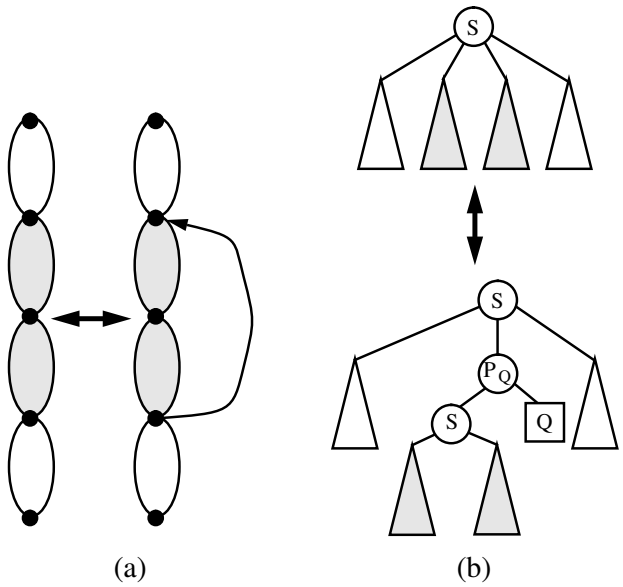

Figure 5.5: Modification of T in operation *InsertEdge* and *DeleteEdge*: splitting/merging an S-chain. (a) Insertion/deletion of an edge between nonconsecutive vertices of the skeleton of an S-chain. (b) Transformation of the SPQ-tree.

node-triple (ν, μ', μ''), where node ν is a P-node and nodes μ' and μ'' are such that f is formed by the parallel composition of $G_{\mu'}$ and $G_{\mu''}$.

5.4 Planar Graphs

In this section we present dynamic techniques for drawing planar graphs. First, we discuss upward drawings of planar *st*-digraphs, and next we extend the results to (undirected) biconnected planar graphs. Planar *st*-digraphs, which include series-parallel digraphs as a special case, were first introduced by Lempel, Even, and Cederbaum [72] in connection with a planarity testing algorithm, and they have subsequently been used in several applications, including planar graph embedding [18,27,107], graph drawing [26,29], and planar point location [37,56,88,114].

A *planar st-digraph* is a planar acyclic directed graph with exactly one source vertex s and exactly one sink vertex t, which is embedded in the plane such that s and t are on the boundary of the external face.

The following generalizes our definition of components of series-parallel digraphs. A digraph is *weakly connected* if its underlying undirected graph is connected. Let G be a planar *st*-digraph. An *open component* of G is a maximal weakly-connected subgraph G' of the digraph obtained from G by removing a separation pair $\{p, q\}$, such

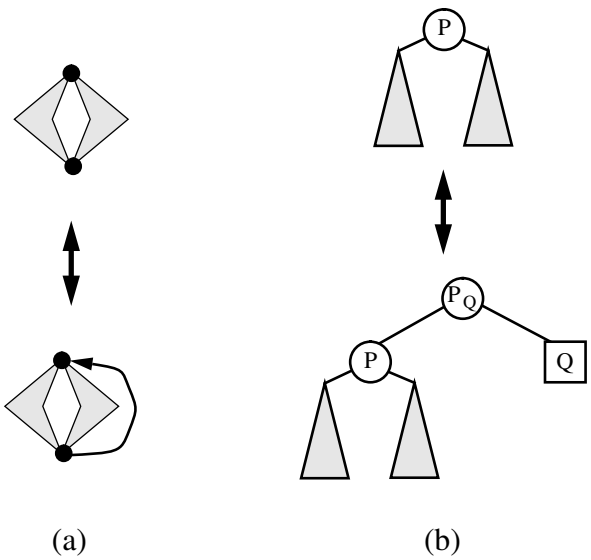

Figure 5.6: Modification of T in operation *InsertEdge* and *DeleteEdge*: extending/shortening a P-chain. (a) Insertion/deletion of an edge between consecutive vertices of the skeleton of an S-chain (the vertices are the source and sink of a parallel composition). (b) Transformation of the SPQ-tree.

that G' does not contain s or t. A *closed component* of G is an induced subgraph G' of G such that:

- G' is a planar pq-digraph;
- G' contains every vertex of G that is on some path from p to q;
- G' contains every outgoing edge of p, every incoming edge of q, and every incident edge of the remaining vertices of G'.

A *component* of G is either a closed or an open component.

5.4.1 Upward Drawings

We consider the following static drawing predicate $\mathcal{P}_S$:

Upward: The drawing is upward.

Planar: The drawing is planar.

Embedding-Preserving: The drawing preserves the embedding.

Grid: Vertices are placed at integer coordinates.

Polyline: Edges are drawn as polygonal lines.

Transitive-Bends: Transitive edges have at most one bend. The other edges are straight-lines. Hence the total number of bends is at most $2n - 5$.

Isomorphic: Isomorphic components have drawings that are congruent up to a translation.

Symmetric: Symmetric components have drawings that are congruent up to a translation and a reflection.

Quadratic-Area: The drawing has $O(n^2)$ area.

We dynamize the polyline drawing method of [29], which has the important property of displaying symmetries and isomorphisms of subgraphs. Note that we do not consider straight-line drawings because they may require exponential area [29].

Dynamic Environment

We consider a fully dynamic environment for the maintenance of upward drawings on a collection of embedded planar st-digraphs. Namely, we introduce the following set $\mathcal{O} = \mathcal{Q} \cup \mathcal{U}$ of operations:

- Query operations ($\mathcal{Q}$):
  - *Draw*(*vertex* v) — Return the (x, y) position of vertex v. The source is considered to be at $(0, 0)$.
  - *Draw*(*edge* e) — Return the (x, y) position of the endpoints of edge e. If e is a transitive edge, then also return the position of the bend of e.
- Update operations ($\mathcal{U}$):
  - *MakeDigraph* — Create a new elementary planar st-digraph G, consisting of a single edge.
  - *DeleteDigraph*(*edge* e) — Remove the elementary planar st-digraph consisting of single edge e.
  - *InsertEdge*(*vertex* v', v''; *edge* e; *face* f, f', f'') — Add edge $e = (v', v'')$ inside face f, which is decomposed into faces f' and f'', with f' to the left of e and f'' to the right.
  - *DeleteEdge*(*edge* e; *face* f) — Delete edge e and merge the two faces formerly on the two sides of e into a new face f.
  - *Expand*(*vertex* v, v', v''; *edge* e; *face* f', f'') — Expand vertex v into vertices v' and v'', which are connected by a new edge e with face f' to its left and face f'' to its right.
  - *Contract*(*vertex* v; *edge* e) — Contract edge e, and merge its endpoints into a new vertex v. Parallel edges resulting from the contraction are merged into a single edge.

Each update operation is allowed if the resulting digraph is itself a planar *st*-digraph.

Consider a vertex v of planar *st*-digraph G. Let $R^+(v)$ be the set of vertices of G reachable from v and let $R^-(v)$ be the set of vertices u of G such that u is reachable from v. For a pair of vertices (v', v'') the *subgraph of stable reachablilty* of (v', v'') is the subgraph induced by $R^-(v') \cup R^+(v'')$.

In the rest of this section we prove the following theorem:

Theorem 5.3 *Consider the following dynamic graph drawing problem:*

- *Class of graphs $\mathcal{G}$: embedded planar st-digraphs.*
- *Static drawing predicate $\mathcal{P}_S$: upward, planar, embedding-preserving, grid, polyline, transitive-bends, isomorphic, symmetric, quadratic-area drawing.*
- *Repertory of operations $\mathcal{O}$:* Draw, MakeDigraph, DeleteDigraph, InsertEdge, DeleteEdge, Expand, *and* Contract.
- *Dynamic drawing predicate $\mathcal{P}_D$:*
  - *the drawing of a component not affected by an update operation changes only by a translation;*
  - *After inserting or deleting an edge between v' and v'', the drawing of the subgraph of stable reachability of (v', v'') changes only by a translation.*

There exists a fully dynamic algorithm for the above problem with the following performance:

- *A planar st-digraph uses $O(n)$ space;*
- *Operations* MakeDigraph *and* DeleteDigraph *take each $O(1)$ time;*
- *Operations* Draw, InsertEdge, DeleteEdge, Expand, *and* Contract *take each $O(\log n)$ time.*

Data Structure

Let V be the set of vertices, E, be the set of edges, and F be the set of faces of planar *st*-digraph G. As shown in [29,108], there are two orderings on the set $V \cup E \cup F$, denoted L and R, such that if G has no transitive edges, a planar upward grid drawing of G is obtained by assigning to each vertex v x- and y-coordinates equal to the ranks of v in the restriction to V of L and R, respectively. This drawing method is extended to general planar *st*-digraphs by inserting a dummy vertex (a bend) along each transitive edge.

We represent sequences L and R by means of a path attribute system (see Section 3.2.1). Each node represents a vertex, edge, or face. Node attributes assume binary values: 1 if the node is associated with a vertex or a transitive edge, and 0

otherwise. The path attribute is the sum of the attributes of the nodes on the path. Sequences L and R are hence represented by two paths, denoted π_L and π_R, respectively. In a drawing query, we compute $x(v)$ (resp., y(v)) by splitting path π_L (resp., π_R) at the node associated with v, and evaluating the attribute of the left subpath so obtained.

After an update operation $O(1)$ edges of G become or cease to be transitive, and each such edge can be identified in $O(\log n)$ time. The corresponding modifications of node attributes can be done in $O(\log n)$ time. Also, sequences $<_L$ and $<_R$ are updated by means of $O(1)$ split/concatenate operations [108], so that the corresponding updates on π_L and π_R take $O(\log n)$ time. We conclude that our dynamic data structure uses $O(n)$ space and supports each operation in $O(\log n)$ time.

5.4.2 Visibility Drawings

The concept of *visibility* plays a fundamental role in a variety of geometric problems and applications, such as art gallery problems [80], VLSI layout [61,97,123], motion planning [58], and graph drawing [26,110]. A *visibility representation* Θ for a directed graph G maps each vertex v of G to a horizontal segment $\Theta(v)$ and each edge (u, v) to a vertical segment $\Theta(u, v)$ that has its lower endpoint on $\Theta(u)$, its upper endpoint on $\Theta(v)$, and does not intersect any other horizontal segment. Besides having many applications, visibility representations are also of intrinsic theoretical interest, and their combinatorial properties have been extensively investigated [33,109,111,113,124,125].

We consider the following static drawing predicate $\mathcal{P}_S$:

Visibility: The drawing is a visibility representation.

Grid: The endpoints of vertex- and edge-segments are placed at integer coordinates.

Isomorphic: Isomorphic components have drawings that are congruent up to a translation.

Quadratic-Area: The drawing has $O(n^2)$ area.

In the rest of this section we prove the following theorem:

Theorem 5.4 *Consider the following dynamic graph drawing problem:*

- *Class of graphs $\mathcal{G}$: embedded planar st-digraphs.*
- *Static drawing predicate $\mathcal{P}_S$: visibility, grid, isomorphic, quadratic-area drawing.*
- *Repertory of operations $\mathcal{O}$:* Draw, MakeDigraph, DeleteDigraph, InsertEdge, DeleteEdge, Expand, *and* Contract.
- *Dynamic drawing predicate $\mathcal{P}_D$: the drawing of an open component not affected by an update operation changes only by a translation.*

There exists a fully dynamic algorithm for the above problem with the following performance:

- *a planar st-digraph uses $O(n)$ space;*
- *Operations* MakeDigraph *and* DeleteDigraph *take each $O(1)$ time;*
- *Operations* Draw, InsertEdge, DeleteEdge, Expand, *and* Contract *take each $O(\log n)$ time.*

Data Structure

We recall that in a planar *st*-digraph the incoming edges of each vertex appear consecutively around the vertex, and so do the outgoing edges [109]. The faces separating the incoming and outgoing edges of vertex v to the left and right of v are called $left(v)$ and $right(v)$, respectively. Also, the boundary of each face f consists of two directed paths enclosing f, each starting from the unique lowest vertex $low(f)$ and ending at the unique highest vertex $high(f)$. A visibility representation for G can be constructed by the following variation of previous sequential algorithms [26,96,109].

1. Construct the directed dual of planar *st*-digraph G as follows: (a) Construct the dual graph G^* of G. (b) Orient the dual of each edge e of G from the face to the left of e to the face to the right of e. (c) Expand the vertex of G^* associated with the external face of G into two vertices, denoted s^* and t^*, between faces s and t of G^*. (d) Remove edge (t^*, s^*) of G^* and let D be the resulting planar s^*t^*-digraph.
2. Compute a topological ordering $Y(v)$ of the vertices of G.
3. Compute a topological ordering $X(f)$ of the vertices of D.
4. Draw each vertex-segment $\Theta(v)$ at y-coordinate $Y(v)$ and between x-coordinates $X(left(v))$ and $X(right(v)) - 1$.
5. Draw each edge-segment $\Theta(e)$ at x-coordinate $X(left(e))$ and between y-coordinates $Y(low(e))$ and $Y(high(e))$.

Consider the orderings L and R defined in Section 5.4.2. The restriction of sequence L (or R) to V is a topological ordering [108]. Hence, we can use a path attribute system (see Section 3.2.1) to dynamically maintain topological orderings X and Y, such that the position of a vertex- or edge-segment can be computed in $O(\log n)$ time.

5.4.3 Biconnected Planar Graphs

Finally, we extend our results to (undirected) biconnected planar graphs. We consider the following static drawing predicate $\mathcal{P}_S$:

Planar: The drawing is planar.

Embedding-Preserving: The drawing preserves the embedding.

Grid: Vertices are placed at integer coordinates.

Polyline: Edges are drawn as polygonal lines.

One-Bend: Each edge has at most one bend and the total number of bends is at most $2n - 5$.

Quadratic-Area: The drawing has $O(n^2)$ area.

We consider a semi dynamic environment for the maintenance of polyline drawings on a collection of biconnected planar graphs. Namely, we introduce the following set $\mathcal{O} = \mathcal{Q} \cup \mathcal{U}$ of operations:

- Query operations ($\mathcal{Q}$):
  - *Draw*(*vertex* v) — Return the (x, y) position of vertex v.
  - *Draw*(*edge* e) — Return the (x, y) position of the endpoints of edge e. If e has a bend, then also return the position of the bend.
- Update operations ($\mathcal{U}$):
  - *MakeGraph* — Create a new elementary biconnected planar graph G, consisting of a cycle with three vertices.
  - *InsertEdge*(*vertex* v', v''; *edge* e; *face* f, f', f'') — Add edge $e = (v', v'')$ inside face f, which is decomposed into faces f' and f''.
  - *InsertVertex*(*vertex* v; *edge* e, e', e'') — Insert vertex v on edge e, which is decomposed into edges e' and e''.

As shown in [27], this repertory of operation is complete; i.e., any n-vertex biconnected planar graph can be assembled by means of $O(n)$ operations of the repertory. In the rest of this section we prove the following theorem:

Theorem 5.5 *Consider the following dynamic graph drawing problem:*

- *Class of graphs $\mathcal{G}$: biconnected planar graphs.*
- *Static drawing predicate $\mathcal{P}_S$: planar, embedding-preserving, grid, poly-line, one-bend, quadratic-area drawing.*
- *Repertory of operations $\mathcal{O}$:* Draw, MakeGraph, InsertEdge, *and* InsertVertex.

There exists a semi dynamic algorithm for the above problem with the following performance:

- *A biconnected planar graph uses $O(n)$ space;*

- *Operation* MakeDigraph *takes* $O(1)$ *time;*
- *Operations* Draw, InsertEdge, *and* InsertVertex *take each* $O(\log n)$ *time.*

Note that we do not maintain a dynamic drawing predicate.

Data Structure

We maintain on-line an orientation of G into a planar st-digraph. This can be done using the techniques of [107].

We can extend Theorem 5.5 to support the insertion of an edge between two vertices that are not on the same face of the current embedding, using the techniques of [27]. In this case the embedding has to be modified in order to preserve planarity, and the time complexity of operation *InsertEdge* is amortized.

With a similar approach, we can derive from the data structure of Theorem 5.4 a semi-dynamic data structure for maintaining on-line visibility representations of biconnected planar graphs. The space and time complexity is the same as in Theorem 5.5.

Chapter 6

Conclusions and Open Problems

As stated in the introduction, the goal of this thesis was to demonstrate generalized techniques to maintain the solutions of dynamic algorithms for graph and graph drawing problems, and to present dynamic algorithms based on our techniques. We have presented a fully-dynamic data structure, called a tree attribute system, for the maintenance of tree based algorithms. Using tree attribute systems, we introduce two new data structures, linear expression trees and linear attribute grammars, for the maintenance of tree based expressions.

Using these techniques, we give the first polylogarithmic time algorithm for the maintenance of a decomposition of tree-width two graphs. We present fully dynamic techniques to maintain the solution to a large number problems on tree-width two graphs, as well as series-parallel graphs and trees.

We introduce a model for dynamic graph drawing. Using linear attribute grammars, we present fully dynamic techniques to maintain a number of types of drawings of planar graphs and classes of planar graphs. We include window and point location queries.

Open problems include:

- Lower the complexity of maintaining the decomposition of tree-width two graphs to $O(\log m)$ time per update.
- Extend our techniques to the dynamic maintenance of graphs in $TW(k)$ for $k > 2$. Recently, an on-line algorithm has been presented for the maintenance of tree-width three graphs [21].
- Lower the complexity of returning a connectible-pair shortest path / minimum-cut to $O(k + \log n)$ (or $O(k + \log^2 n)$).
- Lower the complexity of window queries in trees and series-parallel digraphs to $O(k + \log n)$.
- Extend the techniques for planar st-digraphs and general planar graphs to support point-location and window queries.

- Develop dynamic algorithms for planar straight-line drawings of general planar graphs. The techniques of [46,98] appear difficult to dynamize.
- Dynamically maintain orthogonal drawings with the minimum number of bends. The static algorithm of [106] is based on network flow techniques for which no dynamic methods are known.
- Devise dynamic algorithms to test whether a digraph admits an upward planar drawing. Static algorithms that perform this test are known only for triconnected digraphs [10]and for single-source digraphs [62]. Semidynamic planarity testing can be done with $O(\log n)$ query and insertion time [27]. Recently, a fully dynamic planarity testing technique with $O(n^{2/3})$ query and update time has been discovered [52].

Bibliography

[1] G.M. Adel'son-Vel'skii and E.M. Landis, "An Information Organization Algorithm," *Doklady Akad. Nauk SSSR* 146 (1962), 263–266.

[2] B. Alpern, R. Hoover, B. Rosen, P. Sweeney, and F.K. Zadeck, "Incremental Evaluation of Computational Circuits," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1990), 32–42.

[3] S. Arnborg, "Reduced State Enumeration–Another Algorithm for Reliability Evaluation," *IEE Transactions on Reliability* R-27 (1978), 101–105.

[4] S. Arnborg, "Efficient Algorithms for Combinatorial Problems on Graphs with Bounded Decomposbility, A Survey ," *BIT* 25 (1985), 2–33.

[5] Stefan Arnborg, Derek. G. Corneil, and Andrej Proskurowski, "Complexity of Finding Embeddings in a k-Tree," *SIAM. J. Alg. Disc. Meth* 8 (1987), 277–284.

[6] 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.

[7] S.W. Bent, D.D. Sleator, and R.E. Tarjan, "Biased Search Trees," *SIAM J. Computing* 14 (1985), 545–568.

[8] A.M. Berman, M.C. Paull, and B.G. Ryder, "Proving Relative Lower Bounds for Incremental Algorithms," *Acta Informatica* (to appear).

[9] P. Bertolazzi, R.F. Cohen, G. Di Battista, R. Tamassia, and I.G. Tollis, "How to Draw a Series-Parallel Digraph," *Proc. SWAT* (1992).

[10] P. Bertolazzi and G. Di Battista, "On Upward Drawing Testing of Triconnected Digraphs," *Proc. ACM Symp. on Computational Geometry* (1991).

[11] G. M. Beshers and R. H. Campbell, "Maintained and Constructor Attributes," *Proc. ACM Symp. on Language Issues in Programming Environments, ACM SIGPLAN Notices* (1985), 34–42.

[12] Hans. L. Bodlaender, "NC-Algorithms for Graphs with small Treewidth," Dept. of Computer Science, University of Utrechet, Utrechet, Technical Report RUUU-CS-88-4, 1988.

[13] K. Booth and G. Lueker, "Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-Tree Algorithms," *J. of Computer and System Sciences* 13 (1976), 335–379.

[14] 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.

[15] Y.-J. Chiang, F.P. Preparata, and R. Tamassia, "A Unified Approach to Dynamic Point Location, Ray Shooting and Shortest Paths in Planar Maps," Dept. Computer Science, Brown Univ., Technical Report CS-92-07, 1992.

[16] Y.-J. Chiang and R. Tamassia, "Dynamic Algorithms in Computational Geometry," Dept. of Computer Science, Brown Univ., Technical Report CS-91-24, 1991.

[17] Y.-J. Chiang and R. Tamassia, "Dynamization of the Trapezoid Method for Planar Point Location," *Int. J. of Computational Geometry Applications* (to appear).

[18] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa, "A Linear Algorithm for Embedding Planar Graphs Using PQ-Trees," *J. of Computer and System Sciences* 30 (1985), 54–76.

[19] F. Chin and D. Houk, "Algorithms for Updating Minimum Spanning Trees," *J. Computer Systems Sciences* 16 (1978), 333–344.

[20] R. F. Cohen, S. Sairam, R. Tamassia, and J. S. Vitter, "An Algorithmic Framework for Dynamic Property Systems in Bounded Tree Width Graphs," 1991, (submitted for publication).

[21] R.F. Cohen, S. Sairam, R. Tamassia, and J.S. Vitter, "Dynamic Algorithms for Bounded Tree-Width Graphs," Brown University, Manuscript, 1992.

[22] R.F. Cohen and R. Tamassia, "Dynamic Expression Trees and their Applications," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1991), 52–61.

[23] C. J. Colbourn and A. Proskurowski, "Concurrent Transmissions in Broadcast Networks," *Proc. ICALP, Springer-Verlag , Berlin-Heidelberg-New York, Lecture Notes in Computer Science* 172 (1984), 128–136.

[24] C. Crane, "Linear Lists and Priority Queues as Balanced Binary Trees," Computer Science Dept., Stanford Univ., Technical Report STAN-CS-72-259 (Ph.D. Dissertation), 1972.

[25] G. Di Battista, W.-P. Liu, and I. Rival, "Bipartite Graphs, Upward Drawings, and Planarity," *Information Processing Letters* 36 (1990), 317–322.

[26] G. Di Battista and R. Tamassia, "Algorithms for Plane Representations of Acyclic Digraphs," *Theoretical Computer Science* 61 (1988), 175–198.

[27] G. Di Battista and R. Tamassia, "Incremental Planarity Testing," *Proc. 30th IEEE Symp. on Foundations of Computer Science* (1989), 436–441.

[28] G. Di Battista and R. Tamassia, "On-Line Graph Algorithms with SPQR-Trees," *Automata, Languages and Programming (Proc. 17th ICALP), Lecture Notes in Computer Science* 442 (1990), 598–611.

[29] G. Di Battista, R. Tamassia, and I.G. Tollis, "Area Requirement and Symmetry Display in Drawing Graphs," *Proc. ACM Symp. on Computational Geometry* (1989), 51–60.

[30] G. Di Battista, R. Tamassia, and I.G. Tollis, "Area Requirement and Symmetry Display of Planar Upward Drawings," *Discrete & Computational Geometry* 7 (1992), 381–401.

[31] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R.E. Tarjan, "Dynamic Perfect Hashing: Upper and Lower Bounds," *Proc. 29th IEEE Symp. on Foundations of Computer Science* (1988), 524–531.

[32] D. Dolev, F.T. Leighton, and H. Trickey, "Planar Embedding of Planar Graphs," in *Advances in Computing Research, vol. 2*, F.P. Preparata, ed., JAI Press Inc., Greenwich, CT, 1984, 147–161.

[33] P. Duchet, Y. Hamidoune, M. Las Vergnas, and H. Meyniel, "Representing a Planar Graph by Vertical Lines Joining Different Levels," *Discrete Mathematics* 46 (1983), 319–321.

[34] R.J. Duffin, "Topology of Series Parallel Networks," *J. Math. Anal. Appl.* 10 (1965), 303–318.

[35] P. Eades and X. Lin, "How to Draw Directed Graphs," *Proc. IEEE Workshop on Visual Languages (VL'89)* (1989), 13–17.

[36] P. Eades and R. Tamassia, "Algorithms for Automatic Graph Drawing: An Annotated Bibliography," Dept. of Computer Science, Brown Univ., Technical Report CS-89-09, 1989.

[37] H. Edelsbrunner, L.J. Guibas, and J. Stolfi, "Optimal Point Location in a Monotone Subdivision," *SIAM J. Computing* 15 (1986), 317–340.

[38] D. Eppstein, G.F. Italiano, R. Tamassia, R.E. Tarjan, J. Westbrook, and M. Yung, "Maintenance of a Minimum Spanning Forest in a Dynamic Planar Graph," *Proc. First ACM-SIAM Symp. on Discrete Algorithms* (1990), 1–11.

[39] D. Eppstein, G.F. Italiano, R. Tamassia, R.E. Tarjan, J. Westbrook, and M. Yung, "Maintenance of a Minimum Spanning Forest in a Dynamic Plane Graph," *J. of Algorithms* 13 (1992), 33–54.

[40] S. Even and H. Gazit, "Updating Distances in Dynamic Graphs," *Methods of Operations Research* 49 (1985), 371–387.

[41] S. Even and Y. Shiloach, "An On-Line Edge Deletion Problem," *J. ACM* 28 (1981), 1–4.

[42] S. Even and R.E. Tarjan, "Computing an st-Numbering," *Theoretical Computer Science* 2 (1976), 339–344.

[43] A. M. Farley, "Networks Immune to Isolated Failures," *Networks* 11 (1981), 255–268.

[44] A. M. Farley and A. Proskurowski, "Networks Immune to Isolated Line Failures," *Networks* 12 (1982), 393–403.

[45] M. Formann, T. Hagerup, J. Haralambides, M. Kaufmann, F.T. Leighton, A. Simvonis, E. Welzl, and G. Woeginger, "Drawing Graphs in the Plane with High Resolution," *Proc. IEEE Symp. on Foundations of Computer Science* (1990), 86–95.

[46] H. de Fraysseix, J. Pach, and R. Pollack, "How to Draw a Planar Graph on a Grid," *Combinatorica* 10 (1990), 41–51.

[47] H. de Fraysseix and P. Rosenstiehl, "A Depth-First-Search Characterization of Planarity," *Annals of Discrete Mathematics* 13 (1982), 75–80.

[48] G.N. Frederickson, "Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications," *SIAM J. Computing* 14 (1985), 781–798.

[49] G.N. Frederickson, "Ambivalent Data Structures for Dynamic 2-Edge-Connectivity and k Smallest Spanning Trees," *Proc. 32th IEEE Symp. on Foundations of Computer Science* (1991).

[50] Z. Galil and G.F. Italiano, "Fully Dynamic Algorithms for Edge-Connectivity Problems," *Proc. 23th ACM Symp. on Theory of Computing* (1991), 317–327.

[51] Z. Galil and G.F. Italiano, "Maintaining Biconnected Components of Dynamic Planar Graphs," *Automata, Languages and Programming (Proc. 18th ICALP), Lecture Notes in Computer Science* (1991).

[52] Z. Galil and G.F. Italiano, "Fully Dynamic Planarity Testing ," *Proc. 24th ACM Symp. on Theory of Computing (to appear)* (1992).

[53] Z. Galil and G.F. Italiano, "Fully Dynamic Algorithms for 2-Edge Connectivity," *SIAM J. on Computing* (to appear).

[54] Z. Galil and G.F. Italiano, "Maintaining the 3-Edge Connected Components of a Graph On-Line.," *SIAM J. on Computing* (to appear).

[55] Z. Galil and A. Naamad, "An $O(EV \log^2 V)$ algorithm for the maximal flow problem," *J. Computer and System Sciences* 21 (1980), 203–217.

[56] M.T. Goodrich and R. Tamassia, "Dynamic Trees and Dynamic Point Location," *Proc. 23th ACM Symp. on Theory of Computing* (1991), 523–533.

[57] L.J. Guibas and R. Sedgewick, "A Dichromatic Framework for Balanced Trees," *Proc. 19th IEEE Symp. on Foundations of Computer Science* (1978), 8–21.

[58] L.J. Guibas and F.F. Yao, "On Translating a Set of Rectangles," in *Advances in Computing Research, vol. 1*, F.P. Preparata, ed., JAI Press Inc., Greenwich, CT, 1983, 61–77.

[59] X. He, "Efficient Parallel Algorithms for Series Parallel Graphs," *J. Algorithms* 12 (1991), 409–430.

[60] J. Hopcroft and R.E. Tarjan, "Efficient Planarity Testing," *J. ACM* 21 (1974), 549–568.

[61] M.Y. Hsueh and D.O. Pederson, "Computer-Aided Layout of LSI Circuit Building-Blocks," *Proc. IEEE Int. Symp. on Circuits and Systems* (1979), 474–477.

[62] M.D. Hutton and A. Lubiw, "Upward Planar Drawing of Single Source Acyclic Digraphs," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1991), 203–211.

[63] T. Ibaraki and N. Katoh, "On-Line Computation of Transitive Closure of Graphs," *Information Processing Letters* 16 (1983), 95–97.

[64] G.F. Italiano, "Amortized Efficiency of a Path Retrieval Data Structure," *Theoretical Computer Science* 48 (1986), 273–281.

[65] G.F. Italiano, "Finding Paths and Deleting Edges in Directed Acyclic Graphs," *Information Processing Letters* 28 (1988), 5–11.

[66] G.F. Italiano, A. Marchetti-Spaccamela, and U. Nanni, "Dynamic Data Structures for Series-Parallel Graphs," *Proc. WADS' 89,* LNCS 382 (1989), 352–372.

[67] L. G. Jones, "Incremental Compaction of Flat Symbolic IC Layouts," Department of Computer Science, University of Illinois, Urbana, Illinois, Technical Report No. UIUCDCS-R-87-1386, 1987.

[68] L. G. Jones and J. Simon, "Hierarchical VLSI Design Systems Based on Attribute Grammars," *Proc. 13th ACM Symp. on Principles of Programming Languages* (1986), 58–69.

[69] A. Kanevsky, R. Tamassia, J. Chen, and G. Di Battista, "On-Line Maintenance of the Four-Connected Components of a Graph," *Proc. 32th IEEE Symp. on Foundations of Computer Science* (1991), 793–801.

[70] D. E. Knuth, "Semantics of Context-Free Languages," *Mathematical Systems Theory* 2 (1968), 127–145.

[71] J. Lagergren, "Efficient Parallel Algorithms for Tree-Decomposition and Related Problems," *Proc. 31rst Foundations of Computer Science* (1990), 173–182.

[72] A. Lempel, S. Even, and I. Cederbaum, "An Algorithm for Planarity Testing of Graphs," in *Theory of Graphs, Int. Symposium (Rome, 1966)*, Gordon and Breach, New York, 1967, 215–232.

[73] C.C. Lin and R.C. Chang, "On the Dynamic Shortest Path Problem," *Proc. Int. Workshop on Discrete Algorithms and Complexity* (1989), 203–212.

[74] R. Lipton, D. Rose, and R.E. Tarjan, "Generalized Nested Dissection," *SIAM J. Numerical Analysis* 16 (1979), 346–358.

[75] S.M. Malitz and A. Papakostas, "On the Angular Resolution of Planar Graphs," *Proc. ACM Symp. on Theory of Computing* (1992).

[76] J. Mataoušk and R. Thomas, "Algorithms Finding Tree-Decomposition of Graphs," *Journal of Algorithms* 12 (1991), 1–22.

[77] S. Moen, "Drawing Dynamic Trees," *IEEE Software* 7 (1990), 21–28.

[78] E. M. Neufeldt and C. J. Colbourn, "The Most Reliable Series Parallel Networks," Dept. of Computing Science, University of Saskatchewan., Technical Report TR-83-7, 1983.

[79] T. Nishizeki and N. Chiba, *Planar Graphs: Theory and Algorithms*, Annals of Discrete Mathematics 32, North-Holland, 1988.

[80] J. O'Rourke, *Art Gallery Theorems and Algorithms*, Oxford University Press, 1987.

[81] M. Overmars, "The Design of Dynamic Data Structures," *Lecture Notes in Computer Science* 156 (1983).

[82] M.S. Patterson and F.F. Yao, "Optimal Binary Space Partitions for Orthoganal Objects," *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms* (1990), 100–106.

[83] J.A. La Poutre, "Dynamic Graph Algorithms and Data Structures," Dept. of Computer Science, University of Utrechet, Utrechet, Ph.D. Thesis, 1991.

[84] J.A. La Poutré, "Maintenance of Triconnected Components of Graphs," *Automata, Languages and Programming (Proc. 19th ICALP), Lecture Notes in Computer Science (to appear)* (1992).

[85] 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.

[86] F.P. Preparata, "A New Approach to Planar Point Location," *SIAM J. Computing* 10 (1981), 473–483.

[87] F.P. Preparata and R. Tamassia, "Fully Dynamic Techniques for Point Location and Transitive Closure in Planar Structures," *Proc. 29th IEEE Symp. on Foundations of Computer Science* (1988), 558–567.

[88] F.P. Preparata and R. Tamassia, "Fully Dynamic Point Location in a Monotone Subdivision," *SIAM J. Computing* 18 (1989), 811–830.

[89] J.H. Reif, "A Topological Approach to Dynamic Graph Connectivity," *Information Processing Letters* 25 (1987), 65–70.

[90] E. Reingold and J. Tilford, "Tidier Drawing of Trees," *IEEE Trans. on Software Engineering* SE-7 (1981), 223–228.

[91] T.W. Reps, *Generating Language-Based Environments*, The MIT Press, 1984.

[92] T.W. Reps and T. Teitelbaum, *The Synthesizer Generator*, Springer-Verlag, 1989.

[93] Neil Robertson and P. D. Seymour, "Graph Minors. XIII. Disjoint Paths Problem," Submitted for Publication.

[94] Neil Robertson and P. D. Seymour, "Graph Minors. II. Algorithmic Aspects of Tree-Width," *Journal of Algorithms* 7 (1986), 309–322.

[95] H. Rohnert, "A Dynamization of the All-Pairs Least Cost Problem," *Proc. STACS '85* (1985), 279–286.

[96] P. Rosenstiehl and R.E. Tarjan, "Rectilinear Planar Layouts of Planar Graphs and Bipolar Orientations," *Discrete & Computational Geometry* 1 (1986), 343–353.

[97] M. Schlag, F. Luccio, P. Maestrini, D.T. Lee, and C.K. Wong, "A Visibility Problem in VLSI Layout Compaction," in *Advances in Computing Research, vol. 2*, F.P. Preparata, ed., JAI Press Inc., Greenwich, CT, 1985, 259–282.

[98] W. Schnyder, "Embedding Planar Graphs on the Grid," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1990), 138–148.

[99] C. Schwarz, M. Smid, and J. Snoeyink, "An Optimal Algorithm for the On-line Closest-Pair Problem," *Proc. ACM Symp. on Computational Geometry* (1992).

[100] D.D. Sleator and R.E. Tarjan, "A Data Structure for Dynamic Trees," *J. Computer Systems Sciences* 24 (1983), 362–381.

[101] D.D. Sleator and R.E. Tarjan, "Self-Adjusting Binary Search Trees," *J. ACM* 32 (1985), 652–686.

[102] P.M. Spira and A. Pan, "On Finding and Updating Spanning Trees and Shortest Paths," *SIAM J. Computing* 4 (1975), 375–380.

[103] L. Stockmeyer, "Optimal Orientation of Cells in Slicing Floorplan Design," *Information and Control* 57 (1983), 91–101.

[104] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for Visual Understanding of Hierarchical Systems," *IEEE Trans. on Systems, Man, and Cybernetics* SMC-11 (1981), 109–125.

[105] K.J. Supowit and E.M. Reingold, "The Complexity of Drawing Trees Nicely," *Acta Informatica* 18 (1983), 377–392.

[106] R. Tamassia, "On Embedding a Graph in the Grid with the Minimum Number of Bends," *SIAM J. Computing* 16 (1987), 421–444.

[107] R. Tamassia, "A Dynamic Data Structure for Planar Graph Embedding," *Proc. 15th ICALP,* LNCS 317 (1988), 576–590.

[108] R. Tamassia and F.P. Preparata, "Dynamic Maintenance of Planar Digraphs, with Applications," *Algorithmica* 5 (1990), 509–527.

[109] R. Tamassia and I.G. Tollis, "A Unified Approach to Visibility Representations of Planar Graphs," *Discrete & Computational Geometry* 1 (1986), 321–341.

[110] R. Tamassia and I.G. Tollis, "Planar Grid Embedding in Linear Time," *IEEE Trans. on Circuits and Systems* CAS-36 (1989), 1230–1234.

[111] R. Tamassia and I.G. Tollis, "Tessellation Representations of Planar Graphs," *Proc. 27th Annual Allerton Conf.* (1989), 48–57.

[112] R. Tamassia and I.G. Tollis, "Reachability in Planar Digraphs," Computer Science Program, Univ. of Texas at Dallas, Technical Report UTDCS-29-90, 1990.

[113] R. Tamassia and I.G. Tollis, "Representations of Graphs on a Cylinder," *SIAM J. on Discrete Mathematics* 4 (1991), 139–149.

[114] R. Tamassia and J.S. Vitter, "Parallel Transitive Closure and Point Location in Planar Structures," *SIAM J. Computing* 20 (1991), 708–725.

[115] R.E. Tarjan, "Amortized Computational Complexity," *SIAM J. Algebraic Discrete Methods* 6 (1985), 306–318.

[116] R.E. Tarjan and J. van Leeuwen, "Worst-Case Analysis of Set-Union Algorithms," *J. ACM* 31 (1984), 245–281.

[117] C. Thomassen, "Planar Acyclic Oriented Graphs," *Order* 5 (1989), 349–361.

[118] J. Valdes, R.E. Tarjan, and E.L. Lawler, "The Recognition of Series Parallel Digraphs," *SIAM J. on Computing* 11 (1982), 298–313.

[119] A. Wald and C. J. Colbourn, "Steiner Trees, Partial 2-Trees, and Minimum IFI networks," *Networks* 13 (1983), 159–167.

[120] J. Westbrook, "Algorithms and Data Structures for Dynamic Graph Problems," Dept. Computer Science, Princeton Univ., Ph.D. dissertation (Technical Report CS-TR-229-89), 1989.

[121] J. Westbrook, "Fast Incremental Planarity Testing," *Automata, Languages and Programming (Proc. 19th ICALP), Lecture Notes in Computer Science (to appear)* (1992).

[122] J. Westbrook and R.E. Tarjan, "Maintaining Bridge-Connected and Biconnected Components On-Line," *Algorithmica* (1989 (to appear in Algorithmica)).

[123] S. Wimer, I. Koren, and I. Cederbaum, "Floorplans, Planar Graphs, and Layouts," *IEEE Trans. on Circuits and Systems* 35 (1988), 267–278.

[124] S.K. Wismath, "Characterizing Bar Line-of-Sight Graphs," *Proc. ACM Symp. on Computational Geometry* (1985), 147–152.

[125] S.K. Wismath, "Weighted Visibility Graphs of Bars and Related Flow Problems," *Algorithms and Data Structures (Proc. WADS'89)* (1989), 325–334.

[126] D. Yellin, "A Dynamic Transitive Closure Algorithm," IBM T.J. Watson Research Center, Research Report, 1988.