A Framework for Dynamic Graph Drawing

R .F. Cohen[1]
G. Di Battista[2]
R. Tamassia,[3]
I. G. Tollis[4]

Technical Report No. CS-92-34
August 1992

[1]Brown University Computer Science Box 1910, Providence, RI 02912
[2]Dipartimento di Informatica e Sistemistica Universita di Roma La Sapienza, Via Salaria, 113 - Rome, Italy 00198
[3]Brown University Computer Science Box 1910, Providence, RI 02912
[4]Department of Computer Science The University of Texas at Dallas Richardson, Texas 75083-0688

Dynamic Graph Drawing*

Robert F. Cohen[⋆] rfc@cs.brown.edu

Giuseppe Di Battista[○] dibattista@iasi.rm.cnr.it

Roberto Tamassia[⋆] rt@cs.brown.edu

Ioannis G. Tollis[◇] tollis@utdallas.edu

Abstract

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.

The main contributions of this paper can be summarized as follows:

- We devise a model for dynamic graph algorithms, based on performing queries and updates on an implicit representation of the drawing, and we show its applications.
- We present several efficient dynamic drawing algorithms for trees, series-parallel digraphs, planar *st*-digraphs, and planar graphs. These algorithms adopt a variety of representations (e.g., straight-line, polyline, visibility), and update the drawing in a smooth way.

(May 1992)

*Research supported in part by the National Science Foundation under grant CCR-9007851, by the U.S. Army Research Office under grant DAAL03-91-G-0035, by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-91-J-4052, ARPA order 8225, by Cadre Technologies, Inc., by the Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo of the Italian National Research Council, and by the Esprit II BRA of the European Community (project ALCOM). An extended abstract of this paper was presented at the *Eighth ACM Symposium on Computational Geometry*, Berlin, 1992.

⋆Dept. of Computer Science, Brown University, Providence, R.I. 02912–1910. The work of these authors was carried out in part while visiting the "Istituto di Analisi dei Sistemi ed Informatica" of the Italian National Research Council (IASI – CNR).

○Dipart. di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Rome, Italy.

◇Dept. of Computer Science, The University of Texas at Dallas, Richardson, TX 75083–0688. The work of this author was carried out in part while visiting the "Istituto di Analisi dei Sistemi ed Informatica" of the Italian National Research Council (IASI – CNR).

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 [15]. While this area has recently received increasing attention (see, e.g., [3,10,18,19,25,28, 36]), the study of drawing graphs in a dynamic setting has been an open problem. Previous work [29] 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 paper 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. In the following we denote with n and m the number of vertices and edges of a given graph.

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. [30]). 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 [3,6,7,9,14,25,39,49]. 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 [11]. Further results on the area of planar drawings appear in [2,10,19,36].

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 $\mathcal{P}_D$.

1.3 Overview

The rest of this paper is organized as follows. In Section 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 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 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.

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.

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 [33] 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 [33] are less wide than those produced by our algorithm. Note that finding drawings of minimum width that satisfy the above properties is NP-hard [40].

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

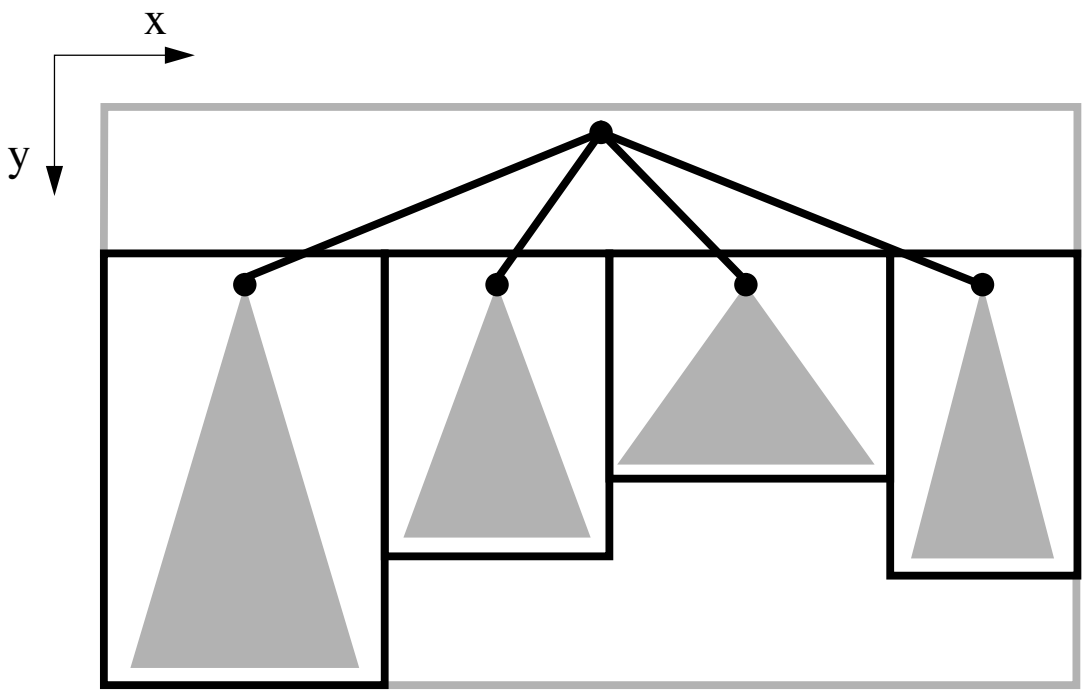

Figure 1: Geometric constructions in the $\square$-algorithm

traversal computes in pre-order the positions of the vertices of T. Each traversal can be performed in linear time. $\square$

2.2 Dynamic Environment

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

- 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 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.*

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 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 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.

Lemma 2 *Consider a node σ and an ancestor τ of σ in tree T. Then the dependence of values* width(τ), level(σ), *and* reference(σ) *on values* width(σ), level(τ), *and* reference(τ) *can be expressed as:*

$$\begin{aligned} \text{width}(\tau) &= \text{width}(\sigma) + a \\ \text{level}(\sigma) &= \text{level}(\tau) + b \\ \text{reference}(\sigma) &= \text{reference}(\tau) + c \end{aligned}$$

where a, b and c are independent from width(σ), level(τ), *and* reference(τ).

Proof: By induction on the length of the path from node σ to node τ. If $\sigma = \tau$ then the lemma is trivially true. The inductive step follows from the equations of Table 1. □

Given a path Π with head σ and tail τ, the *transfer vector* $h(\Pi)$ is the 3-tuple (a, b, c) of values for the calculation of $width(\tau)$, $level(\sigma)$, and $reference(\sigma)$ (see Lemma 2).

Our data structure consists of representing an n-node tree T as a collection of disjoint directed paths, where edges are directed from child to parent. 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.

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 a child μ_i. The *left-tree* of ν with respect to Π consists of node ν with children $\mu_1, \ldots, \mu_{i-1}$. The *right-tree* of ν with respect to Π is defined similarly. If node ν is the head of path Π, then the left tree contains children $\mu_1, \ldots, \mu_d$, and the right tree is empty. 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 according to the left-to-right order of the children. The *right edge-path* of ν with respect to Π, $\Pi_r(\nu)$ is defined similarly.

Our data structure consists of representing as balanced search trees the following paths:

- the solid paths of T; and
- for each node ν of T, the left and right edge-paths of ν with respect to the solid paths through ν.

For each such path Π, the associated balanced binary tree is denoted B_Π and is called the *path-tree* of Π. Also, we identify Π with the root ζ of B_Π. Each leaf λ of B_Π represents a node of Π, and the left-to-right order of the leaves of B_Π corresponds to the head-to-tail order of the nodes of Π.

Suppose Π is a solid path. Each leaf λ of B_Π represents a node μ of Π. Each internal node η of B_Π 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 node η four versions of the transfer vector: $h(\eta)$, $\bar{h}(\eta)$, $h^*(\eta)$, $\overline{h^*}(\eta)$, corresponding to no restructuring of T, reversing the parent-child relationship for nodes of Π, reflecting the subtrees rooted at nodes of Π, and both reversing the parent-child relationship and reflecting the subtrees rooted at nodes of Π. If Π is an edge path we keep the value $width(\Pi)$ corresponding to the sum of the widths of the bounding boxes of the tree nodes associated with the nodes of $\Pi(\eta)$. We also keep the *value invariant* that the tail τ of every solid path stores the actual value of $width(\tau)$.

In order to achieve the logarithmic time per dynamic operation, we use biased search trees [1] to represent the path-trees. Each leaf of a biased search tree T is assigned a weight. Biased search trees are structured such that if w is the weight of leaf λ, and W is the sum of the weights of all the leaves of T, the the depth of λ in T will be $O(\log(W/w))$.

For a node ν of a solid path Π, we define $weight(\nu)$ as one plus the sum of the sizes of the subtrees connected to ν by dashed edges. Similarly, for a node $\nu(\mu)$ of edge-path $\Pi_X(\nu)$ ($X = \ell$ or $X = r$), $weight(\nu(\mu))$ is defined to be one plus the total weight of the solid path containing node μ. For each leaf of a path-tree we define its weight equal to the weight of the corresponding path node. Each path tree is then implemented as a search tree biased by the weights of its leaves.

Consider a node μ of T on solid path Π. We keep a boolean value $reversed(\mu)$ indicating which of the neighbors of μ on Π is its parent. We also keep a boolean value $reflected(\mu)$ 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 values $reversed(\eta)$ and $reflected(\eta)$. These offset values are kept such that for any node μ, the true value of $reflected(\mu)$ or $reversed(\mu)$ is the exclusive-or of the values stored at the path tree nodes on the path from μ to the root of the path-tree containing μ.

At each node μ of a path Π, we store pointers $left(\mu)$ and $right(\mu)$ to the left and right neighbors of μ on Π. At each path-tree node η we store pointers to $head(\Pi(\eta))$ and $tail(\Pi(\eta))$. If $reversed(\mu) = 1$, then $left(\mu)$ points to the right neighbor of μ and $right(\mu)$ points to the left neighbor of μ. If $reversed(\eta) = 1$, then $head(\eta)$ points to $tail(\Pi(\eta))$ and $tail(\eta)$ points to $head(\Pi(\eta))$. Also, $reversed(\eta) = 1$ swaps the meaning of the left and right edge-path pointers in the subtree of η. The values of the *reversed* and *reflected* bits determine the actual meaning of the four values kept for h at η.

The following operation moves the values of *reflected* and *reversed* 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 the values *reversed* and *reflected* at nodes η' and η'' with those at η, and set $reversed(\eta) = reflected(\eta) = 0$.

Operation $push(\eta)$ is implemented as follows. If η is an internal node, we exclusive-or the values of *reflected* and *reversed* at node η into the values at η' and η''. If $reversed(\eta) = 1$, we exchange pointers $head(\eta)$ and $tail(\eta)$, values $h(\eta)$ and $\bar{h}(\eta)$, and values $h^*(\eta)$ and $\overline{h^*}(\eta)$. If $reflected(\eta) = 1$, we exchange values $h(\eta)$ and $h^*(\eta)$, and values $\bar{h}(\eta)$ and $\overline{h^*}(\eta)$. We then set $reversed(\eta) = reflected(\eta) = 0$.

Now, suppose η is a path-tree leaf. If $reversed(\eta) = 1$, then we exchange pointers $left(\eta)$ and $right(\eta)$. If Π is a solid path, we also exchange pointers to the left and right edge paths. If $reflected(\eta) = 1$, then we flip the *reflected* bits at the root of any path tree pointed to by node μ. If Π is a solid path, we also flip the *reversed* bits at the root of the path-trees representing the left and right edge-paths of η. We conclude by setting $reversed(\eta) = reflected(\eta) = 0$.

2.4 Query Operations

Operation $Draw(\nu)$ is realized by obtaining a path Π from ν to the root ρ of T. Node ν is the head of path Π. Therefore, by the value invariant, we store the value of $width(\nu)$ at node ν. Additionally, since the reference point of $\Box(\rho)$ is at $(0, 0)$, the transfer vector gives us the actual values of $level(\nu)$, and $reference(\mu)$.

We get path Π using the following operations, derived from dynamic trees [37]:

- *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.
  Suppose node μ' is a sibling of μ such that there is a solid edge from μ' to ν. 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 with respect to Π'. We convert solid edge (μ', ν) to dashed as follows. We begin by splitting path Π' at ν. We then create a path tree node $\nu(\mu')$ for the new dashed edge, and set the left edge-path of ν to be the concatenation $\Pi_\ell(\nu)$, $\nu(\mu)$, and $\Pi_r(\nu)$. The right edge-path becomes the empty path.
  Converting the dashed edge from μ to ν to solid is performed similarly, reversing the roles of the solid and edge-paths. Let Π'' be the path with $\nu = head(\Pi'')$. 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.

- *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*($\lambda$) consists of a sequence of *splice* operations on the solid paths containing the nodes on the path from λ to ρ. This operation is always followed by a *conceal* operation, (see below) 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 [37].

To implement concatenations and splits of the balanced binary trees representing solid paths, we use the following elementary tree operations, each taking $O(1)$ time:

- *join*(*node* ζ', ζ'') — Given the roots ζ' and ζ'' of two binary trees representing 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''. Let σ' and τ' be the head and tail of Π' and σ'' and τ'' be the head and tail of Π''. If paths Π' and Π'' are edge-paths, then we have $weight(\Pi) = weight(\Pi') + weight(\Pi'')$.
  Now, suppose Π' and Π'' are solid paths. We show how to calculate transfer vector $h(\Pi)$ in $O(1)$ time. The calculations for transfer vectors $\bar{h}(\Pi)$, $h^*(\Pi)$, and $\overline{h^*}(\Pi)$ are performed similarly.
  Suppose $h(\Pi) = (a, b, c)$, $h(\Pi') = (a', b', c')$ and $h(\Pi'') = (a'', b'', c'')$ are the transfer vectors for paths Π, Π' and Π''. Suppose w_ℓ and w_r are the total width of the left and right edge paths of node σ''. We calculate $h(\Pi)$ in the following manner:

$$\begin{aligned} width(\tau'') &= width(\sigma'') + a'' \\ width(\tau'') &= (width(\tau') + w_\ell + w_r) + a'' \\ width(\tau'') &= ((width(\sigma') + a') + w_\ell + w_r) + a'' \end{aligned}$$

$$\begin{aligned} level(\sigma') &= level(\tau') + b' \\ level(\sigma') &= (level(\sigma'') + 1) + b' \\ level(\sigma') &= ((level(\tau'') + b'') + 1) + b' \end{aligned}$$

$$\begin{aligned} reference(\sigma') &= reference(\tau') + c' \\ reference(\sigma') &= (reference(\sigma'') + w_\ell) + b' \\ reference(\sigma') &= ((reference(\tau'') + c'') + w_\ell) + c' \end{aligned}$$

Therefore, we have:

$$\begin{aligned} a &= a' + a'' + w_\ell + w_r \\ b &= b' + b'' + 1 \\ c &= c' + c'' + w_\ell \end{aligned}$$

- *separate*(*node* ζ) — Given the root ζ of a binary tree, 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 (a, b) pairs of ζ' and ζ'' do not change.
- *rotateleft*(*node* η) (*rotateright*(*node* η)) — Perform a left (right) rotation at node η. This operation can be performed with $O(1)$ *separate* and *join* operations.

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 $Draw(\nu)$ is then performed by first issuing $expose(\nu)$ to get path Π. By the value invariant, node ν will store the value of $width(\nu)$. The transfer vector for Π contains the value of $level(\nu)$ and $reference(\nu)$. Therefore, the y-coordinate returned is $level(\nu)$ and the x-coordinate is $reference(\nu) + (width(\nu)/2)$. Operation $Offset(\nu)$ is realized similarly: we first call $expose(\nu)$. The x-coordinate of the reference point of $\Box(\nu)$ is $reference(\nu)$ and the y value is $level(\nu)$. 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.

In order to implement operation *Window*, we introduce selection functions, which are used to find distinguished nodes of a path or tree. We will use this structure in the implementation of operations in this and the next section.

A *path-selection function* S maps a path Π and a query argument q into a node $\mu = S(\Pi, q)$ of Π, such that if Π is the concatenation of Π' and Π'', then one can determine in $O(1)$ time whether μ is in Π' or Π'' from q and the transfer values stored for Π.

A *tree-selection function* S maps a tree T and a query argument q into a node $\mu = S(T, q)$ of T. Function S is a path-selection function. If node μ is a descendant of the tail of path Π of tree T, then $find(\Pi, S, q)$ returns the deepest node μ' on Π such that μ is a descendant of μ'.

We then use the following operations to find the distinguished nodes:

- *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.

Operations *PathFind* and *TreeFind* use the following function which returns the distinguished node of a path:

- *find*(*path* Π, *selectionfunction* S, *value* q) — Find node μ of Π returned by path-selection function S with query argument q.

We need the following operation from [38] in our implementation of $find(\Pi, S, q)$:

- *splaystep*(*node* η) — For binary tree B with root ζ and node η a grandchild of ζ, restructure B such that the relative order of the leaves of 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' and η are both left children then perform $rotateright(\zeta)$ twice. If η' is a left child and η is a right child, then perform $rotateleft(\eta')$ followed by $rotateright(\zeta)$. The other two cases are symmetric. (See figure 2.)

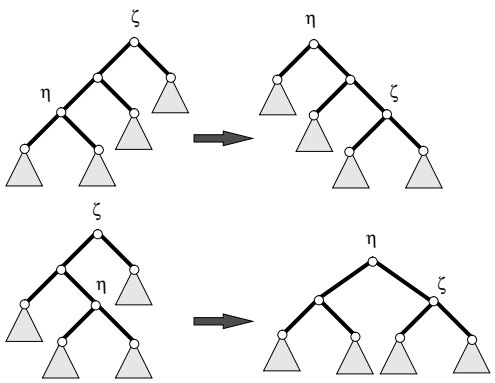

Figure 2: The rotations performed in operation *splaystep*(η).

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

Lemma 3 *Suppose S is a path-selection function 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 B_Π 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.

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.

Assume node μ' is a tree node. Let path Π'' be the result of joining the left edge path $\Pi_\ell(\mu')$ and Π'. This is well defined by the expansion invariant. By the definition of tree and path selection functions, we can determine in $O(1)$ time if node μ is in $\Pi_\ell(\mu')$. If not, repeat using the right edge path $\Pi_r(\mu')$.

If neither edge path has μ as a descendant, then $\mu' = \mu$. Otherwise, we reset path Π to be Π'' and continue. If μ' is an edge path node, we proceed similarly, except we join the

solid-path associated with μ' to Π'. 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 B_Π, 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 $expose(\mu)$. The time to restore the path trees is equivalent. Therefore operation *TreeFind* is implemented in $O(\log n)$ time.

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 $locatepoint(\nu, p)$ 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 $drawproper(\mu, p, q)$ draws (at least) a portion of an edge.

Operation $locatepoint(\nu, p)$ first calls $expose(\nu)$ 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_1 *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 $TreeFind(\nu, S_1, p)$ 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 $\mu = TreeFind(\nu, S_1, p)$. 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 $drawproper(\mu, p, q)$ by first calling $expose(\mu)$. 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_2 *Suppose η is the root of a path tree with children η' and η''. Let μ' be the child of μ associated with* $\text{tail}(\eta')$. *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 $\mu_\ell = \mathit{PathFind}(\Pi_\ell(\mu), S_2, (p,q))$. 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_3 *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, 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_4 *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_4 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 $\mathit{findrightint}(\nu,p,q)$ is then implemented as follows. Let node $\mu = \mathit{TreeFind}(\nu, S_4, x_p)$. 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 $\mathit{findleftint}(\nu,p,q)$ 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 $\mathit{Window}(\nu,p,q)$ by performing the following steps.

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

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

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

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 μ = *TreeFind*(ν, 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*(ν, 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 μ = *locatepoint*(ν, s). We call *drawtree*(μ, 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*(ν, p, q) is implemented by performing the following steps.

1. Perform *drawproper*(ν, 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*(ν) is to the left (right) of W. Let $\mu = findrightint(\nu, p, q)$ ($\mu = findleftint(\nu, p, q)$). If μ = *nil* then stop. Otherwise call *drawtree*(μ, p, q)
4. If step 3 did draw edges then let $y_p = depth(\nu) + 1$ and call *Window*(μ, 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.

2.5 Update Operations

Operations *MakeGraph* and *DeleteGraph* can be trivially implemented in $O(1)$ time. Operations *Link*, and *Cut* can each be implemented as variations of the dynamic tree operations *Link* and *Cut*.

Operation *Evert*(μ) consists of issuing *expose*(μ), flipping the reverse bit of the resulting solid path, then restoring the size-invariant by calling *conceal*(μ). This is implemented on $O(\log n)$ 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 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 *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 edges preceding edge (μ', μ), Π'' containing the edges between edges (μ', μ) and (μ'', μ), and Π''' containing the edges following edge (μ'', μ). Paths Π' and Π''' become the left and right edge-paths of μ. We create a new node ν with left edge-path Π'' and right edge-path the empty path. We extend the solid path containing μ by concatenating a new node ν. Finally, we call *conceal*(ν) to restore the size invariant.

Operation *Contract*(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. Let node ν be the parent of μ. Next, 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.

3 Series Parallel Digraphs

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

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

A series-parallel digraph G is associated with a rooted tree T, called *SPQ-tree.* A node $\mu \in T$ represents a subgraph of G, called the *pertinent digraph* of μ, and denoted by 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 digraphs of the children of μ. The nodes of T are of four types: S-nodes, P-nodes, $\mathrm{P_Q}$-nodes, and Q-nodes. Tree T is defined recursively as follows:

- If G is a single edge, then T consists of a single *Q-node.*
- If G is the parallel composition of series-parallel digraphs $G_1, \ldots, G_k$ with SPQ-trees $T_1, \ldots, T_k$ with roots $\rho_1, \ldots, \rho_k$, and none of the G_i, $1 \leq i \leq k$ is a single edge, then T consists of a *P-node* root with children $\rho_1, \ldots, \rho_k$.
- 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.

- If G is the series composition of series-parallel digraphs $G_1, \ldots, G_k$ with SPQ-trees $T_1, \ldots, T_k$ with roots $\rho_1, \ldots, \rho_k$, then T consists of an *S-node* root with children $\rho_1, \ldots, \rho_k$.

We keep two types of nodes representing parallel composition since transitive edges are handled differently from general parallel composition. Clearly, we can represent the right-pushed embedding of any series-parallel digraph using an SPQ-tree.

The *type* of a node ν is either Q, P, $\mathrm{P_Q}$, or S. We keep the type invariant that each node of an SPQ-tree T does not have a child of the same type, and that a P-node cannot have $\mathrm{P_Q}$-node as a child. If G has m edges, then T has $O(m)$ nodes. Tree T can be constructed in $O(m)$ time using the recognition algorithm of [50].

Let ν be an S-node with children $\mu_1, \ldots, \mu_k$. The *skeleton* of μ, denoted *skeleton*(μ), is the series-parallel digraph consisting of k edges $e_i = (v_{i-1}, v_i)$, $1 \leq i \leq k$, where v_0 and v_k are the source and sink of the pertinent digraph of ν, and for $1 \leq i < k$, v_i is the sink of the pertinent digraph of μ_i. The *proper node* of vertices $v_1, \ldots, v_{k-1}$ is node ν. Note that v_i is a join-vertex used in the series composition at its proper node. Hence, if G is a series-parallel digraph with associated SPQ-tree T then each vertex of G, with the exception of its poles, has a proper node.

Given a digraph G, the *reverse* digraph $\bar{G}$ of G is formed by reversing the orientation of all edges of G. It is easy to see that if G is a series-parallel digraph, then $\bar{G}$ is also a series-parallel digraph. If s and t are the source and sink of G, respectively, then t and s are the source and sink of $\bar{G}$, respectively. If G is the parallel composition of series-parallel digraphs $G_1, \ldots, G_k$, then $\bar{G}$ is the parallel composition of series-parallel digraphs $\overline{G_1}, \ldots, \overline{G_k}$. Similarly, if G is the series composition of series-parallel digraphs $G_1, \ldots, G_k$, then $\bar{G}$ is the series composition of series-parallel digraphs $\overline{G_k}, \ldots, \overline{G_1}$.

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.

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.

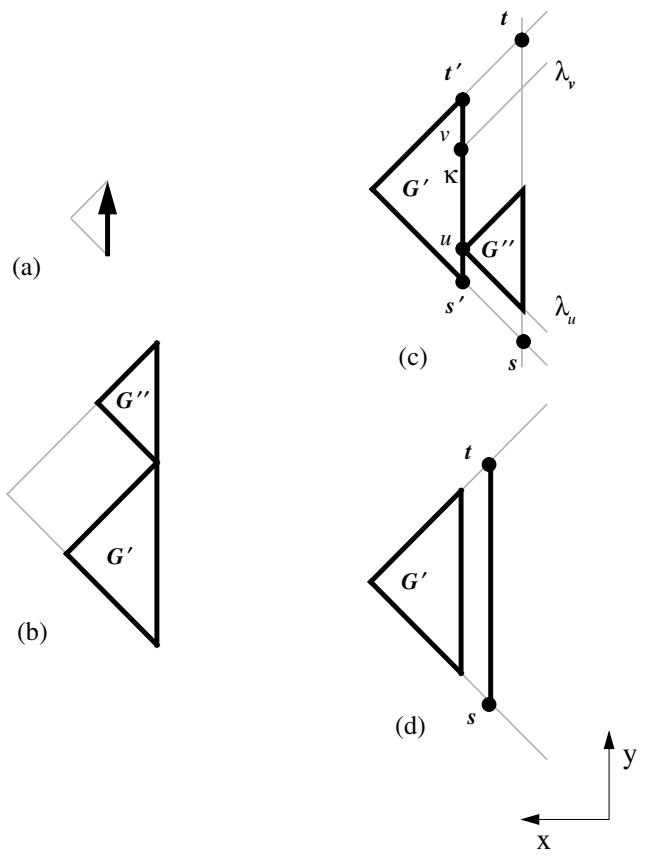

Figure 3: Geometric construction of a Δ-drawing: (a) base case; (b) series composition; (c) parallel composition (general case); (d) parallel composition with the "right-pushed" transitive edge.

It is important to note that in order to get polynomial area the embedding cannot be completely preserved. Namely, it is shown in [2] 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 [2] 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 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 [2].

- 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. 3.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. 3.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. 3.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 4 [2] *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.*

3.2 Dynamic Environment

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 the Δ-algorithm on a collection $\mathcal{G}$ of series-parallel digraphs. Namely, we introduce the following operations:

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 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.*

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:

- $width(\mu)$ — The width of $\Delta(\mu)$. Note that the height of $\Delta(\mu)$ will be $2 \cdot width(\mu)$.
- $position(\mu)$ — 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 $(0, 0)$:

- $sourceright(\mu)$ — The location of the drawing of the vertex connected to the rightmost outgoing edge from the source of G_μ.
- $sourceleft(\mu)$ — The location of the drawing of the vertex connected to the leftmost outgoing edge from the source of G_μ.
- $sinkright(\mu)$ — The location of the drawing of the vertex connected to the rightmost incoming edge to the sink of G_μ.
- $sinkleft(\mu)$ — 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 2, 3, 4, and 5. As an example, Fig. 4 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 $position(\mu) = (0, 0)$.

Lemma 5 *Consider a node* μ *and an ancestor* ν *in SPQ-tree* T*. Then* $\text{width}(\nu)$ *can be expressed as* $\text{width}(\mu) + d$ *for some constant* d*. The value* $y(\text{sourceright}(\nu))$ *can be expressed*

$$
\begin{aligned}
width(\mu) &= \sum_{i=1}^{k} width(\mu_i) \\
x(position(\mu_j)) &= x(position(\mu)) + \sum_{i=j+1}^{k} width(\mu_i) \\
y(position(\mu_j)) &= y(position(\mu)) + \sum_{i=j+1}^{k} width(\mu_i) + \sum_{i=1}^{j-1} y(sourceright(\mu_i)) \\
x(sourceright(\mu)) &= 0 \\
y(sourceright(\mu)) &= \sum_{i=1}^{k} y(sourceright(\mu_i)) \\
x(sourceleft(\mu)) &= \sum_{i=2}^{k} width(\mu_i) + x(sourceleft(\mu_1)) \\
y(sourceleft(\mu)) &= \sum_{i=2}^{k} width(\mu_i) + y(sourceleft(\mu_1)) \\
x(sinkright(\mu)) &= 0 \\
y(sinkright(\mu)) &= \sum_{i=1}^{k-1} y(sourceright(\mu_i)) + y(sinkright(\mu_k)) \\
x(sinkleft(\mu)) &= \sum_{i=2}^{k} width(\mu_i) + x(sinkleft(\mu_1)) \\
y(sinkleft(\mu)) &= \sum_{i=2}^{k} width(\mu_i) + y(sinkleft(\mu_1))
\end{aligned}
$$

Table 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.

$$
\begin{aligned}
sourceright(\mu) &= (0, 2 \cdot width(\mu)) \\
sinkright(\mu) &= (0, 0)
\end{aligned}
$$

Table 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 2).

as:

$$a \cdot \text{width}(\mu) + b \cdot y(\text{sourceright}(\mu)) + c$$

for some a, b, c. The other values $y(\text{sourceleft}(\nu))$, $y(\text{sinkright}(\nu))$, and $y(\text{sinkleft}(\nu))$ can be expressed similarly. Additionally, the value of $y(\text{position}(\mu))$ can be expressed as:

$$y(\text{position}(\nu)) + d'$$

for some constant d'. Similar equations hold for x-values.

$$
\begin{aligned}
width(\mu) &= \sum_{i=1}^{k} width(\mu_i) \\
x(position(\mu_j)) &= x(position(\mu)) \\
y(position(\mu_j)) &= y(position(\mu)) + 2 \cdot \sum_{i=1}^{j-1} width(\mu_i) \\
sourceright(\mu) &= sourceright(\mu_1) \\
sourceleft(\mu) &= sourceleft(\mu_1) \\
x(sinkright(\mu)) &= 0 \\
y(sinkright(\mu)) &= 2 \cdot \sum_{i=1}^{k-1} width(\mu_i) + y(sinkright(\mu_k)) \\
x(sinkleft(\mu)) &= x(sinkleft(\mu_k)) \\
y(sinkleft(\mu)) &= 2 \cdot \sum_{i=1}^{k-1} width(\mu_i) + y(sinkleft(\mu_k))
\end{aligned}
$$

Table 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.

$$
\begin{aligned}
width(\mu) &= 1 \\
sourceright(\mu) &= (0,2) \\
sourceleft(\mu) &= (0,2) \\
sinkright(\mu) &= (0,0) \\
sinkleft(\mu) &= (0,0)
\end{aligned}
$$

Table 5 The values of *width*, *sourceright*, *sourceleft*, *sinkright*, and *sinkleft* for a Q-node μ.

Proof: By induction on the length of the path from node μ to node ν. If $\mu = \nu$ then the lemma is trivially true. The inductive step follows from the equations of tables 2, 3, 4, and 5. □

For a solid or edge-path Π with head σ and tail τ, the transfer vector for SPQ-trees $g(\Pi)$ a vector of length 20 containing the constants in the equations of lemma 5 associated with the calculations of the values *width*, *position*, *sourceright*, *sourceleft*, *sinkright*, and *sinkleft*.

We structure SPQ-tree in the same manner as in section 2, decomposing T into solid and edge paths. Suppose η is a path-tree node. If η is in the representation of a solid-path, we store at η the transfer vector $h(\eta)$. We do not evert or reflect SPQ-trees, so we do not keep multiple versions of the transfer vector. If η is part of the representation of an edge-path, we store at η the value $width(\eta)$ corresponding to the sum of the widths of the bounding triangles of the tree nodes associated with the nodes of $\Pi(\eta)$. We can maintain the transfer vectors under operation *join* in $O(1)$ time. This is immediate from the equations in tables 2, 3, 4, and 5.

We keep the following additional values for each solid path Π from node σ to node τ.

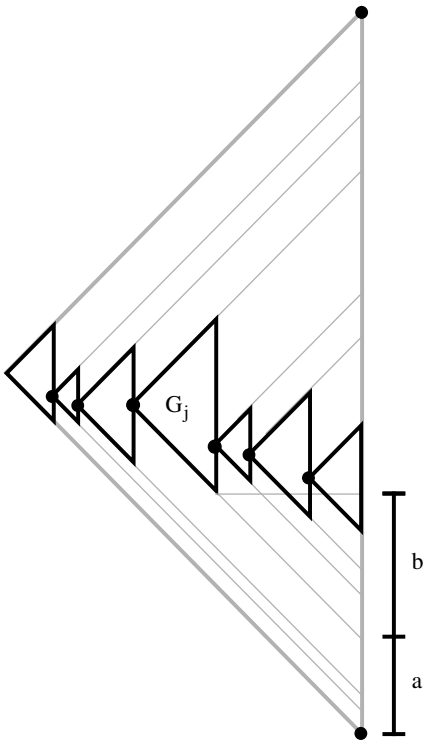

Figure 4: A pictoral representation of the calculations of $position(\mu_j)$, where μ_j is a child of a P-node μ. Graph G_j is the pertinent graph of μ_j. Notice that lengths $a = \sum_{i=1}^{j-1} y(sourceright(\mu_i))$ and $b = \sum_{i=j+1}^{k} width(\mu_i)$.

- $samesource(\Pi)$ — **true** if and only the source of G_σ is the same vertex as the source of G_τ.
- $samesink(\Pi)$ — **true** if and only the sink of G_σ is the same vertex as the sink of G_τ.
- $noleftp(\Pi)$ — **true** if and only if path Π does not contain both a P-node and its leftmost child.

We use the following operation to find proper nodes:

- $Proper(vertex\ v)$ — Returns the node triple (μ, μ', μ''), where μ is the proper node of v, node μ' is the child of ν such that v is the sink of G'_μ, and μ'' is the child of ν such that v is the source of G''_μ. If v is a pole of G, then $Proper(v)$ returns $(\rho, -, -)$, where ρ is the root of T.

We use the following path selection function to find $Proper(v)$. Selection function S_5 takes as an argument integer x which has value either 1 or 2 indicating if v is the source or sink of the pertinent digraph of the head of $\Pi(\eta)$.

Selection Function S_5 *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* samesource(Π') = **true** *or if $x = 2$ and* samesink(Π') = **true** *then return η''. Else return η'.*

When used with operation *PathFind*, path selection function S_5 returns the tail of the longest subpath beginning at $head(\eta)$ such that pole of $head(\eta)$ indicated by x is also a pole

of the returned node. Therefore, the parent of the returned node will be the proper node of v.

If v is the source or sink of G, then return $(\rho, -, -)$, where ρ is the root of T. Otherwise, let e be any edge such that v is the source of e and let node λ be the associated Q-node. We find node μ'' by calling $PathFind(\lambda, S_5)$. Nodes μ and μ' are the parent and left sibling of μ''. This takes $O(\log m)$ time.

3.4 Query Operations

Operation $Offset(\nu)$ is implemented as for trees. We call $expose(\nu)$, return the value of $position(\nu)$, then restore the size invariant by calling $conceal(\nu)$.

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 4).

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', $R_s(\kappa) =$

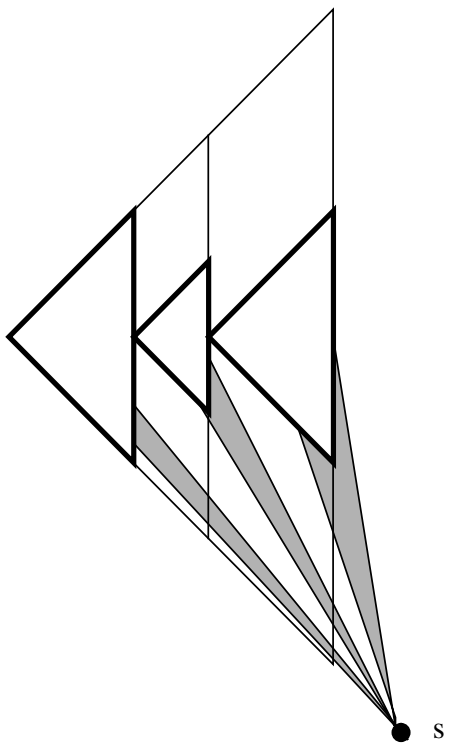

Figure 5: Finding the face containing a query point p. If p is in a shaded region then its face is found at a descendant P-node.

> $R_s(\kappa')$. 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).

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_6 *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 $\kappa = \mathit{TreeFind}(\nu, S_6, p)$. 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_7 *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 $\kappa'' = \mathit{PathFind}(\Pi, S_7)$. 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.

Selection Function S_8 *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 $\kappa = \mathit{TreeFind}(\mu, S_8, p)$. 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 [42] 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 $\mathit{Locate}(\nu, s)$. 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.

3.5 Update Operations

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 $\mathit{InsertVertex}(v, e, e', e'')$, 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 $\mathit{DeleteVertex}(e, v, e', e'')$ is implemented similarly with a constant number of *MakeDigraph*, *DeleteDigraph*, *Link*, *Cut*, and *Expand* operations.

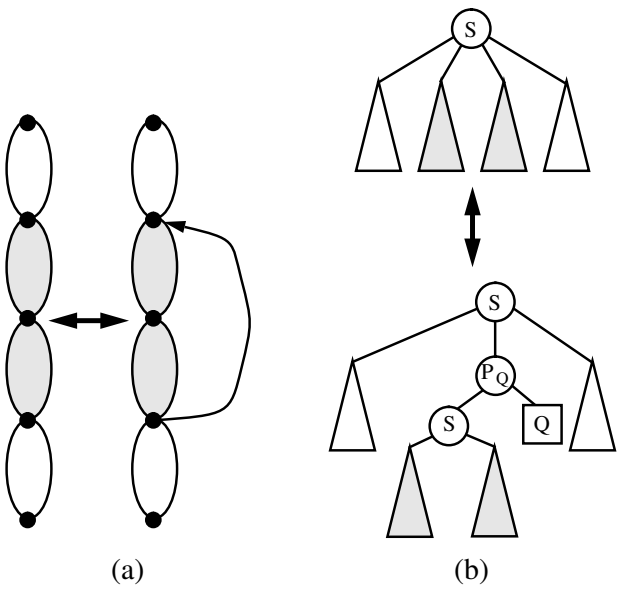

Figure 6: Modification of T in operation *InsertEdge* and *DeleteEdge*: splitting/merging an S-chain. (a) Insertion/deletion of an edge between nonconsecutive vertices of the skeleton of an S-chain. (b) Transformation of the SPQ-tree.

The restructuring of the SPQ-tree caused by *InsertEdge* and *DeleteEdge* is more complex. It is known [5] 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 6 and 7. 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 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 $G_{\mu''}$.

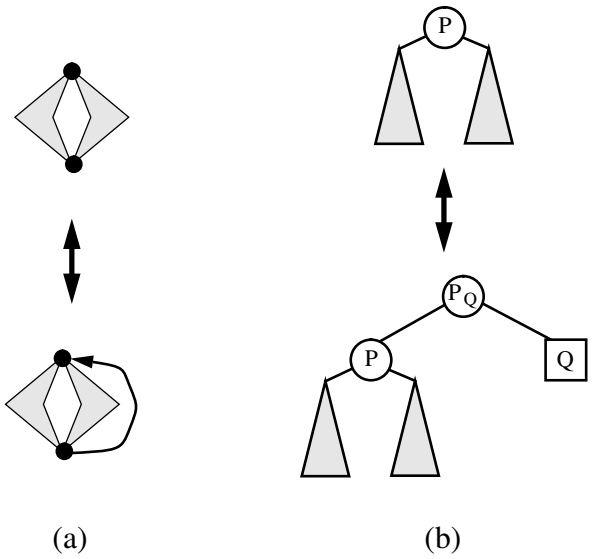

Figure 7: Modification of T in operation *InsertEdge* and *DeleteEdge*: extending/shortening a P-chain. (a) Insertion/deletion of an edge between consecutive vertices of the skeleton of an S-chain (the vertices are the source and sink of a parallel composition). (b) Transformation of the SPQ-tree.

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 [27] in connection with a planarity testing algorithm, and they have subsequently been used in several applications, including planar graph embedding [4,8,42], graph drawing [7,9], and planar point location [16, 21,32,48].

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 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.

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 [9], 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 [9].

4.1.1 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 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 [9,43], 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 balanced binary trees T_L and T_R (e.g., using red-black trees [22]). Each leaf represents a vertex, edge, or face. At each leaf, we keep the following binary value: 1 if the node is associated with a vertex or a transitive edge, and 0 otherwise. At each internal node η, we store the sum of the values at the leaves in the subtree of η. In a drawing query, we compute $x(v)$ (resp., $y(v)$) by splitting tree T_L (resp., T_R) at the node associated with v, and finding the value stored at the root of the resulting left tree.

After an update operation at most two 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 values can be done in $O(\log n)$ time. Also, sequences L and R are updated by means of $O(1)$ split/concatenate operations [43], so that the corresponding updates on T_L and T_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.

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 [31], VLSI layout [24,35,51], motion planning [23], and graph drawing [7,45]. 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 [12,44,46,47,52,53].

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 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 [44]. 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 [7,34,44].

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 4.2. The restriction of sequence L (or R) to V is a topological ordering [43]. Hence, we can use balanced binary trees 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.

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 [8], 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 *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 [42].

We can extend Theorem 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 [8]. 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 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 Open Problems

Open problems include:

- Decrease 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 [19,36] appear difficult to dynamize.

- Dynamically maintain planar orthogonal drawings with the minimum number of bends. The static algorithm of [41] 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 [3] and for single-source digraphs [25]. Semidynamic planarity testing can be done with $O(\log n)$ query and insertion time [8]. Recently, a fully dynamic planarity testing technique with $O(n^{2/3})$ query and update time has been discovered [20].
- Dynamize drawing methods for general graphs that are based on physical models of the layout process, such as the "spring" algorithm [13,26].

Acknowledgement

We are grateful to Paola Bertolazzi for helpful discussions and insights.

References

[1] S.W. Bent, D.D. Sleator, and R.E. Tarjan, "Biased Search Trees," *SIAM J. Computing* 14 (1985), 545–568.

[2] P. Bertolazzi, R.F. Cohen, G. Di Battista, R. Tamassia, and I.G. Tollis, "How to Draw a Series-Parallel Digraph," *Proc. SWAT* (1992).

[3] P. Bertolazzi and G. Di Battista, "On Upward Drawing Testing of Triconnected Digraphs," *Proc. ACM Symp. on Computational Geometry* (1991).

[4] 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.

[5] R.F. Cohen and R. Tamassia, "Dynamic Expression Trees and their Applications," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1991), 52–61.

[6] G. Di Battista, W.-P. Liu, and I. Rival, "Bipartite Graphs, Upward Drawings, and Planarity," *Information Processing Letters* 36 (1990), 317–322.

[7] G. Di Battista and R. Tamassia, "Algorithms for Plane Representations of Acyclic Digraphs," *Theoretical Computer Science* 61 (1988), 175–198.

[8] G. Di Battista and R. Tamassia, "Incremental Planarity Testing," *Proc. 30th IEEE Symp. on Foundations of Computer Science* (1989), 436–441.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] P. Eades, "A Heuristic for Graph Drawing," *Congressus Numerantium* 42 (1984), 149–160.

[14] P. Eades and X. Lin, "How to Draw Directed Graphs," *Proc. IEEE Workshop on Visual Languages (VL'89)* (1989), 13–17.

[15] 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.

[16] H. Edelsbrunner, L.J. Guibas, and J. Stolfi, "Optimal Point Location in a Monotone Subdivision," *SIAM J. Computing* 15 (1986), 317–340.

[17] 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.

[18] 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.

[19] H. de Fraysseix, J. Pach, and R. Pollack, "How to Draw a Planar Graph on a Grid," *Combinatorica* 10 (1990), 41–51.

[20] Z. Galil and G.F. Italiano, "Fully Dynamic Planarity Testing," *Proc. ACM Symp. on Theory of Computing* (1992).

[21] M.T. Goodrich and R. Tamassia, "Dynamic Trees and Dynamic Point Location," *Proc. 23th ACM Symp. on Theory of Computing* (1991), 523–533.

[22] L.J. Guibas and R. Sedgewick, "A Dichromatic Framework for Balanced Trees," *Proc. 19th IEEE Symp. on Foundations of Computer Science* (1978), 8–21.

[23] 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.

[24] 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.

[25] M.D. Hutton and A. Lubiw, "Upward Planar Drawing of Single Source Acyclic Digraphs," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1991), 203–211.

[26] T. Kamada, *Visualizing Abstract Objects and Relations*, World Scientific, Teaneck, NJ, 1989.

[27] 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.

[28] S.M. Malitz and A. Papakostas, "On the Angular Resolution of Planar Graphs," *Proc. ACM Symp. on Theory of Computing* (1992).

[29] S. Moen, "Drawing Dynamic Trees," *IEEE Software* 7 (1990), 21–28.

[30] T. Nishizeki and N. Chiba, *Planar Graphs: Theory and Algorithms*, Annals of Discrete Mathematics 32, North-Holland, 1988.

[31] J. O'Rourke, *Art Gallery Theorems and Algorithms*, Oxford University Press, 1987.

[32] F.P. Preparata and R. Tamassia, "Fully Dynamic Point Location in a Monotone Subdivision," *SIAM J. Computing* 18 (1989), 811–830.

[33] E. Reingold and J. Tilford, "Tidier Drawing of Trees," *IEEE Trans. on Software Engineering* SE-7 (1981), 223–228.

[34] P. Rosenstiehl and R.E. Tarjan, "Rectilinear Planar Layouts of Planar Graphs and Bipolar Orientations," *Discrete & Computational Geometry* 1 (1986), 343–353.

[35] 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.

[36] W. Schnyder, "Embedding Planar Graphs on the Grid," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1990), 138–148.

[37] D.D. Sleator and R.E. Tarjan, "A Data Structure for Dynamic Trees," *J. Computer Systems Sciences* 24 (1983), 362–381.

[38] D.D. Sleator and R.E. Tarjan, "Self-Adjusting Binary Search Trees," *J. ACM* 32 (1985), 652–686.

[39] 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.

[40] K.J. Supowit and E.M. Reingold, "The Complexity of Drawing Trees Nicely," *Acta Informatica* 18 (1983), 377–392.

[41] R. Tamassia, "On Embedding a Graph in the Grid with the Minimum Number of Bends," *SIAM J. Computing* 16 (1987), 421–444.

[42] R. Tamassia, "A Dynamic Data Structure for Planar Graph Embedding," *Proc. 15th ICALP,* LNCS 317 (1988), 576–590.

[43] R. Tamassia and F.P. Preparata, "Dynamic Maintenance of Planar Digraphs, with Applications," *Algorithmica* 5 (1990), 509–527.

[44] R. Tamassia and I.G. Tollis, "A Unified Approach to Visibility Representations of Planar Graphs," *Discrete & Computational Geometry* 1 (1986), 321–341.

[45] R. Tamassia and I.G. Tollis, "Planar Grid Embedding in Linear Time," *IEEE Trans. on Circuits and Systems* CAS-36 (1989), 1230–1234.

[46] R. Tamassia and I.G. Tollis, "Tessellation Representations of Planar Graphs," *Proc. 27th Annual Allerton Conf.* (1989), 48–57.

[47] R. Tamassia and I.G. Tollis, "Representations of Graphs on a Cylinder," *SIAM J. on Discrete Mathematics* 4 (1991), 139–149.

[48] R. Tamassia and J.S. Vitter, "Parallel Transitive Closure and Point Location in Planar Structures," *SIAM J. Computing* 20 (1991), 708–725.

[49] C. Thomassen, "Planar Acyclic Oriented Graphs," *Order* 5 (1989), 349–361.

[50] J. Valdes, R.E. Tarjan, and E.L. Lawler, "The Recognition of Series Parallel Digraphs," *SIAM J. on Computing* 11 (1982), 298–313.

[51] S. Wimer, I. Koren, and I. Cederbaum, "Floorplans, Planar Graphs, and Layouts," *IEEE Trans. on Circuits and Systems* 35 (1988), 267–278.

[52] S.K. Wismath, "Characterizing Bar Line-of-Sight Graphs," *Proc. ACM Symp. on Computational Geometry* (1985), 147–152.

[53] S.K. Wismath, "Weighted Visibility Graphs of Bars and Related Flow Problems," *Algorithms and Data Structures (Proc. WADS'89)* (1989), 325–334.