Dynamic Expression Trees

Robert F. Cohen and Roberto Tamassia

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-90-35
December 1990

Dynamic Expression Trees *

Robert F. Cohen Roberto Tamassia
Department of Computer Science
Brown University
Providence, Rhode Island 02912–1910
(rfc@cs.brown.edu, rt@cs.brown.edu)

Abstract

We present a technique for dynamically maintaining a collection of arithmetic expressions represented by binary trees (whose leaves are variables and whose internal nodes are operators). A query operation asks for the value of an expression (associated with the root of a tree). Update operations include changing the value of a variable and combining or decomposing expressions by linking or cutting the corresponding trees. Our dynamic data structure uses linear space and supports queries and updates in logarithmic time. An important application is the dynamic maintenance of maximum flow and shortest path in series-parallel digraphs under a sequence of vertex and edge insertions, series and parallel compositions, and their respective inverses. Queries include reporting the maximum flow or shortest *st*-path in a series-parallel subgraph.

(December 1991)

*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, and by Cadre Technologies, Inc.

1 Introduction

The evaluation of tree-structured expressions is a fundamental computational task. In this paper we study the dynamic maintenance of expressions over a semiring $\mathcal{S}$ with binary operators $\oplus$ and $\otimes$, such that $\otimes$ is distributive with respect to $\oplus$. We also allow the inverse operators of $\oplus$ and $\otimes$, whenever they exist. An expression over $\mathcal{S}$ is naturally represented by a binary tree $\mathcal{T}$ where the internal nodes represent operators and the leaves represent variables. The important classes of arithmetic and boolean expressions are special cases of this general model.

We consider the problem of dynamically maintaining a collection of expressions of total size n. A query operation asks for the value of an expression (or a subexpression associated with a subtree). The repertory of dynamic operations consists of: (a) changing the value of a variable; (b) combining two expressions by linking the corresponding trees; and (c) decomposing an expression into two parts by cutting its tree. In Section 2 we present an efficient solution to this problem by introducing a new data structure, called *dynamic expression tree*, that uses $O(n)$ space and supports each query or update operation in $O(\log n)$ time.

Incremental graph evaluation has been extensively studied in the area of programming languages. However, the most advanced techniques previously developed, when applied to our problem, still yield $O(n)$ worst-case time per update operation (see, e.g., [2,21]).

Our dynamic expression trees are inspired by the dynamic trees of Sleator and Tarjan [23] and the expression restructuring scheme of Brent [6,7]. We show how to combine these classical techniques to yield a powerful and versatile dynamic data structure.

Related work has been focused on the use of parallelism in expression evaluation (see [18]). The most efficient parallel "tree-contraction" algorithms run in time $O(\log n)$ using an EREW PRAM with $n/\log n$ processors.

An important application of dynamic expression trees is the dynamic maintenance of the maximum flow and shortest path in series-parallel digraphs. Such digraphs arise in a variety of problems such as scheduling [19], electrical networks (see, e.g., [30]), data-flow analysis [8], database logic programs [1], and circuit layout [25]. Also, some problems that are NP-complete for general graphs admit linear-time solutions when restricted to series-parallel digraphs [26].

We define an extensive repertory of update operations for series-parallel digraphs, which includes insertion of vertices and edges, series and parallel composition of two series-parallel digraphs, and their respective inverses. Query operations include finding the value of a maximum flow (or the length of a shortest source-to-sink path), and determining whether an edge is on some minimum cut (or shortest source-to sink path). In a static environment maximum flow and shortest source-to-sink path in a series-parallel digraph G can be computed in $O(m)$ time, where m is the current number of edges of G. In Section 3 we present an $O(m)$ space dynamic data structure based on dynamic expression trees that supports each query or update operation in time $O(\log m)$.

Maximum flow and shortest path are fundamental problems in network optimization and have been extensively investigated [9,14]. A long standing question has been the development of efficient dynamic algorithms for such problems.

The dynamic data structures for all-pairs shortest paths presented in [12,22] have $O(n^2)$ space, $O(1)$ query time, $O(n^2)$ time for edge insertion, and $O(mn + n^2 \log n)$ time for edge deletion, where n is the number of vertices and m is the number of edges. Such performance bounds, especially for deletion, are quite unsatisfactory and indicate the difficulty of the problem (for lower bound arguments, see [5,24]). The best known semi-dynamic data structure supporting insertions in digraphs with unit edge lengths uses $O(n^2)$ space and has $O(1)$ query time; the total time to process all edge insertions is $O(n^3 \log n)$, which amortizes to $O(n \log n)$ time per insertion for dense digraphs [3,20].

Regarding maximum flow, previous results are confined to the *parametric maximum flow problem*, where the capacities of the edges are functions of a parameter λ. In [13,16] it is shown that the value of a maximum flow for $O(n)$ values of λ can be computed on-line with the same time complexity as the fastest maximum flow algorithm. No efficient algorithms are known for dynamic or semi-dynamic flow problems where update operations change arbitrarily edge capacities or insert/delete edges. Hence, our results give the first efficient technique for maintaining a maximum flow, although limited to the class of series-parallel digraphs.

Further applications of dynamic expression trees to constructive solid geometry and layout compaction are given in Section 4. Finally, open problems are discussed in Section 5.

2 Dynamic Expression Trees

Let $\mathcal{S}$ be a semiring with binary operators $\oplus$ and $\otimes$ such that $\otimes$ is distributive with respect to $\oplus$. For simplicity, we assume that variables can be stored in $O(1)$ space and binary operations can be performed in $O(1)$ time. The extension to the general case is straightforward. An expression $\mathcal{E}$ over $\mathcal{S}$ is represented by a binary tree $\mathcal{T}$, whose internal nodes are the operators and whose leaves are the variables of $\mathcal{E}$ (see Fig. 1).

Our dynamic environment supports the following operations on a collection of expressions (see Fig. 2).

- *Evaluate*(*node* μ) — Return the value of the subexpression associated with the subtree rooted at μ.
- *Change*(*node* λ; *value* x) — This operation assumes that λ is a leaf. Set the value of the variable associated with λ equal to x.
- *MakeVar*(*value* x) — Create a new tree consisting of a single leaf whose variable has value x.
- *DeleteVar*(*node* λ) — Remove the single-node tree consisting of leaf λ.
- *Construct*(*operator* $\odot$; *node* ρ', ρ'') — This operation assumes that ρ' and ρ'' are the roots of two trees $\mathcal{T}'$ and $\mathcal{T}''$, respectively. Create a new tree $\mathcal{T}$ whose root ρ stores operator $\odot$, and with left subtree $\mathcal{T}'$ and right subtree $\mathcal{T}''$.
- *Destruct*(*node* ρ) — This operation assumes that ρ is the root of a tree $\mathcal{T}$ and is not a leaf. Remove node ρ, which separates $\mathcal{T}$ into trees $\mathcal{T}'$ and $\mathcal{T}''$ given by the former left and right subtrees of $\mathcal{T}$, respectively.
- *Graft*(*node* ρ, λ) — This operation assumes that λ is a leaf of a tree $\mathcal{T}$, and ρ is the root of a tree $\mathcal{T}'$. Replace λ with ρ, so that $\mathcal{T}'$ becomes a subtree of $\mathcal{T}$.
- *Prune*(*node* μ) — Let $\mathcal{T}$ be the tree containing node μ. Detach from $\mathcal{T}$ the subtree $\mathcal{T}'$

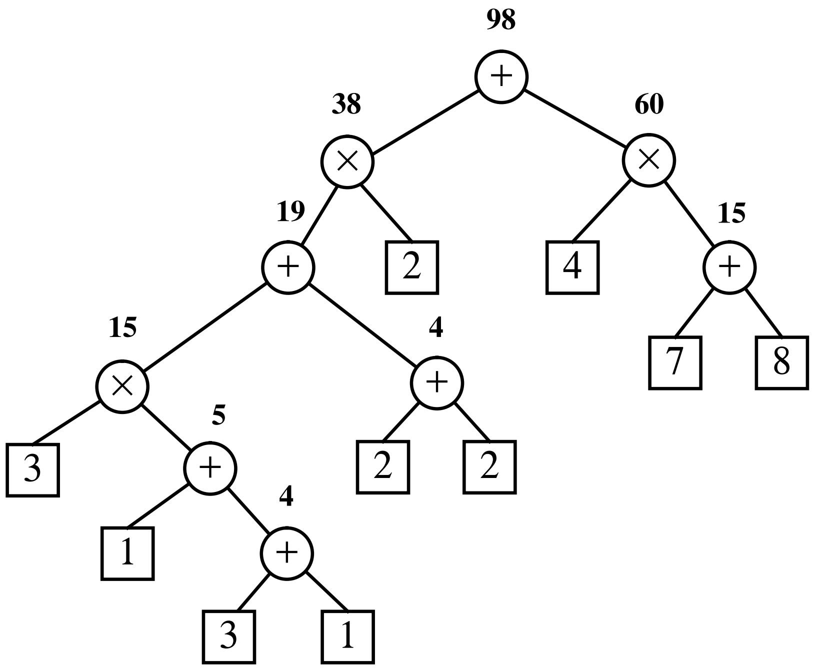

Figure 1: The tree for the arithmetic expression $((((3 \times (1 + (3 + 1))) + (2 + 2)) \times 2) + (4 \times (7 + 8)))$ with ordinary $+$ and $\times$ operators. Next to each node is shown the value of the corresponding subexpression.

rooted at μ and replace it with a leaf with value $\mathit{Evaluate}(\mu)$.

The dependence of the value y of an expression $\mathcal{E}$ from the value of variable x can be be expressed in the canonical form $y = (a \otimes x) \oplus b$, where a and b are elements of $\mathcal{S}$ [6]. In terms of the tree $\mathcal{T}$ representing $\mathcal{E}$, this property extends immediately to the dependence of the value y of a node ν from the value x of one of its descendants μ. Hence, we can assign the pair (a, b) to the path of $\mathcal{T}$ from μ to ν. We show later (in the description of the *join* operation) how to find and maintain these values for a path between such nodes μ and ν.

We keep an expression-tree $\mathcal{T}$ in a data structure based on dynamic trees [23]. Edges of $\mathcal{T}$ are considered to be directed from the child to the parent. An edge (μ, ν) of $\mathcal{T}$ is said to be *solid* if the subtree rooted at node μ has more than half of the nodes of the subtree rooted at its parent ν. Otherwise, edge (μ, ν) is said to be *dashed*. Therefore, there is at most one solid edge entering any node (from its children) and every node is in exactly one maximal path of solid edges (of length 0 or more). We refer to these paths as *solid paths* (see Fig. 3). A solid path Π is represented as a balanced binary tree $\mathcal{B}_\Pi$ (see Fig. 4), so that $\mathcal{T}$ is then stored as a collection of these *path trees*. Note that any leaf-to root path in $\mathcal{T}$ contains $O(\log n)$ dashed edges, where n is the size of $\mathcal{T}$.

We maintain the invariant, called the *path invariant*, that the tail τ and head σ of each solid path store the actual values $\mathit{Evaluate}(\tau)$ and $\mathit{Evaluate}(\sigma)$ of their subexpressions. The value of $\mathit{Evaluate}(\sigma)$ is known if σ is a leaf of $\mathcal{T}$, or is calculated from the values of $\mathit{Evaluate}(\sigma_1)$ and $\mathit{Evaluate}(\sigma_2)$ for the children σ_1 and σ_2 of σ. These values will be known by the path invariant, since σ_1 and σ_2 are the tails of their respective solid paths. We find

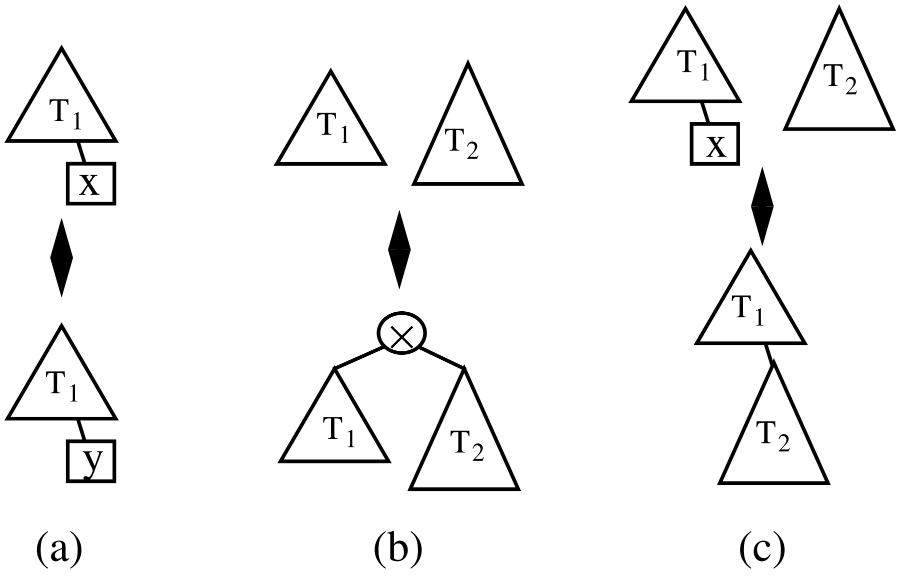

Figure 2: (a) Operation *Change*. (b) Operations *Construct* and *Destruct*. (c) Operations *Graft* and *Prune*.

the value of $Evaluate(\tau)$ using the path trees.

Let Π be a solid path from σ to τ. The leaves of tree $\mathcal{B}_\Pi$ are the nodes of Π in left-to-right order from σ to τ. Each internal node $\eta \in \mathcal{B}_\Pi$ represents the subpath of Π between its external descendants $head(\eta)$ and $tail(\eta)$. We store at η the pair (a, b) such that (see Fig. 4):

$$Evaluate(tail(\eta)) = (a \otimes Evaluate(head(\eta))) \oplus b.$$

Now, we show how to perform operation $Evaluate(\mu)$ (see Fig. 5). Let Π be the solid path from σ to τ containing μ. If $\mu = \sigma$ or $\mu = \tau$, by the path invariant μ stores its actual value. Otherwise, the path from σ to μ can be decomposed into $O(\log d)$ subpaths each associated with a node of $\mathcal{B}_\Pi$, where d is the distance between σ and μ in $\mathcal{T}$. Since σ is the head of the first subpath, its value is known. The value of the tail of each subpath is computed from the value of the head using its (a, b) pair. The value of the head of the next subpath is directly calculated from the value of its children, which are the tail of the previous subpath and the tail of another solid path. Hence, since the values stored at σ and at the siblings of the nodes of Π are actual, we can compute the value of μ in time $O(\log d)$.

Operation $Change(\lambda, x)$ affects only the values of the nodes in the path from λ to the root ρ. If such path contains dashed edges, then we temporarily convert it into a solid path by changing dashed edges to solid and solid edges to dashed such that any node continues to have at most one incoming solid edge. This is done with the following operations, derived from dynamic trees [23]:

- *splice*(*path* Π) — This operation assumes that Π is a solid path ending at $\mu \neq \rho$. Convert the dashed edge leaving μ to solid and convert the solid edge (if it exists) entering the parent ν of μ to dashed. Let Π' be the solid path containing ν. To

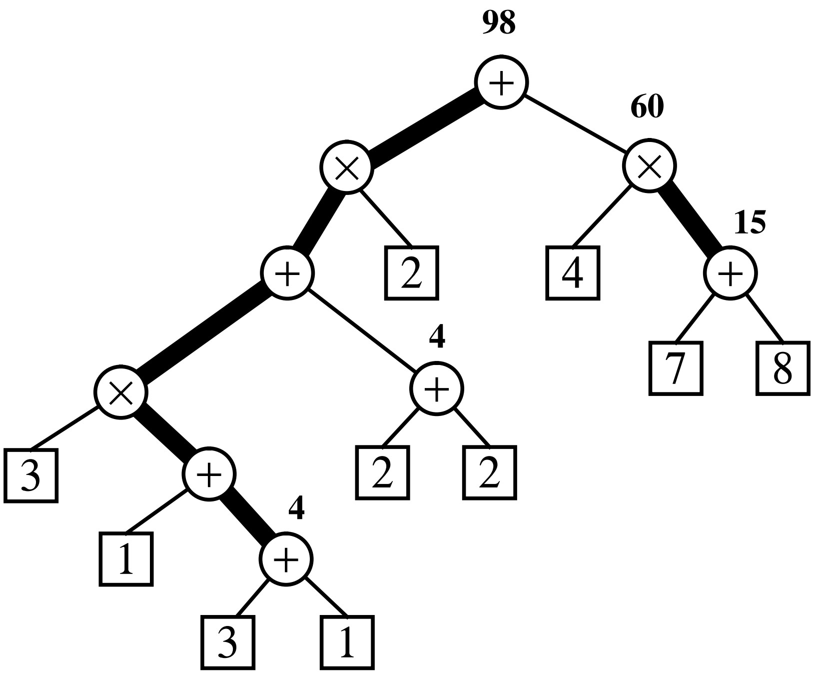

Figure 3: Solid paths for the expression tree of Fig. 1. The values of the first and last node of each solid path are shown.

implement *splice*(Π) we need to convert edge $(\mathrm{sib}(\mu), \nu)$ from solid to dashed ($\mathrm{sib}(\mu)$ denotes the sibling of node μ). Let Π'' be the resulting solid path starting at ν. We obtain Π'' by splitting $\mathcal{B}_{\Pi'}$. To maintain the path invariant, we call *Evaluate*($\mathrm{sib}(\mu)$) and *Evaluate*(ν). We complete *splice* by concatenating $\mathcal{B}_{\Pi}$ and $\mathcal{B}_{\Pi''}$.

- *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, which undoes its effect. An example of *expose* operation on the tree of Fig. 1 is shown in Fig. 4.a.
- *conceal*(μ) — Restore the original type (solid or dashed) of the edges entering the nodes on the path from μ to ρ. This operation is the inverse of *expose*, and also consists of a sequence of *splice* operations.

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''. The (a, b) pair of ζ is computed from the corresponding pairs (a', b') of ζ' and (a'', b'') of ζ'', the operator $\odot$ on the head node of Π'', and the value c of the sibling of the tail node of Π' in the following manner:

$$Evaluate(\tau') \;=\; a' \otimes Evaluate(\sigma') \oplus b'$$

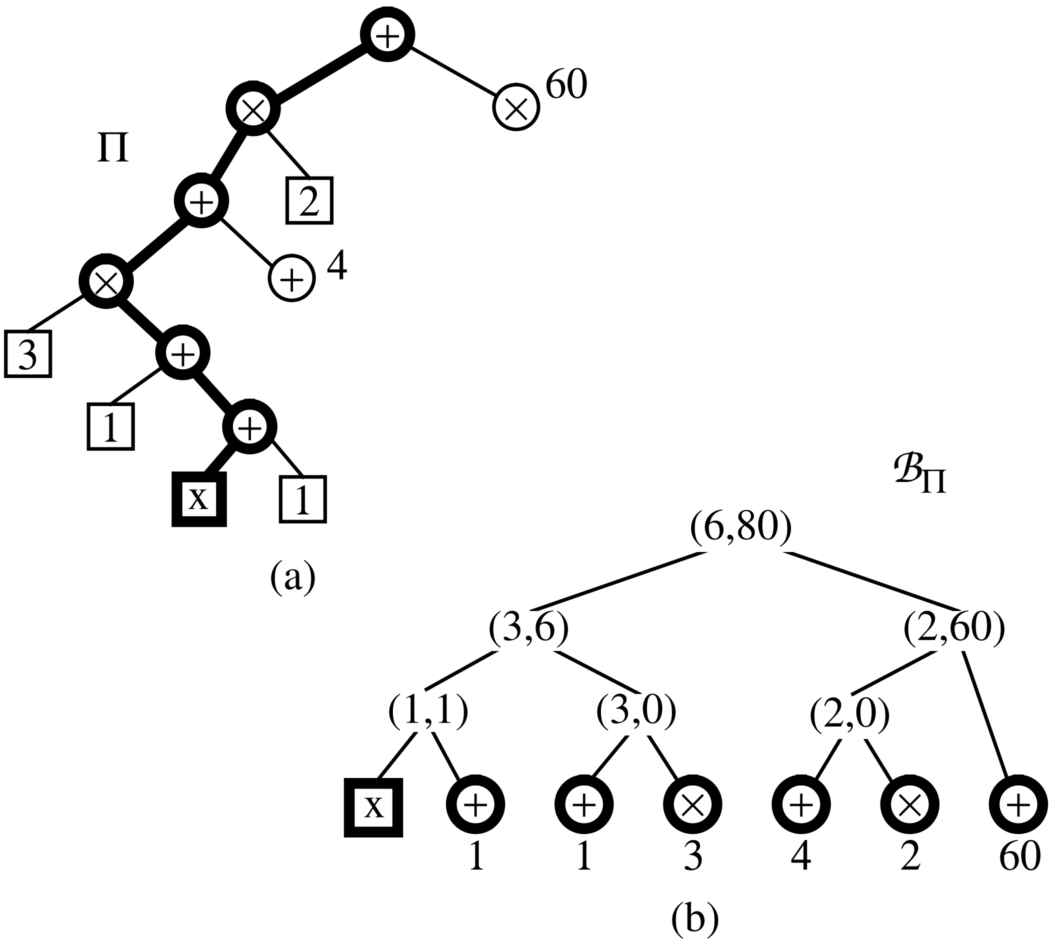

Figure 4: (a) A solid path Π created by an *expose* operation in the tree $\mathcal{T}$ of Fig. 1. (b) The path tree $\mathcal{B}_\Pi$ for the solid path Π. The leaves of $\mathcal{B}_\Pi$ represent the nodes of Π. Shown at each leaf is the operator at the corresponding node of Π, and the value of the child of such node that is not on Π. Shown at each internal node η of $\mathcal{B}_\Pi$ is the (a, b) pair for the subpath of Π represented by η. For example, the value of the tail of Π in terms of the value x of the head of Π is $6 \cdot x + 80$.

$$\begin{aligned}
\mathit{Evaluate}(\sigma'') &= \mathit{Evaluate}(\tau') \odot c \\
\mathit{Evaluate}(\tau'') &= a'' \otimes \mathit{Evaluate}(\sigma'') \oplus b'' \\
\mathit{Evaluate}(\tau'') &= a'' \otimes (\mathit{Evaluate}(\tau') \odot c) \oplus b'' \\
\mathit{Evaluate}(\tau'') &= a'' \otimes ((a' \otimes \mathit{Evaluate}(\sigma') \oplus b') \odot c) \oplus b''
\end{aligned}$$

Therefore, we have:

$$a = \begin{cases} a' \otimes a'' & \text{if } \odot = \oplus \\ a' \otimes a'' \otimes c & \text{if } \odot = \otimes \end{cases}$$

$$b = \begin{cases} (a'' \otimes (b' \oplus c)) \oplus b'' & \text{if } \odot = \oplus \\ (a'' \otimes b' \otimes c) \oplus b'' & \text{if } \odot = \otimes \end{cases}$$

An example of operation *join* is shown in Fig. 6.

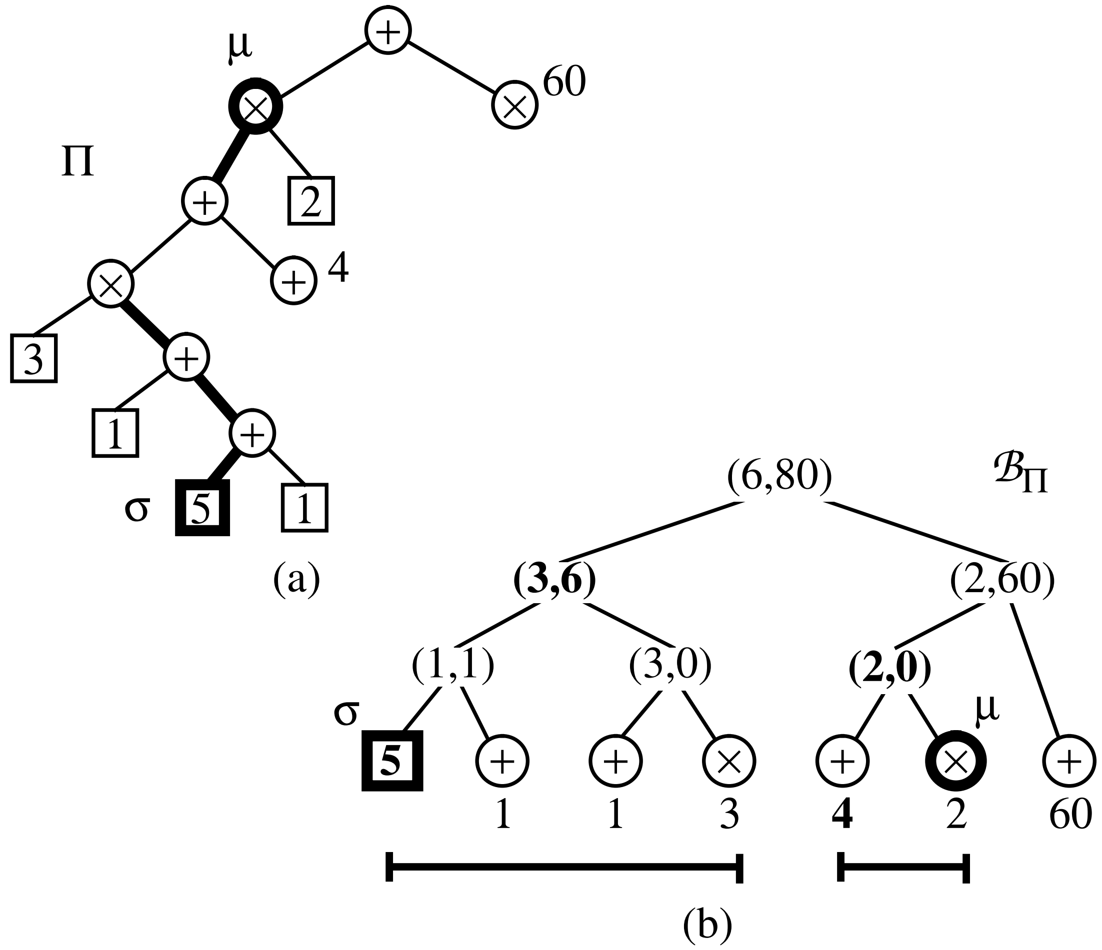

Figure 5: (a) A solid path Π with head node σ and an intermediate node μ. (b) Computation of the value of μ using the path tree $\mathcal{B}_\Pi$. The path from σ to μ is decomposed into two subpaths so that the value of μ is given by $\mathit{Evaluate}(\mu) = 2 \cdot (4 + (3 \cdot \mathit{Evaluate}(\sigma) + 6)) = 50$.

- *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.

Operation *Change*(λ, x) is realized as follows. First, we issue *expose*(λ). Next, we change the value of λ to x and re-evaluate the expression with *Evaluate*(ρ), where ρ is the root of the tree containing λ. Finally, we perform *conceal*(λ) to restore the original solid and dashed edges. We use biased search trees [4] to represent the path-trees, where the weight of node μ of tree $\mathcal{T}$ is 1 if μ is a leaf of $\mathcal{T}$, or one plus the sum of the weights of the children of μ that are connected through dashed edges. With this biasing, we can implement *expose* and *conceal* with $O(\log n)$ elementary tree operations. Hence, operation *Change*(λ, x) takes time $O(\log n)$, where n is the size of the tree containing λ.

Operations *MakeVar* and *DeleteVar* can be trivially implemented in $O(1)$ time. Opera-

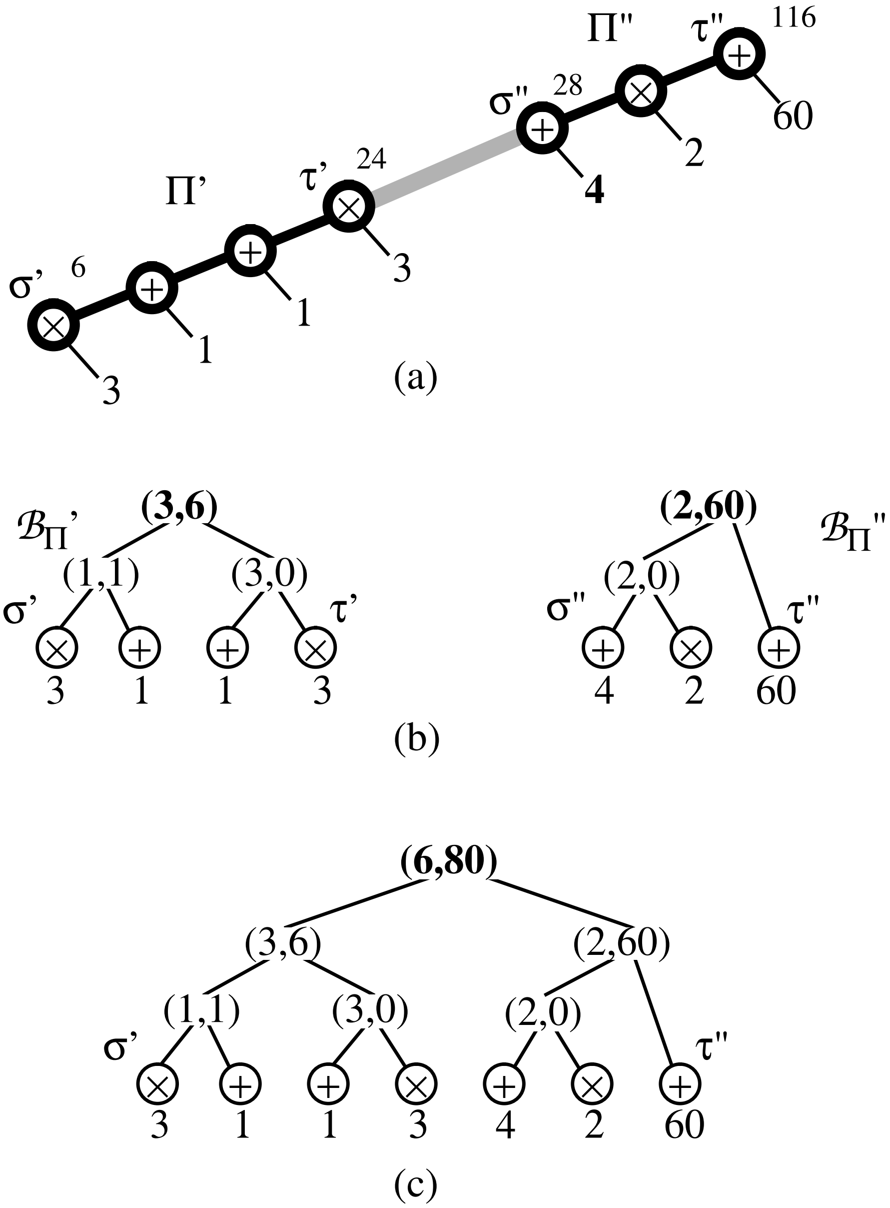

Figure 6: Example of operation *join*: (a) Solid paths Π' and Π'' connected by a dashed edge from the tail of Π' to the head of Π''. (b) Path trees $\mathcal{B}_{\Pi'}$ and $\mathcal{B}_{\Pi''}$. (c) Path tree for the solid path obtained by joining the roots of $\mathcal{B}_{\Pi'}$ and $\mathcal{B}_{\Pi''}$.

tions *Construct*, *Destruct*, *Graft* and *Prune* are performed using variations of the following dynamic tree operations:

- *link*(*node* ρ, μ) — This operation assumes that ρ is the root of a tree $\mathcal{T}$. Add an edge from ρ to μ, thus making $\mathcal{T}$ a subtree of the tree containing μ.
- *cut*(*node* μ) — This operation assumes that μ is not the root of a tree. Remove the edge from μ to its parent, thus separating the subtree rooted at μ.

By maintaining solid and dashed edges by using partitioning by weight [23], and representing the path trees as globally biased binary trees [4], we get:

Theorem 1 *Let S be a semiring with binary operators $\oplus$ and $\otimes$. There is a fully dynamic data structure for maintaining a collection of expressions over S with the following performance:*

1. *an expression of size n uses $O(n)$ space;*
2. *operations* MakeVar *and* DeleteVar *take each $O(1)$ worst-case time;*
3. *operations* Evaluate, Change, Construct, Destruct, Graft *and* Prune *take each $O(\log n)$ worst-case time, where n is the total size of the expressions involved in the operation.*

The above theorem can be extended to the case where either or both operations $\oplus$ and $\otimes$ admit an inverse. For example, if $\oslash$ is the inverse of $\otimes$, the dependency of the value y of a node μ from the value x of a descendant of μ can be represented by the following canonical form [6]:

$$y = ((a \otimes x) \oplus b) \oslash ((c \otimes x) \oplus d).$$

Hence, we modify the data structure so that the internal nodes of the path trees store a quadruple (a, b, c, d). We omit the details on the corresponding modifications to the elementary tree operations, and conclude:

Corollary 1 *The result of Theorem 1 holds also for expressions that include the inverse of either or both $\oplus$ and $\otimes$.*

3 Series-Parallel Digraphs

An important application of the technique presented in Section 2 is the dynamic maintenance of maximum flow and shortest st-path in series-parallel digraphs.

A *source* of a digraph G is a vertex without incoming edges. A *sink* of G is a vertex without outgoing edges. A *pole* of G is either a source or a sink of G. A *series-parallel digraph* is a digraph G with exactly one source s and one sink t, recursively defined as follows (see Fig. 7.a):

- A digraph G consisting of a single edge from s to t is a series-parallel digraph.
- Given series-parallel digraphs G' and G'' the digraph G obtained by identifying the source of G' with the source of G'' and the sink of G' with the sink of G'' is a series-parallel digraph. This is called a *parallel composition*.
- Given series-parallel digraphs G' and G'', the digraph G obtained by identifying the sink t' of G' with the source s'' of G'' is a series-parallel digraph. This is called a *series composition*, and the vertex $v = s'' = t'$ is called the *join-vertex* of such composition.

A series-parallel digraph is connected, planar and acyclic. It may have multiple edges and can be embedded with the source and sink on the external face. In the following we denote with m the number of edges of the series-parallel digraph G currently being considered (the number of vertices is at most $m+1$), and with s and t the source and sink of G. It is possible to test in $O(m)$ time whether a digraph is a series-parallel digraph [29]. It has been recently shown that the transitive closure of a series-parallel digraph can be dynamically maintained in $O(\log m)$ time per query or update operation using $O(m)$ space [17,27].

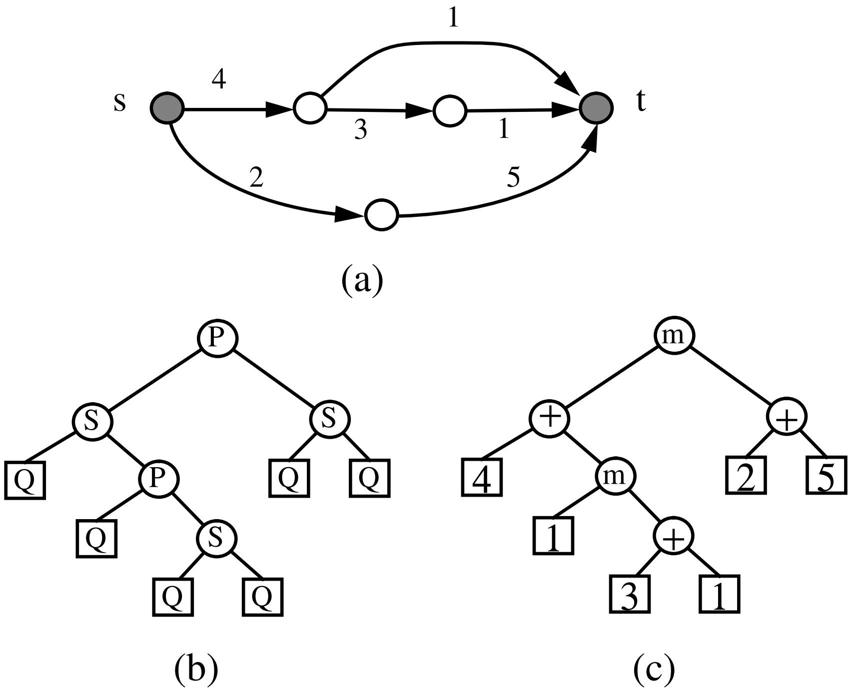

Figure 7: (a) A series-parallel digraph G with weighted edges. (b) The SPQ-tree $\mathcal{T}$ of G. (c) Expression tree for computing the length of a shortest st-path in G.

A series-parallel digraph G is naturally associated with a rooted binary tree $\mathcal{T}$, which we call *SPQ-tree* of G (other authors call it *decomposition tree* or *parse tree*). A node $\mu \in \mathcal{T}$ represents a subdigraph 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 node of $\mathcal{T}$ are of three types: S-nodes, P-nodes, and Q-nodes. Tree $\mathcal{T}$ is defined recursively as follows (see Fig. 7.b):

- If G is a single edge, then $\mathcal{T}$ consists a single *Q-node* μ.
- If G is created by the parallel composition of series-parallel digraphs G' and G'', let $\mathcal{T}'$ and $\mathcal{T}''$ be the SPQ-trees of G' and G'', respectively. The root of $\mathcal{T}$ is a *P-node* and has subtrees $\mathcal{T}'$ and $\mathcal{T}''$. (The order of the subtrees is arbitrary.)
- If G is created by the series composition of series-parallel digraphs G' and G'', where the sink of G' is identified with the source of G'', let $\mathcal{T}'$ and $\mathcal{T}''$ be the SPQ-trees of G' and G'', respectively. The root of $\mathcal{T}$ is an *S-node* and has left subtree $\mathcal{T}'$ and right subtree $\mathcal{T}''$.

The leaves of $\mathcal{T}$ are Q-nodes. The internal nodes are either P-nodes or S-nodes. If G has m edges, then $\mathcal{T}$ has m leaves and hence $2m-1$ nodes. Tree $\mathcal{T}$ can be constructed in $O(m)$ time using the recognition algorithm of [29].

Consider a digraph G with weighted edges, and let s and t be two vertices of G. Let e be an edge of G. For the shortest st-path problem, the weight of e represents the length of e, denoted *length*(e). The shortest st-path problem consists of finding a directed path in

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

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

Figure 8: Equations to calculate the value of $maxflow(\mu)$ and $shortest(\mu)$ for G_μ, the pertinent digraph for node μ.

G from s to t of minimum total length. For the maximum flow problem, the weight of e represents the capacity of e, denoted $capacity(e)$. A flow for G is a function $f(e)$ over the edges of G that satisfies the following conditions:

- for every edge e, $0 \leq f(e) \leq capacity(e)$;
- for every vertex $v \neq s, t$, $\sum_{e \in In(v)} f(e) = \sum_{e \in Out(v)} f(e)$, where $In(v)$ and $Out(v)$ denote the sets of incoming and outgoing edges of v, respectively.

The *value* of f is given by $\sum_{e \in Out(s)} f(e) = \sum_{e \in In(t)} f(e)$. The maximum flow problem consists of finding a flow for G with maximum value. It is well-known that the maximum flow value is equal to the total capacity of a *minimum cut* separating s from t [28].

For a series-parallel digraph G the problems of determining the value of a maximum flow and the length of a shortest st-path are closely related. Let $\mathcal{T}$ be the SPQ-tree of G. We associate with each node μ of $\mathcal{T}$ the values $maxflow(\mu)$ and $shortest(\mu)$, where $maxflow(\mu)$ is the maximum flow value of the pertinent digraph G_μ of μ, and $shortest(\mu)$ is the length of a shortest st-path in G_μ. (For a Q-node λ, $maxflow(\lambda) = capacity(e)$ and $shortest(\mu) = length(e)$, where e is the edge associated with λ.) Let $children(\mu)$ denote the set of children of node $\mu \in \mathcal{T}$. The resulting equations are shown in Fig. 8.

Therefore, the SPQ-tree of a series-parallel digraph can be viewed as an expression tree with operators min and + over the reals (note that + distributes over min). For the maximum flow (resp. shortest st-path) problem, P-nodes store operator + (resp. min) and S-nodes store operator min (resp. +). The Q-nodes store the variables of the expression, i.e., the edge weights (see Fig. 7.c).

First, we consider the problem of performing the following operations on a series-parallel digraph G:

- *MaxFlow*(*node* μ) — Return the maximum flow value for the pertinent digraph G_μ of μ.
- *ShortestPath*(*node* μ) — Return the length of a shortest path from the source to the sink of the pertinent digraph G_μ of μ.
- *Update*(*node* λ; *real* x) — This operation assumes that λ is a Q-node associated with an edge e. Set the weight of edge e equal to x.

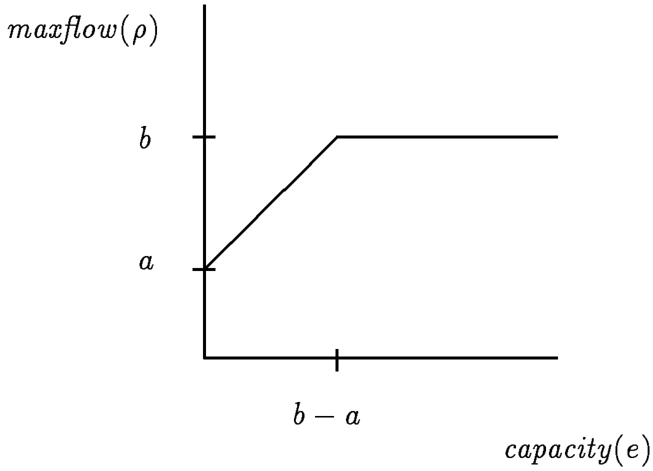

Figure 9: The maximum flow value for G (evaluated at the root ρ of $\mathcal{T}$) versus the capacity of an edge e, $maxflow(\rho) = \min(a + capacity(e)\,,\, b)$, when $a < b$.

By Theorem 1, we have:

Theorem 2 *Given a series-parallel digraph G with m weighted edges, there exists a dynamic data structure that uses $O(m)$ space and supports operations* MaxFlow, ShortestPath, *and* Update *in $O(\log m)$ worst-case time.*

Let λ be a leaf of the SPQ-tree of G rooted at ρ, and e be the edge associated with λ. The dependence of the value of the maximum flow in G from the capacity of e is characterized by a pair (a, b) and we have:

$$maxflow(\rho) = \min(a + maxflow(\lambda), b).$$

If $b \leq a$, changes in the capacity of edge e do not affect the maximum flow of G. Otherwise, $b - a$ is the largest contribution of edge e to the maximum flow, which is achieved by setting $capacity(e) = b - a$ (see Fig. 9). Hence, for any flow f in G we have $f(e) \leq b - a$. Also, edge e is on some minimum cut if and only if $capacity(e) \leq b - a$. Similar considerations hold for the shortest path problem.

Corollary 2 *The data structure of Theorem 2 allows to determine whether an edge e is on some minimum cut or shortest st-path of G in $O(\log m)$ worst-case time. Also, it allows to determine in $O(\log m)$ time the maximum value of $f(e)$ over all admissible flows f in G.*

Now, we consider a fully dynamic environment for the maintenance of maximum flow and shortest st-path on a collection $\mathcal{G}$ of series-parallel digraphs. Namely, in addition to *MaxFlow*, *ShortestPath*, and *Update*, we introduce the following operations:

- *ReportPath*(*node* μ) — Report a shortest st-path in the pertinent digraph G_μ of node μ.
- *ReportCut*(*node* μ) — Report a minimum cut in the pertinent digraph G_μ of node μ.
- *MakeDigraph* — Create a new elementary series-parallel digraph G, represented by a single Q-node with zero weight, 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 of zero weight. The series-parallel digraph G_μ is added to $\mathcal{G}$.
- *InsertEdge*(*vertex* v', v''; *edge* e) — Insert a new zero-weight 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 zero-weight 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 zero-weight edge e. The operation is performed only if e' and e'' are the only incident edges of v.

A series-parallel digraph G can be represented by an SPQ-tree $\mathcal{T}$ such that if node ν is the parent of μ and ν and μ are of the same type, then μ is the left child of ν. In the following, we will assume that this property, called *chain invariant*, holds, and we refer to the maximal subpaths in $\mathcal{T}$ of nodes of the same type as *chains*. We call Γ an *S-chain* if it consists of S-nodes and a *P-chain* if it consists of P-nodes. Note that the chain invariant is not restrictive. Any series-parallel digraph can be represented by a SPQ-tree that meets the chain invariant.

Let $\Gamma = (\mu_1, \mu_2, \cdots, \mu_{r-1})$ be an S-chain ordered such that μ_i is the left child of μ_{i+1} $(1 \leq i \leq r-2)$. Note that this implies that the chain subpaths run from the leaves toward the root of the tree. The *skeleton* of Γ, denoted *skeleton*(Γ), is the digraph consisting of r edges $e_i = (v_{i-1}, v_i)$, $1 \leq i \leq r$, where v_0 and v_1 are the source and sink of the pertinent digraph of the left child of μ_1, and v_i, $2 \leq i \leq r$, is the sink of the pertinent digraph of μ_{i-1}. For $i = 1, \cdots, r-1$, node μ_i is called the *proper node* of vertex v_i. Note that v_i is the join-vertex used in the series composition at node μ_i. Vertices are implicitly represented by the SPQ-tree. Each vertex distinct from s and t is mapped to its unique proper node. Each node is mapped to the source and sink of its pertinent digraph. Additionally, each S-node is the proper node of a unique vertex. The concepts of proper node and skeleton are analogous to the corresponding ones developed for planar st-graphs in [10].

We equip the tree $\mathcal{T}$ with a secondary data structure for maintaining the S-chains and their skeletons. Namely, we store with each S-chain $\Gamma = (\mu_1, \mu_2, \cdots, \mu_{r-1})$ the poles v_0 and v_r of its skeleton and a balanced binary tree, called *routing tree*. The i-th $(1 < i < r)$ leaf of the routing tree of Γ represents edge e_i of *skeleton*(Γ) and the i-th internal node $(1 < i < r-1)$ represents the join vertex of node μ_i. The routing trees allows us to find in $O(\log m)$ time the proper S-node of a vertex v and the first and last nodes of the S-chain containing a node μ. A similar technique of augmenting dynamic trees with balanced trees is described in [11].

Operations *MakeDigraph* and *DeleteDigraph* can be trivially implemented in $O(1)$ time. Operations *Attach* and *Detach* correspond to *Graft* and *Prune*, respectively, and hence take $O(\log m)$ time. We perform operation $Compose(X, \rho', \rho'')$, with $X = S$ or $X = P$, as follows. If ρ'' is not an X-node, we do a *Construct* operation with the appropriate operator. Otherwise, we need to maintain the chain invariant. Using the routing tree of the chain Γ'' containing ρ'' we access the first node of Γ'' and prune with a *Detach* operation its left subtree, which is replaced by a leaf λ. Then, we perform a *Construct* operation on ρ' and the pruned subtree. Finally, we graft the resulting tree to λ with an *Attach* operation (for example, see figure 10. Hence, *Compose* takes $O(\log m)$ time.

Consider now operation $InsertVertex(v, e, e', e'')$, and let λ be the Q-node of e. If λ is a left child or its parent is a P-node, we simply replace λ with an S-node having Q-node children for e' and e''. This can be done in $O(\log m)$ time with two *MakeDigraph* operations, followed by a *Compose* operation and an *Attach* operation. Otherwise (λ is a right child of an S-node μ) we perform the following sequence of operation: (a) $Detach(\nu)$, where $\nu = \mathrm{sib}(\lambda)$; (b) create new Q-nodes λ' and λ'' for e' and e'' using *MakeDigraph*; (c) $Attach(Compose(S, \nu, \lambda'), \mathrm{sib}(\lambda))$; and (d) $Attach(\lambda'', \lambda)$. All this takes $O(\log m)$ time.

The inverse operation $DeleteVertex(e, v, e', e'')$ is implemented similarly with a constant number of *MakeDigraph*, *DeleteDigraph*, *Attach* and *Detach* operations.

The restructuring of the SPQ-tree caused by *InsertEdge* and *DeleteEdge* is more complex. The following lemmas determine when it is possible to perform such operations.

Lemma 1 *Let G be a series-parallel digraph with SPQ-tree $\mathcal{T}$, and v' and v'' be two vertices of G. Then the digraph $\bar{G}$ obtained by inserting an edge from v' to v'' is a series-parallel digraph if and only if one of the following conditions is true:*

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

Proof:

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

(Only If) We use the property that a series-parallel digraph has no subgraph homeomorphic to the digraph W shown in Fig. 11 [29]. Suppose that v' and v'' do not meet either condition of the lemma. Assume that $\mathcal{T}$ keeps the chain invariant. Let μ be the least common ancestor in $\mathcal{T}$ of the proper nodes of v' and v'', and Γ be the chain containing μ. Let e be the edge from v' to v''. If μ is a P-node then we consider the subgraph of $\bar{G}$ consisting of edge e and two directed paths from the poles of G_μ through the pertinent digraphs of the children of μ. Such a digraph is homeomorphic to W, which implies that $\bar{G}$ is not a series-parallel digraph. Now, suppose that μ is an S-node; G has a directed path either from v' to v'' or from v'' to v'. If G has a path from v'' to v', then $\bar{G}$ has a cycle and hence is not a series-parallel digraph. Thus, G has a path from v' to v''. Since Condition 2 is not met, at least one of v' and v'', say v'', is not a vertex of $skeleton(\Gamma)$, so that the pertinent digraph of a child of μ contains v'' and has a directed path p between its poles that avoids v''. Thus, we can identify in $\bar{G}$ a subgraph homeomorphic to W that consists of edge e, path p, and a path from v' to the sink of G_μ through v'', so that $\bar{G}$ is not a series-parallel digraph. □

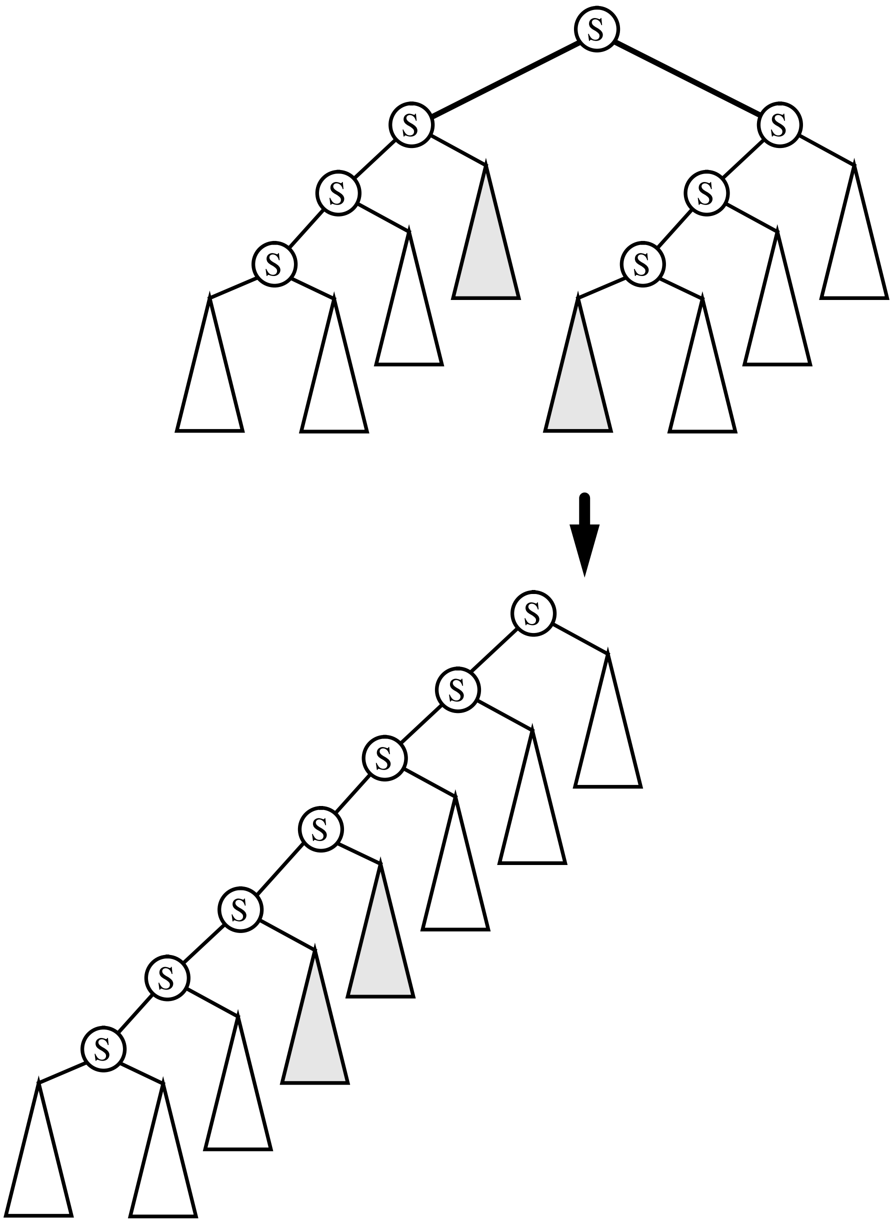

Figure 10: An example of restoring the chain invariant after a composition.

Lemma 2 *Let G be a series-parallel digraph with SPQ-tree $\mathcal{T}$, and v' and v'' be two vertices of G. Let Γ' and Γ'' be the S-chains containing the proper nodes μ' of v' and μ'' v'', respectively, and Γ be the S-chain containing the least-common ancestor of μ' and μ''. Then the digraph obtained by inserting an edge from v' to v'' is a series-parallel digraph if and only if one of the following conditions is true:*

1. *$v' = s$ and $v'' = t$,*
2. *$\Gamma = \Gamma' = \Gamma''$ and v' precedes v'' in skeleton(Γ),*
3. *$\Gamma = \Gamma'$ and v' is the source of skeleton(Γ''), or*

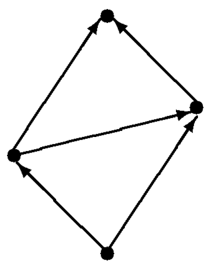

Figure 11: Digraph W that is a forbidden homeomorphic subgraph for a series-parallel digraph.

4. *$\Gamma = \Gamma''$ and v'' is the sink of skeleton(Γ').*

Proof: Follows directly from Lemma 1. □

The following lemma determines when we can delete an edge.

Lemma 3 *Let G be a series-parallel digraph G with SPQ-tree $\mathcal{T}$, and e be an edge of G represented by Q-node λ. The digraph obtained by removing edge e is a series-parallel digraph if and only if one of the following conditions is true:*

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

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

Testing for the applicability of *InsertEdge*(v', v'') is performed as follows: (a) find the proper nodes of v' and v'' using the routing trees (this takes $O(\log m)$ time); (b) test for the conditions in Lemma 2. Condition 2 can be checked in $O(\log m)$ time using the routing tree of Γ. Conditions 1, 3, and 4, can be checked in $O(1)$ time. Hence, the total time is $O(\log m)$.

Operation *InsertEdge*(v', v'', e) is done as follows (see Figs. 12–13). Find the proper nodes of v' and v'' and their corresponding chains Γ' and Γ''. Exit if none of the conditions in Lemma 2 is satisfied. Without loss of generality, assume that Γ'' is a descendant of Γ' or $\Gamma' = \Gamma''$. Using the the routing tree of Γ'', determine the nodes of Γ'' associated with the subchain from v' to v''. Finally, perform the appropriate transformation from Fig. 12 or Fig. 13. This can be done with a constant number of *MakeDigraph*, *Compose*, *Attach*, and *Detach* operations. Therefore, *InsertEdge* takes $O(\log m)$ time.

After testing the conditions of Lemma 3, operation *DeleteEdge*(e) is done with a constant number of *DeleteDigraph*, *Attach*, and *Detach* operations. Therefore, also *DeleteEdge*(e) takes $O(\log m)$ time.

Now, we show how to perform operation *ReportPath*(μ). A similar technique can be used

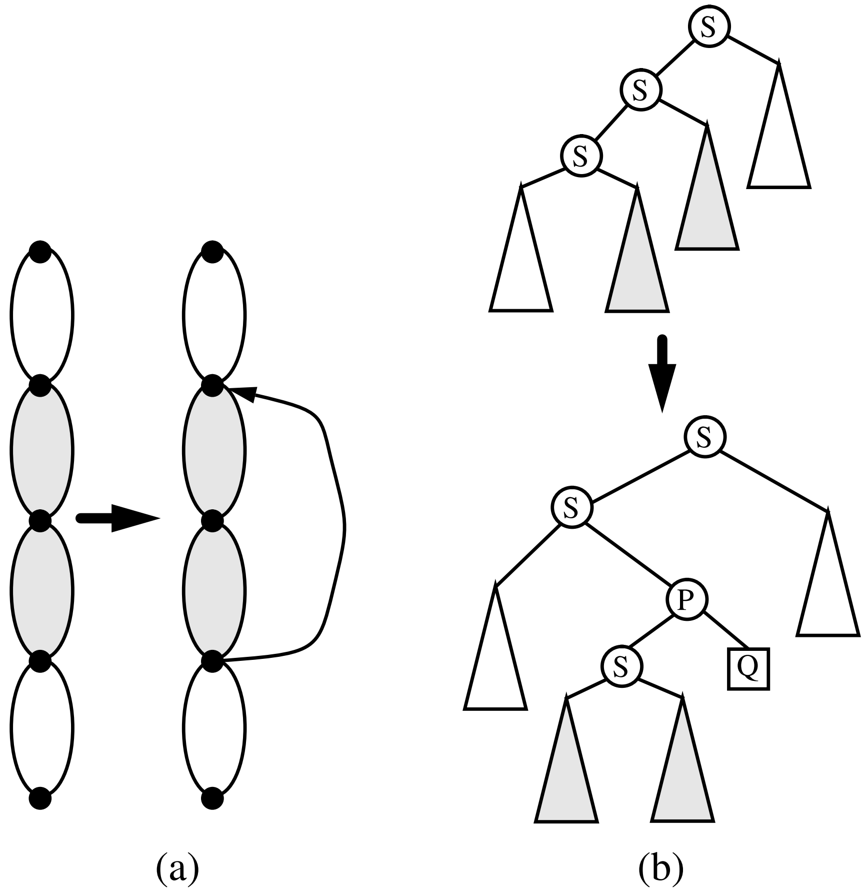

Figure 12: Modification of $\mathcal{T}$ in operation *InsertEdge*: splitting an S-chain. (a) Insertion of an edge between nonconsecutive vertices of the skeleton of an S-chain. (b) Transformation of the SPQ-tree.

for *ReportCut*(μ). If μ is a leaf, we report the corresponding edge and exit. If μ is an S-node, a shortest path in G_μ must go through the pertinent digraphs of both children of μ. Hence, we call recursively *ReportPath* on the left and right child of μ. If μ is a P-node, we consider the P-chain Γ of μ and we denote with Δ_μ the set of all nodes ν such that ν is a child of a node of Γ but ν is not on Γ. A shortest path in μ goes through the pertinent digraph $G_{\nu'}$ of a node $\nu' \in \Delta_\mu$ such that *shortest*(ν') is minimum. Hence, we call recursively *ReportPath* on such node ν'.

To determine quickly node ν' we modify the path trees such that each internal node η of a path tree $\mathcal{B}_\Pi$ stores a pointer to a node κ of the SPQ-tree that has the minimum value of *shortest* over all nodes that are children of nodes in the subpath of Π from *head*(η) to *tail*(η) but are not on such subpath. Since these nodes are tails of solid paths, their *shortest* values are actual. Also, we set-up routing trees associated with the P-chains. With these additional structures we find node ν' in time $O(\log m)$ as follows: (a) determine the bottommost node μ' of the P-chain Γ of μ using the routing tree of Γ; (b) create a solid path Π form μ' to μ by calling *expose*(μ') followed by *expose*(μ''), where μ'' is the parent of μ; (c) find ν' by following the pointer stored at the root of $\mathcal{B}_\Pi$; (d) restore the original solid paths and dashed

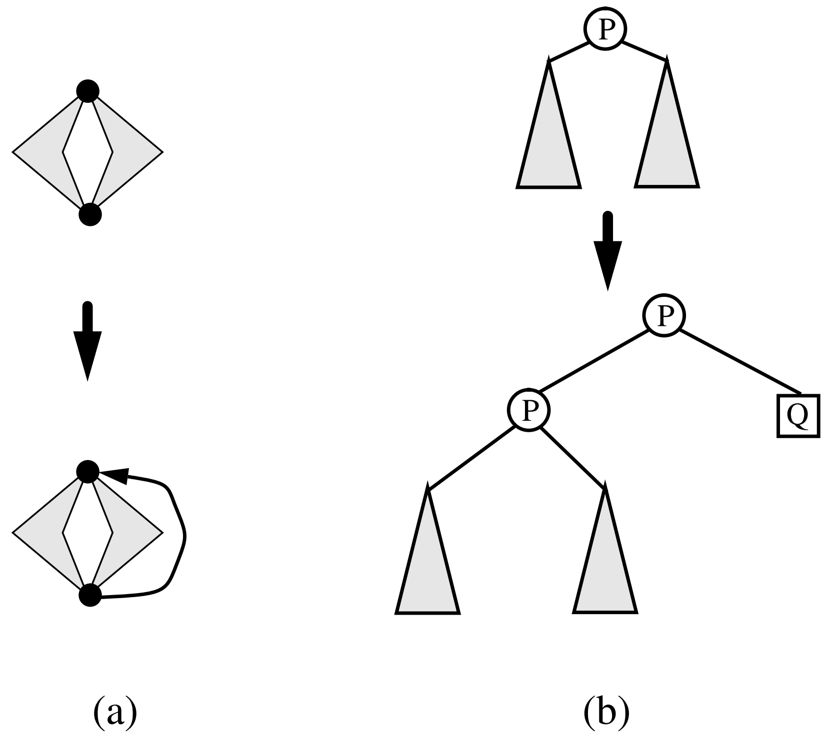

Figure 13: Modification of $\mathcal{T}$ in operation *InsertEdge*: extending a P-chain. (a) Insertion of an edge between consecutive vertices of the skeleton of an S-chain (the vertices are the source and sink of a parallel composition). (b) Transformation of the SPQ-tree.

edges with two *conceal* operations.

We summarize with the following theorem:

Theorem 3 *There is a fully dynamic data structure for maintaining a collection of edge-weighted series-parallel digraphs with the following performance:*

1. *a series-parallel digraph with m edge uses $O(m)$ space;*
2. *operations* MakeDigraph *and* DeleteDigraph *take each $O(1)$ worst-case time;*
3. *operations* MaxFlow, ShortestPath, Update, Compose, Attach, Detach, InsertEdge, DeleteEdge, InsertVertex, *and* DeleteVertex *take each $O(\log m)$ worst-case time, where m is the total number of edges of the series-parallel digraphs involved;*
4. *operations* ReportCut *and* ReportPath *take each $O(k \log m)$ worst-case time, where m is the number of edges of the series-parallel digraph involved, and k is the number of edges of the cut or path being reported.*

4 Further Applications

Complex geometric solids are represented in Constructive Solid Geometry (CSG) as a hierarchic combination of primitive objects (e.g., spheres and tetrahedra) using set operations such as union, intersection, and difference. The resulting compound object is described by a CSG-tree whose leaves are the primitive objects and whose internal nodes are set operators. CSG representations are a fundamental tool in computer graphics. Several efficient algorithms are given in [15] for CSG classification problems, such as determining whether a query point is inside a compound object.

We study a dynamic point inclusion problem where a collection of compound objects is modified by changing their primitive constituents and by update operations, such as link and cut, on their CSG-trees. Queries consist of reporting whether a given compound object contains a fixed point p. By applying dynamic expression trees to the technique of [15], we obtain:

Theorem 4 *The above dynamic point inclusion problem on a collection of compound objects defined by CSG-trees of total size n can be solved using an $O(n)$-space data structure that supports query and update operations in $O(\log n)$ worst-case time.*

Dynamic expression trees have immediate application to the problem of compacting *slicing floorplans*, a layout technique widely used in VLSI (see, e.g., [25]). A slicing floorplan is either a rectangle (called basic rectangle), or is the union of two slicing floorplans that share a horizontal side (called horizontal slice) or a vertical side (called vertical slice). Each basic rectangle r has a minimum width w_r and a minimum height h_r, which describe the dimensions of a circuit module to be placed inside r.

An important problem in VLSI layout is the minimization of the area of the slicing floorplan, subject to the above constraints on the height and width of the basic rectangles. This problem can be solved sequentially in $O(n)$ time, n being the number of basic rectangles, by representing the floorplan with two expression-trees, $\mathcal{T}_w$ and $\mathcal{T}_h$, where $\mathcal{T}_w$ is used to find the minimum width and $\mathcal{T}_h$ is used to find the minimum height [25]. Each leaf of $\mathcal{T}_w$ represents a basic rectangle r and stores w_r. Each internal node of $\mathcal{T}_w$ represents a slice and stores operator "+" for a vertical slice, and operator "max" for a horizontal slice. Hence, the minimum width of the slicing floorplan is the value at the root of $\mathcal{T}_w$. Analogous considerations can be done for tree $\mathcal{T}_h$.

By implementing trees $\mathcal{T}_w$ and $\mathcal{T}_h$ with dynamic expression trees, we can solve a dynamic version of the compaction problem for slicing floorplans, where query operations ask for the minimum area of a floorplan (or sub-floorplan) and update operations change the dimensions of a basic rectangle or join (or separate) two floorplans.

Theorem 5 *There exists a fully dynamic $O(n)$-space data structure for maintaining a collection of slicing floorplans of total size n, such that a query or update operation takes $O(\log n)$ worst-case time.*

5 Open Problems

We conclude the paper with the following open problems:

- Modify the data structure for series-parallel digraphs so that a shortest path or minimum cut can be reported in time $O(\log n + k)$.
- Develop an efficient technique for the dynamic maintenance of the shortest st-path or maximum flow in planar digraphs. Our technique allows to solve the problem for

planar st-graphs with $O(r \log n)$ query and update time, where n is the number of vertices and r is the size of the largest triconnected component of the graph. A planar st-graph is a planar acyclic digraph with one source and one sink, both on the external face. Planar st-graphs include series-parallel digraphs as a special case. No algorithm with poylogarithmic query and update times is known even for planar st-graphs with unit edge weights.

- Develop an efficient technique for the dynamic maintenance of the output value(s) of an arithmetic or boolean circuit. For general circuits it is unlikely that an algorithm with poylogarithmic query and update times exists, since the related problem of parallel evaluation of circuits is P-complete [18]. However, efficient dynamic solutions may exist for special classes of circuits that include trees.

Acknowledgement

We would like to thank Ajit Agrawal, Giuseppe Di Battista and Giuseppe Italiano for useful discussions.

References

[1] F.N. Afrati, D.Q. Goldin, and P.C. Kanellakis, "Efficient Parallelism for Structured Data: Directed Reachability in S-P Dags," Dept. Computer Science, Brown Univ., Technical Report CS-88-07, 1988.

[2] B. Alpern, R. Hoover, B. Rosen, P. Sweeney, and F.K. Zadeck, "Incremental Evaluation of Computational Circuits," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1990), 32–42.

[3] G. Ausiello, G.F. Italiano, A. Marchetti-Spaccamela, and U. Nanni, "Incremental Algorithms for Minimal Length Paths," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1990), 12–21.

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

[5] A.M. Berman, M.C. Paull, and B.G. Ryder, "Proving Relative Lower Bounds for Incremental Algorithms," *Acta Informatica* (to appear).

[6] R.P. Brent, "The Parallel Evaluation of Arithmetic Expressions in Logarithmic Time," in *Complexity of Sequential and Parallel Numerical Algorithms*, Academic Press, New York, 1973, 83–102.

[7] R.P. Brent, "The Parallel Evaluation of General Arithmetic Expressions," *J. ACM* 21 (1974), 201–206.

[8] M.D. Carrol and B.G. Ryder, "Incremental Data Flow Analysis via Dominator and Attribute Updates," *Proc. 15th ACM Symp. on Principles of Programming Languages* (1988), 274–284.

[9] N. Deo and C. Pang, "Shortest-Path Algorithms: Taxonomy and Annotations," *Networks* 14 (1984), 275–323.

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

[11] 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.

[12] S. Even and H. Gazit, "Updating Distances in Dynamic Graphs," *Methods of Operations Research* 49 (1985), 371–387.

[13] G. Gallo, M. Grigoriadis, and R.E. Tarjan, "A Fast Parametric Network Flow Algorithm," *SIAM J. on Computing* 18 (1989), 30–55.

[14] A.V. Goldberg, E. Tardos, and R.E. Tarjan, "Network Flow Algorithms," Dept. of Computer Science, Stanford Univ., Technical Report STAN-CS-89-1252, 1989.

[15] M.T. Goodrich, "Applying Parallel Processing Techniques to Classification Problems in Constructive Solid Geometry," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1990), 118–128.

[16] D. Gusfield and C. Martel, "A Fast Algorithm for the Generalized Parametric Minimum Cut Problem and Applications," Dept. of Computer Science and Engineering, Univ. of California, Davis, Technical Report CSE-89-21, 1989.

[17] G.F. Italiano, A. Marchetti-Spaccamela, and U. Nanni, "Dynamic Data Structures for Series-Parallel Graphs," *Proc. WADS' 89,* LNCS 382 (1989), 352–372.

[18] R.M. Karp and V. Ramachandran, "A Survey of Parallel Algorithms for Shared Memory Machines," in *Handbook of Theoretical Computer Science*, North Holland, 1990.

[19] E.L. Lawler, "Sequencing Jobs to Minimize Total Weighted Completion Time Subject to Precedence Constraints," *Annals of Discrete Mathematics* 2 (1978), 75–90.

[20] C.C. Lin and R.C. Chang, "On the Dynamic Shortest Path Problem," *Proc. Int. Workshop on Discrete Algorithms and Complexity* (1989), 203–212.

[21] W. Pugh and T. Teitelbaum, "Incremental Computation via Function Caching," *Proc. 16th ACM Symp. on Priciples of Programming Languages* (1989), 315–328.

[22] H. Rohnert, "A Dynamization of the All-Pairs Least Cost Problem," *Proc. STACS '85* (1985), 279–286.

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

[24] P.M. Spira and A. Pan, "On Finding and Updating Spanning Trees and Shortest Paths," *SIAM J. Computing* 4 (1975), 375–380.

[25] L. Stockmeyer, "Optimal Orientation of Cells in Slicing Floorplan Design," *Information and Control* 57 (1983), 91–101.

[26] K. Takamizawa, T. Nishizeki, and N. Saito, "Linear Time Computability of Combinatorial Problems on Series Parallel Graphs," *J. ACM* 29 (1982), 623–641.

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

[28] R.E. Tarjan, "Data Structures and Network Algorithms," *CBMS-NSF Regional Conference Series in Applied Mathematics* 44 (1983).

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

[30] L. Weinberg, "Linear Graphs: Theorems, Algorithms, and Applications," in *Aspects of Network and System Theory*, R.E. Kalman and N. DeClaris, eds., Holt, Rinehart and Winston, 1971.