Algorithmica (1996) 15: 302–318

Algorithmica
© 1996 Springer-Verlag New York Inc.

On-Line Maintenance of Triconnected Components with SPQR-Trees[1]

G. Di Battista[2] and R. Tamassia[3]

Abstract. We consider the problem of maintaining on-line the triconnected components of a graph G. Let n be the current number of vertices of G. We present an $O(n)$-space data structure that supports insertions of vertices and edges, and queries of the type "Are there three vertex-disjoint paths between vertices v_1 and v_2?" A sequence of k operations takes time $O(k \cdot \alpha(k, n))$ if G is biconnected ($\alpha(k, n)$ denotes the well-known Ackermann's function inverse), and time $O(n \log n + k)$ if G is not biconnected. Note that the bounds do not depend on the number of edges of G. We use the *SPQR-tree*, a versatile data structure that represents the decomposition of a biconnected graph with respect to its triconnected components, and the *BC-tree*, which represents the decomposition of a connected graph with respect to its biconnected components.

Key Words. Vertex connectivity, Dynamic algorithm, Dynamic data structure, Graph decomposition, Triconnected components, Network reliability.

1. Introduction. The development of dynamic algorithms for graph problems has acquired increasing theoretical interest in the last years and is motivated by many important applications in network optimization, VLSI layout, computational geometry, and distributed computing. The existing literature includes work on connected components, biconnected components, transitive closure, shortest paths, minimum spanning trees, and planarity testing (for a survey, see [2]).

An *on-line graph problem* consists of performing a sequence of query and update operations on a graph, such that each operation is completed before the next one is processed and future operations are not known in advance. We consider on-line graph problems where the updates are insertions of vertices and edges. Such problems are also referred to as semidynamic or incremental. Specifically, we consider the following update operations:

InsertEdge(v_1, v_2): Add an edge between vertices v_1 and v_2.
InsertVertex(v, v_1, v_2): Split edge (v_1, v_2) into two edges (v_1, v) and (v, v_2) by inserting vertex v.

[1] This research was supported in part by the National Science Foundation under Grant CCR-9007851, by the U.S. Army Research Office under Grants DAAL03-91-G-0035 and DAAH04-93-0134, by the Office of Naval Research and the Advanced Research Projects Agency under Contract N00014-91-J-4052, ARPA Order 8225, by the NATO Scientific Affairs Division under Collaborative Research Grant 911016, 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 17th International Colloquium on Automata, Languages, and Programming, Warwick, 1990.
[2] Dipartimento di Discipline Scientifiche, Sezione Informatica, Terza Università di Roma, via della Vasca Navale 84, 00146 Roma, Italy. dibattista@iasi.rm.cnr.it. Work performed in part while this author was with the Università di Roma "La Sapienza," Dipartimento di Informatica e Sistemistica.
[3] Department of Computer Science, Brown University, Providence, RI 02912-1910, USA. rt@cs.brown.edu.

Received August 17, 1992; revised January 10, 1994, and September 30, 1994. Communicated by T. Nishizeki.

AttachVertex(v, u): Add vertex v and connect it to vertex u by means of an edge.
MakeVertex(v): Add an isolated vertex v.

Any graph can be easily constructed with operations *MakeVertex* and *InsertEdge*. A connected graph can be constructed by using only *AttachVertex* and *InsertEdge* so that all the intermediate graphs are connected. Similarly, a biconnected graph can be constructed using only *InsertVertex* and *InsertEdge* so that all the intermediate graphs are biconnected. From now on, n and m denote the number of vertices and edges of the graph being considered.

In this paper we investigate the problem of maintaining on-line the triconnected components of a graph G. Namely, we want to support a repertory of operations consisting of the aforementioned updates and of the following query operation:

ThreePaths(v_1, v_2): Determine whether three vertex-disjoint paths between vertices v_1 and v_2 exist.

After a review of basic definitions in Section 2, in Section 3 we study the problem of maintaining the triconnected components of a biconnected graph. First, we present a versatile data structure, called the *SPQR-tree*, which essentially represents the decomposition of a biconnected graph with respect to its triconnected components. A first application of SPQR-trees to on-line planarity testing of biconnected graphs has been given in [1] and [2]. In this paper SPQR-trees are defined without reference to planarity. We show that, in a static environment, SPQR-trees support operation *ThreePaths* in $O(1)$ time using $O(n)$ space. Next, we consider update operations *InsertVertex* and *InsertEdge*, and show that the SPQR-tree can be maintained with a constant amortized number of elementary tree operations and set operations per update. Using a combination of fast union-find and split-find data structures stored at the nodes of the SPQR-tree, we obtain an $O(n)$-space data structure that supports operations *ThreePaths*, *InsertVertex*, and *InsertEdge* in $O(\alpha(k, n))$ amortized time, k being the total number of operations performed ($\alpha(k, n)$ denotes the well-known Ackermann's function inverse).

Previously, Kanevsky [7] has provided an implicit static representation of the separation pairs of a biconnected graph that uses $O(n)$ space, but did not consider queries nor updates. Westbrook and Tarjan [13] have presented a technique for the on-line maintenance of biconnected components that supports queries and updates in amortized time $O(\alpha(k, n))$. The on-line maintenance of triconnected components is explicitly mentioned as an open problem in [13]. Note that the static computation of the triconnected components of a graph can be done sequentially in $O(n + m)$ time [5], and on a CRCW PRAM in $O(\log n)$ time with $(n + m)\,\alpha(m, n)$ processors [3].

In Section 4 we present another data structure, the *BC-tree*, that represents the arrangement of the biconnected components of a connected graph. The BC-tree contains as a secondary structure the SPQR-trees of each biconnected component. We show that the BC-tree can be maintained with a constant amortized number of elementary tree operations and set operations, plus an $O(\log n)$-time overhead per vertex update. Graphs that are not connected are represented by a *BC-forest*. Using the BC-forest, we show how to maintain on-line the triconnected components of a general graph with an $O(n)$-space data structure that supports a sequence of k operations in time $O(n \log n + k)$.

2. Preliminaries. We recall some basic definitions on connectivity. A graph, or multigraph, G is k-connected if there are k vertex-disjoint paths between any two vertices of G. 1-connected, 2-connected, and 3-connected graphs are usually called connected, biconnected, and triconnected, respectively. It is well known that G is k-connected if there is no set of $k-1$ elements, each a vertex or an edge, whose removal disconnects G. Such a set is called a separating $(k-1)$-set. Separating 1-sets and 2-sets of vertices are called cutvertices and separation pairs, respectively.

The biconnected components of a connected graph (also called blocks) are:

(a) Its maximal biconnected subgraphs.
(b) Its separating edges together with their endpoints (trivial blocks).

The triconnected components of a biconnected graph G are defined as follows [5], [12]. If G is triconnected, then G itself is the unique triconnected component of G. Otherwise, let (u, v) be a separation pair of G. We partition the edges of G into two disjoint subsets E_1 and E_2 ($|E_1|, |E_2| \geq 2$), such that the subgraphs G_1 and G_2 induced by them have only vertices u and v in common. We continue the decomposition process recursively on $G_1' = G_1 + (u, v)$ and $G_2' = G_2 + (u, v)$ until no decomposition is possible. The resulting graphs are each either a triconnected simple graph, or a set of three multiple edges (*triple bond*), or a cycle of length three (*triangle*). The *triconnected components* of G are obtained from such graphs by merging the triple bonds into maximal sets of multiple edges (*bonds*), and the triangles into maximal simple cycles (*polygons*). The triconnected components of G are unique. See [5] and [12] for further details.

In the description of time bounds we use standard concepts of amortized complexity [10].

3. Biconnected Graphs. In this section we consider the problem of performing on a biconnected graph a sequence of *ThreePaths*, *InsertVertex*, and *InsertEdge* operations.

3.1. *SPQR-Tree.* Let G be a biconnected graph. A *split pair* of G is either a separation pair or a pair of adjacent vertices. A *split component* of a split pair $\{u, v\}$ is either an edge (u, v) or a maximal subgraph C of G such that $\{u, v\}$ is not a split pair of C. Let $\{s, t\}$ be a split pair of G. A *maximal split pair* $\{u, v\}$ of G with respect to $\{s, t\}$ is such that, for any other split pair $\{u', v'\}$, vertices u, v, s, and t are in the same split component.

In the example of Figure 1(a) the split pairs include $\{v_4, v_8\}$ and $\{v_1, v_2\}$, the subgraph induced by v_4, v_5, v_6, v_7, and v_8 is a split component, $\{v_5, v_7\}$ is a split pair that is not maximal with respect to $\{v_1, v_{14}\}$.

Let e be an edge of G between vertices s and t, called the *reference edge*. The SPQR-tree $\mathcal{T}$ of G with respect to e describes a recursive decomposition of G induced by its split pairs. Tree $\mathcal{T}$ is a rooted ordered tree whose nodes are of four types: S, P, Q, and R. Each node μ of $\mathcal{T}$ has an associated biconnected multigraph, called the *skeleton* of μ, and denoted by *skeleton*(μ). Tree $\mathcal{T}$ is recursively defined as follows (see Figure 1):

Trivial Case: If G consists of exactly two parallel edges between s and t, then $\mathcal{T}$ consists of a single Q-node whose skeleton is G itself.

Parallel Case: If the split pair $\{s, t\}$ has at least three split components $G_1, \ldots, G_k$

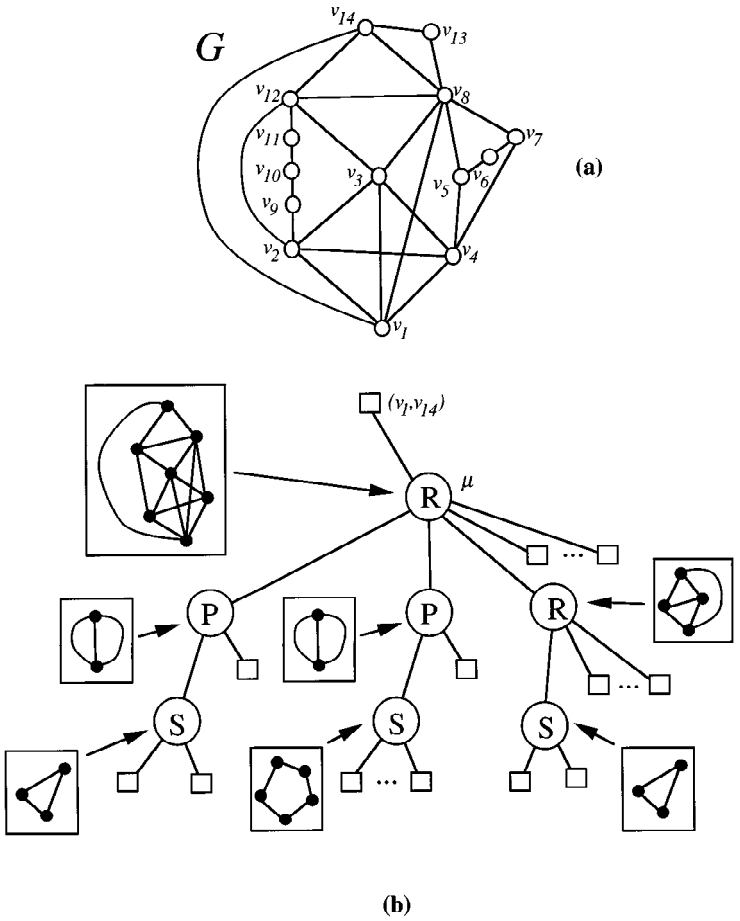

Fig. 1. (a) A biconnected graph G. (b) SPQR-tree $\mathcal{T}$ of G with respect to reference edge (v_1, v_{14}) and skeletons of the P-, S-, and R-nodes.

($k \geq 3$), the root of $\mathcal{T}$ is a P-node μ. Graph *skeleton*(μ) consists of k parallel edges between s and t, denoted $e_1, \ldots, e_k$, with $e_1 = e$.

Series Case: Otherwise, the split pair $\{s, t\}$ has exactly two split components, one of them is the reference edge e, and we denote the other split component by G'. If G' has cutvertices $c_1, \ldots, c_{k-1}$ ($k \geq 2$) that partition G into its blocks $G_1, \ldots, G_k$, in this order from s to t, the root of $\mathcal{T}$ is an S-node μ. Graph *skeleton*(μ) is the cycle $e_0, e_1, \ldots, e_k$, where $e_0 = e$, $c_0 = s$, $c_k = t$, and e_i connects c_{i-1} with c_i ($i = 1, \ldots, k$).

Rigid Case: If none of the above cases applies, let $\{s_1, t_1\}, \ldots, \{s_k, t_k\}$ be the maximal split pairs of G with respect to $\{s, t\}$ ($k \geq 1$), and, for $i = 1, \ldots, k$, let G_i be the union of all the split components of $\{s_i, t_i\}$ but the one containing the reference edge e. The root of $\mathcal{T}$ is an R-node μ. Graph *skeleton*(μ) is obtained from G by replacing each subgraph G_i with the edge e_i between s_i and t_i.

Except for the trivial case, μ has children $\mu_1, \ldots, \mu_k$ in this order, such that μ_i is the root of the SPQR-tree of (multi) graph $G_i \cup e_i$ with respect to reference edge e_i ($i = 1, \ldots, k$). The endpoints of edge e_i are called the *poles* of node μ_i. The tree so obtained has a Q-node associated with each edge of G, except the reference edge e. We complete the SPQR-tree by adding another Q-node, representing the reference edge e, and making it the parent of μ so that it becomes the root.

An example of the SPQR-tree is shown in Figure 1. The maximal split pairs with respect to reference edge (v_1, v_{14}) are:

- Edges (v_1, v_4), (v_1, v_2), (v_2, v_4), (v_1, v_3), (v_4, v_3), (v_2, v_3), (v_1, v_8), (v_3, v_8), (v_3, v_{12}), (v_{12}, v_8), (v_{12}, v_{14}). Each such edge is associated with a Q-node child of R-node μ.
- $\{v_4, v_8\}$, which are the poles of the R-node child of μ.
- $\{v_8, v_{14}\}$, which are the poles of a P-node child of μ.
- $\{v_2, v_{12}\}$, which are the poles of the other P-node child of μ.

The next lemmas follow directly from the above definitions:

LEMMA 1. *Two S-nodes cannot be adjacent in $\mathcal{T}$. Two P-nodes cannot be adjacent in $\mathcal{T}$.*

LEMMA 2. *Let μ be a node of $\mathcal{T}$. We have*:

- *If μ is an R-node, then skeleton(μ) is a triconnected graph.*
- *If μ is an S-node, then skeleton(μ) is a cycle.*
- *If μ is a P-node, then skeleton(μ) is a triconnected multigraph consisting of a bundle of multiple edges.*
- *If μ is a Q-node, then skeleton(μ) is a biconnected multigraph consisting of two multiple edges.*

LEMMA 3. *The skeletons of the nodes of $\mathcal{T}$ are homeomorphic to subgraphs of G. Also, the union of the sets of split-pairs of the skeletons of the nodes of $\mathcal{T}$ is equal to the set of split-pairs of G.*

Let v be a vertex of G. The *allocation nodes* of v are the nodes of $\mathcal{T}$ whose skeleton contains v. The following property is immediate:

LEMMA 4. *Two vertices have a common allocation P-node if and only if they have more than two common allocation nodes.*

The least common ancestor μ of the allocation nodes of v is itself an allocation node of v and is called the *proper* allocation node of v, denoted $\mu = \mathit{proper}(v)$. In the example of Figure 1, vertex v_4 is allocated at the two R-nodes, and has proper allocation node μ. If $v = s$ or $v = t$ (the endpoints of the reference edge) we conventionally define $\mathit{proper}(v)$ as the unique child of the root of $\mathcal{T}$ (recall that the root of $\mathcal{T}$ is the Q-node of the reference edge). If $v \neq s, t$, node $\mu = \mathit{proper}(v)$ is either an R-node or an S-node; also, μ is the only allocation node of v such that v is not a pole of μ. The set of vertices v with proper allocation node μ is denoted $\mathit{properset}(\mu)$. If μ is a (proper) allocation node of v,

we say that v is (properly) allocated at μ. Note that if μ is an S- or R-node, *properset*(μ) is not empty, while if μ is a P- or Q-node, *properset*(μ) is empty unless μ is the unique child of the root.

It is possible to show that SPQR-trees of the same graph with respect to different reference edges are isomorphic and are obtained one from the other by selecting a different Q-node as the root. SPQR-trees are closely related to the classical decomposition of biconnected graphs into triconnected components [5], [12]. Namely, the triconnected components of a biconnected graph G are in one-to-one correspondence with the internal nodes of the SPQR-tree: the R-nodes correspond to triconnected graphs, the S-nodes to polygons, and the P-nodes to bonds. Also, the SPQR-tree extends the notion of tree of triconnected components [12] by introducing the concepts of poles, allocation nodes, and proper allocation nodes, which are crucial to the developments of this paper. Note that our definitions are simpler than the ones originally given by Tutte [12]. SPQR-trees of planar directed graphs were introduced in [1] and applied to the problem of on-line planarity testing. In this paper we extend the concept of SPQR-trees to nonplanar undirected graphs.

LEMMA 5. *The SPQR-tree $\mathcal{T}$ of G has m Q-nodes and $O(n)$ S-, P-, and R-nodes. Also, the total number of vertices of the skeletons stored at the nodes of $\mathcal{T}$ is $O(n)$.*

PROOF. Clearly, there are m Q-nodes. For each S- or R-node μ, we consider a vertex of *properset*(μ). Such vertices are all distinct, so that there are at most n S- and R-nodes. Also, there are at most $n+1$ P-nodes because the parent of a P-node is either an S-node, an R-node, or the Q-node at the root. Finally, since exactly two vertices of the skeleton of a node (the poles) are not properly allocated at that node, the total number of vertices of the skeletons stored at the nodes of $\mathcal{T}$ is $O(n)$. □

3.2. *Queries*

LEMMA 6. *Operation* ThreePaths(v_1, v_2) *returns* true *if and only if there is a P-node or an R-node χ such that v_1 and v_2 are both allocated at χ.*

PROOF. (*If*) By Lemma 2, *skeleton*(χ) is triconnected and hence contains three vertex-disjoint paths between v_1 and v_2. By Lemma 3, *skeleton*(χ) is homeomorphic to a subgraph of G, and thus G contains three vertex-disjoint paths between v_1 and v_2.

(*Only-If*) Assume that there is no P- or R-node where both v_1 and v_2 are allocated. We show that v_1 and v_2 are separated by a pair of vertices or by a vertex and an edge. We consider two cases:

- Vertices v_1 and v_2 are both allocated at an S-node μ.
  If v_1 and v_2 are not adjacent in *skeleton*(μ), then they are separated in G by the pair of vertices neighbor of v_1 (or v_2) in *skeleton*(μ). Else, v_1 and v_2 are both allocated at a Q-node associated with an edge e of G between them, and they are separated by e and by any other vertex of *skeleton*(μ) distinct from v_1 and v_2.
- There are no nodes where v_1 and v_2 are both allocated.
  Let $\mu_1 = \mathit{proper}(v_1)$ and $\mu_2 = \mathit{proper}(v_2)$, and denote with s_i and t_i the poles of μ_i

($i = 1, 2$). If μ_1 is a descendant of μ_2, then v_1 and v_2 are separated by $\{s_1, t_1\}$. If μ_2 is a descendant of μ_1, then v_1 and v_2 are separated by $\{s_2, t_2\}$. Otherwise, v_1 and v_2 are separated by either $\{s_1, t_1\}$ or $\{s_2, t_2\}$. □

The following lemma shows that the condition of Lemma 6 can be efficiently tested if the proper allocation nodes of the vertices are known. It is easily proved by means of a case analysis.

LEMMA 7. *Vertices v_1 and v_2 are allocated at the same P- or R-node χ if and only if one of the following conditions is satisfied*:

- *proper*$(v_1) =$ *proper*$(v_2) = \chi$ (*in this case χ is an R-node or the unique child of the root*).
- v_1 *is a pole of* χ, *with* $\chi =$ *proper*(v_2) (*in this case χ is an R-node*).
- v_2 *is a pole of* χ, *with* $\chi =$ *proper*(v_1) (*in this case χ is an R-node*).
- v_1 *and* v_2 *are the poles of* χ, *and the parent* χ' *of* χ *is an S-node* (*in this case* *proper*$(v_1) = \chi'$ *or* *proper*$(v_2) = \chi'$).

In a static environment operation *ThreePaths* can be efficiently supported with the following data structure:

- The SPQR-tree of G without the Q-nodes children of R-nodes, where each node stores its type, its poles (but not its skeleton), and a pointer to its parent, called the *parent-pointer*. The removal of Q-nodes children of R-nodes eliminates redundant information and reduces the space requirement to $O(n)$.
- For each vertex, a pointer to its proper allocation node. Such pointers are called *proper-pointers*.
- For each vertex v properly allocated at an S-node μ, pointers to the (at most two) children of μ of which v is a pole. Such pointers are called *S-pointers*.

By Lemma 5, this data structure uses $O(n)$ space. Also, it can be constructed in $O(n+m)$ time using a variation of the algorithm given in [5]. The algorithm for operation *ThreePaths* consists of testing the conditions of Lemma 7. This is done by accessing nodes *proper*(v_1), *proper*(v_2), and their parents by using the proper-pointers of v_1 and v_2, and the parent-pointers in $\mathcal{T}$. Also, if *proper*(v_1) is an S-node, we use the S-pointers of v_1 to verify the last condition of Lemma 7, and similarly for v_2. The correctness follows from Lemma 6. We obtain:

THEOREM 1. *Let G be a biconnected graph with n vertices and m edges. There is an $O(n)$-space data structure for G that supports operation* ThreePaths *in $O(1)$ time, and can be constructed in $O(n + m)$ time.*

Note that by storing at each node of the SPQR-tree its distance from root, we can also return a separation pair or a vertex-edge pair whose removal disconnects v_1 from v_2 whenever operation *ThreePaths*(v_1, v_2) returns *false* (see the proof of Lemma 6). We detect the case where v_1 and v_2 are adjacent and allocated at the same S-node μ by using

the S-pointers of v_1 and v_2. In this case two such S-pointers point to the same Q-node, associated with edge (v_1, v_2). The performance bounds are the same as in Theorem 1.

3.3. *Updates*. Our dynamic environment for biconnected graphs consists of performing a sequence of intermixed *ThreePaths*, *InsertEdge*, and *InsertVertex* operations. This repertory of operations is complete for the class of biconnected graphs, since any biconnected graph G with n vertices and m edges can be assembled starting from the triangle graph (a graph consisting of a cycle with three vertices and edges) with a sequence of $n - 3$ *InsertVertex* and $m - 3$ *InsertEdge* operations. Also, such a sequence can be computed in $O(n + m)$ time [2]. Note that if the vertices of G are labeled, the above assembly is performed starting from a triangle graph with properly chosen vertex labels.

The effect of *InsertVertex*(v, v_1, v_2) on the structure of the SPQR-tree is to replace the Q-node of edge (v_1, v_2) with an S-node having children Q-nodes e_1 and e_2. If the new S-node is a child of an S-node, it is absorbed into its parent. If, before operation *InsertVertex*(v, v_1, v_2), edge (v_1, v_2) is the reference edge, then after the operation the reference edge becomes undefined. In this case we choose (v_1, v) as the new reference edge.

In the example of Figure 2, vertex v_{15} is inserted on the reference edge (v_1, v_{14}), and the new reference edge becomes (v_1, v_{15}).

The restructuring of the SPQR-tree caused by *InsertEdge*(v_1, v_2) is more complex (see the example in Figure 3). We describe the restructuring with respect to a nonrooted version of $\mathcal{T}$. Note that the skeletons of the nodes, and hence the allocation nodes of a vertex, do not depend on the choice of root node. By Lemma 6, only the vertices of the skeletons are used to answer operation *ThreePaths*. Hence, we do not need to discuss the updates to the skeleton edges.

Two fundamental restructuring primitives are *merging* R-nodes and *splitting* S-nodes. Merging two R-nodes consists of identifying the nodes and unioning their sets of neighbors and skeleton vertices. Splitting an S-node is defined below.

Let μ be an S-node and let $\mu_0, \ldots, \mu_{k-1}$ $(k \geq 3)$ be the circular sequence of neighbors of μ. Note that, by Lemma 1, nodes $\mu_0, \ldots, \mu_{k-1}$ are not S-nodes. Also, let $v_0, \ldots, v_{k-1}$ be the vertices of *skeleton*(μ), where (v_i, v_{i+1}) is an edge of *skeleton*(μ_i). *Splitting* μ *between* μ_i *and* μ_j consists of replacing μ with two new S-nodes, ν_1 and ν_2, with the circular sequence of neighbors given by $\mu_{i+1} \cdots \mu_{j-1}$ and $\mu_{j+1} \cdots \mu_{i-1}$, respectively. If v_i has no neighbors, then it is removed. *Splitting* μ *between* v_i *and* v_j consists of replacing μ with two new S-nodes, ν_1 and ν_2, with the circular sequence of neighbors given by $\mu_i \cdots \mu_{j-1}$ and $\mu_j \cdots \mu_{i-1}$, respectively. If v_i has no neighbors, then it is removed. *Splitting* μ *between* μ_i *and* v_j is similarly defined.

Now we are ready to describe the restructuring of the (unrooted) SPQR-tree in consequence of operation *InsertEdge*(v_1, v_2). Let λ be a new Q-node, associated with the new edge (v_1, v_2). We distinguish five cases:

1. v_1 and v_2 have exactly one common allocation node μ, which is an R-node.
   We simply add a tree edge between λ and μ.
2. v_1 and v_2 have exactly one common allocation node μ, which is an S-node.
   We split μ between v_1 and v_2, yielding nodes ν_1 and ν_2, and replace it with a new P-node ν adjacent to λ, ν_1, and ν_2.

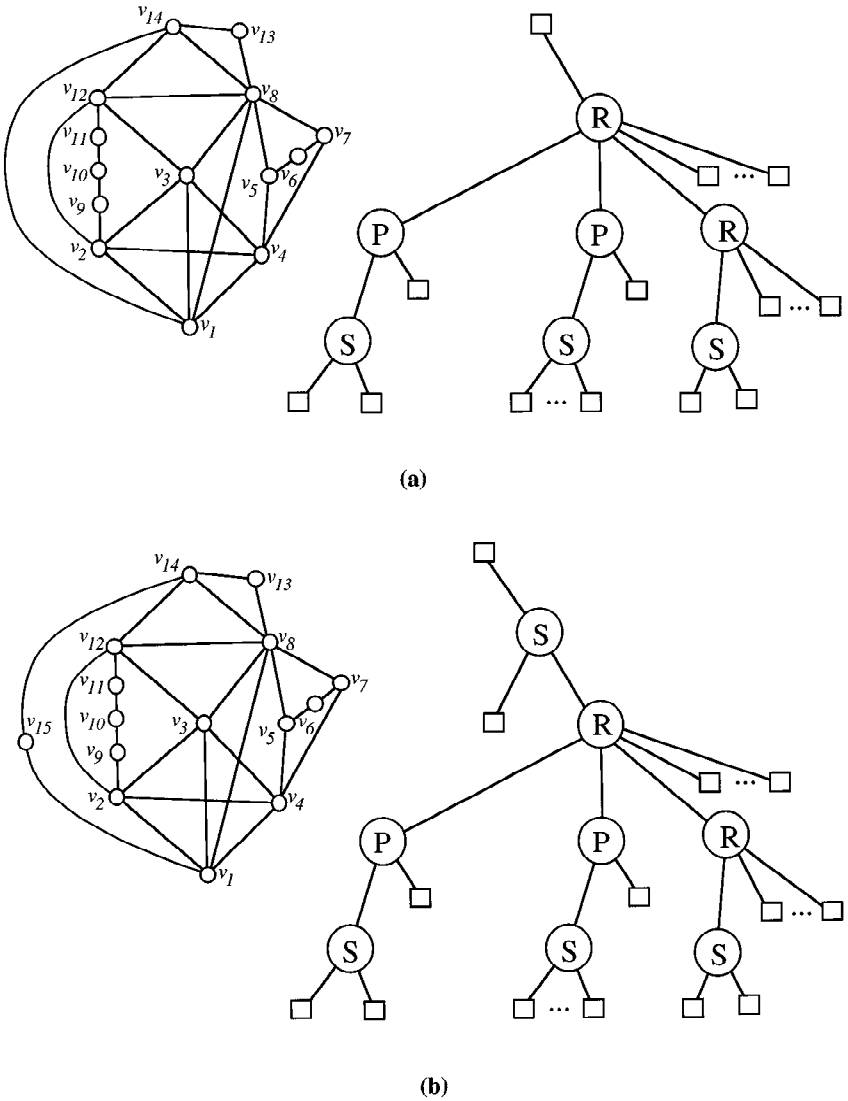

Fig. 2. Example of restructuring of the SPQR-tree in operation *InsertVertex* when the new vertex is inserted on the reference edge: (a) before *InsertVertex*; (b) after *InsertVertex*.

3. v_1 and v_2 have a common allocation P-node μ (such a P-node is unique).
   We simply add a tree edge between λ and μ.
4. v_1 and v_2 have exactly two common allocation nodes μ_1 and μ_2.
   Nodes μ_1 and μ_2 are adjacent in $\mathcal{T}$. We replace the tree edge (μ_1, μ_2) with a new P-node ν adjacent to λ, μ_1, and μ_2.
5. v_1 and v_2 do not have common allocation nodes.
   Let Π be the minimal path in $\mathcal{T}$ whose extreme nodes μ_1 and μ_2 are allocation nodes of v_1 and v_2, respectively.

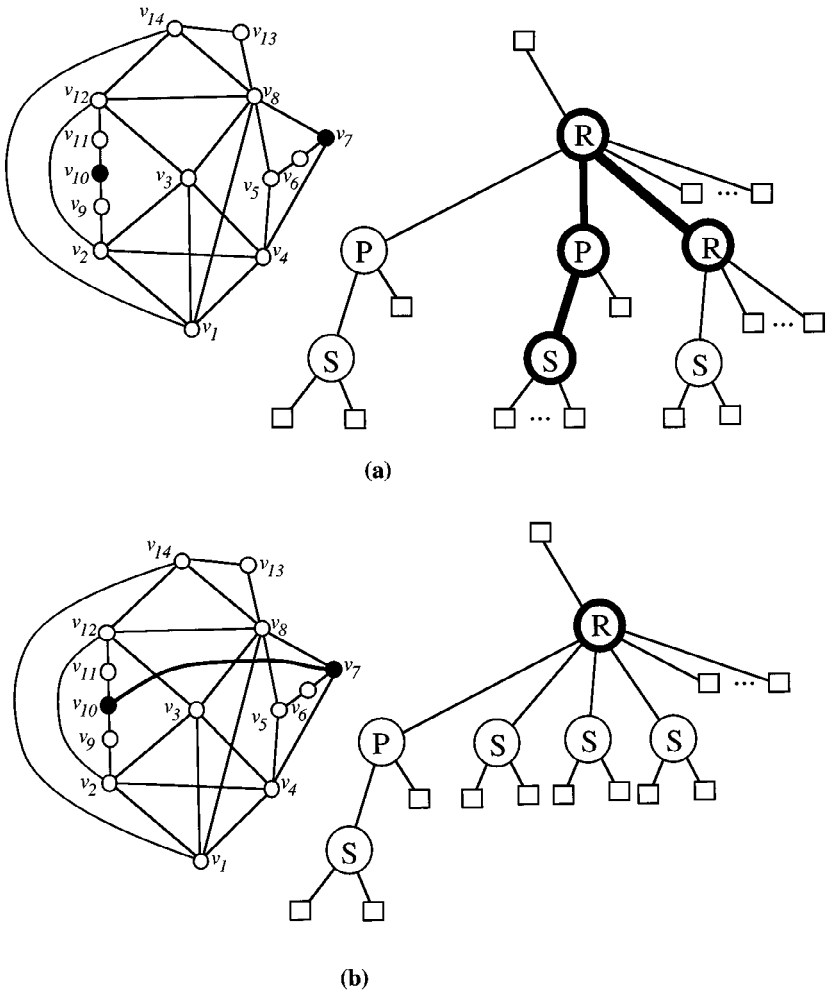

Fig. 3. Example of restructuring of the SPQR-tree in operation *InsertEdge*: (a) before *InsertEdge*; (b) after *InsertEdge*.

(a) Remove all the edges of Π.
(b) Split every S-node μ of Π between its neighbors in Π, or between v_i and the unique neighbor of μ in Π if $\mu = \mu_i$ ($i = 1$ or 2).
(c) Merge all the R-nodes of Π into a new R-node ν, whose skeleton vertices are the union of all the vertices of the skeletons of the R-nodes and P-nodes of Π, plus v_1 and v_2.
(d) Connect to ν, by means of new tree edges, Q-node λ, the former P-nodes of Π, and the new nodes created in Step 5(b).
(e) Absorb the degree-2 S- and P-nodes neighbors of ν into edges.

In the example of Figure 3 we illustrate operation *InsertEdge*(v_7, v_{10}). Part (a) shows the SPQR-tree before the insertion, with path Π drawn with thick lines. Part (b) shows

the SPQR-tree after the insertion. Note the split of the S-node and the absorption of the P-node that are on path Π.

By Lemma 4, the above five cases are exhaustive. Concerning the correctness of Case 5, observe that the split pairs associated with the edges of path Π, except those associated with P-nodes, are no longer split pairs after the insertion.

In the rooted version of the SPQR-tree $\mathcal{T}$, the above restructuring corresponds to performing a sequence of merge and split operations on the sets of children and sets of properly allocated vertices for selected R- and S-nodes of $\mathcal{T}$. Hence, the circular split of an S-node reduces to $O(1)$ standard splits of a linear sequence. The action takes place along a subpath of the path of $\mathcal{T}$ between $\mu_1 = proper(v_1)$ and $\mu_2 = proper(v_2)$.

We measure the complexity of the restructuring in terms of the number of the following *elementary operations*: access to the parent of a node, merge of two sets (of children or of properly allocated vertices), and split of a set (of children or of properly allocated vertices). In the following lemma, the amortization refers to a sequence of updates starting from the triangle graph.

LEMMA 8. *The SPQR-tree can be maintained with $O(1)$ elementary operations per update* (InsertVertex *or* InsertEdge). *The bound is worst case for* InsertVertex *and amortized for* InsertEdge.

PROOF. The bound for *InsertVertex* is immediate from the above description. Regarding *InsertEdge*, the nontrivial case is when v_1 and v_2 do not have common allocation nodes, and the number of elementary operations can be $\Omega(n)$ in the worst case. However, their amortized number is $O(1)$ as the following analysis shows.

We define the *potential* Φ of $\mathcal{T}$ as

$$\Phi = c \cdot |R| + c \cdot \sum_{\mu \in P} deg(\mu),$$

where $|R|$ is the number of R-nodes, P is the set of P-nodes, $deg(\mu)$ is the number of children of node μ, and c is a constant to be determined later.

By Lemma 5, $\Phi = O(n)$. Let N be the number of elementary operations performed in an *InsertEdge* operation, and let $A = N + \Delta\Phi$ be the corresponding amortized quantity.

A constant number of elementary operations is performed at each node of Π, so that $N \leq a \cdot l + b$, where l is the length of path Π, and a and b are constants.

Let R_Π and P_Π be the sets of R- and P-nodes on path Π, respectively. Since, by Lemma 1, there are no two consecutive S-nodes in Π, we have that $|R_\Pi| + |P_\Pi| \geq (l-1)/2$. Since all R-nodes of Π are merged and the degree of all P-nodes of Π is decreased by one, we have that $\Delta\Phi = -c \cdot (|R_\Pi| - 1 + |P_\Pi|) \leq -c \cdot (l-3)/2$.

Hence, the amortized number of elementary operations is

$$A \leq \left(a - \frac{c}{2}\right) \cdot l + b + \frac{3c}{2}.$$

By choosing $c = 2a$, we have that $A = O(1)$. □

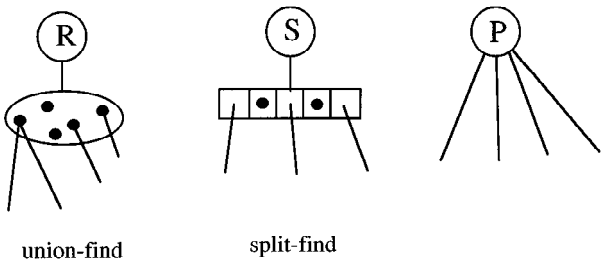

Fig. 4. Representation of the nodes of the SPQR-tree in the dynamic data structure.

We use the following dynamic data structure:

- The SPQR-tree $\mathcal{T}$ of G without the Q-nodes children of R-nodes.
- For each node μ of $\mathcal{T}$, depending on the type of μ, we represent the set of children of μ and the set of properly allocated vertices, *properset*(μ), as follows (see Figure 4):
  - For an R-node we use the *condensible node* structure of [13], where *properset*(μ) is a union-find data structure with amortized time complexity $O(\alpha(k, n))$ [11] per operation ($\alpha(k, n)$ denotes the well-known Ackermann's function inverse). Also, each child of μ has a pointer to an element of *properset*(μ).
  - For an S-node we represent the children and proper vertices (which form a sorted sequence) as a split-find data structure that supports insert, split, and find operations in $O(1)$ amortized time [4], [6].
  - Finally, we use direct pointers for the children of a P-node, whose set *properset*(μ) is empty.

THEOREM 2. *A data structure for biconnected graphs exists that supports a sequence of k operations, each a* ThreePaths, InsertEdge, *or* InsertVertex, *starting from the triangle graph, in time $O(k \cdot \alpha(k, n))$, where n is the number of* InsertVertex *operations. At any time the space requirement is linear in the current number of vertices.*

PROOF. With the above data structure, finding the proper allocation node of a vertex and accessing a parent R- or S-node needs a preliminary find operation and takes $O(\alpha(k, n))$ amortized time. Our analysis of the split and merge operations in the restructuring of the SPQR-tree $\mathcal{T}$ shows that split operations are performed only on S-nodes, while union operations are performed only on R-nodes (see Figure 5). Transfers of children or properly allocated vertices between nodes happens only through insertions and deletions of individual elements (see, e.g., Figure 5(b)).

Hence, the lack of interaction between the union-find and split-find data structures implies an $O(\alpha(k, n))$ amortized time bound for each elementary operation performed in the restructuring of $\mathcal{T}$. By Lemma 8, we conclude that a sequence of k operations, each a *ThreePaths*, *InsertEdge*, or *InsertVertex*, starting from the triangle graph, takes total time $O(k \cdot \alpha(k, n))$. □

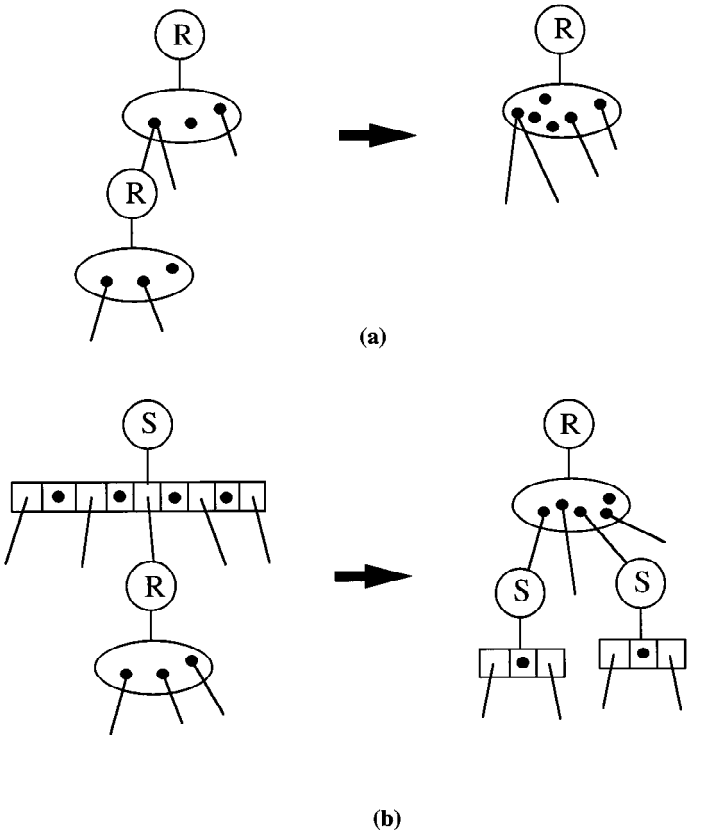

Fig. 5. Examples of merge and split operations: (a) Merge of R-nodes. (b) Split of an S-node. Note that the poles of the R-node, formerly in the proper allocation set of the S-node, are transfered to the proper allocation set of the R-node.

4. General Graphs. In this section we extend our techniques to general (nonbiconnected) graphs. First, we consider connected graphs, and then nonconnected graphs.

4.1. *BC-Tree*. Let G be a connected graph with n vertices. The *BC-tree* $\mathcal{B}$ of G has a B-node for each block (biconnected component) of G, and a C-node for each cutvertex of G. Edges in $\mathcal{B}$ connect each B-node μ to the C-nodes associated with the cutvertices in the block of μ. The BC-tree is rooted at an arbitrary B-node. Also, the B-node of each nontrivial block B stores the SPQR-tree of B. Observe that the number of blocks of G is $O(n)$, and the total number of vertices in the blocks of G is $O(n)$ as well.

The BC-tree is a variation of the data structures for maintaining biconnected components described in [9] and [13]. The innovation introduced here is attaching an SPQR-tree at each B-node.

If vertex v is a cutvertex, $bcproper(v)$ denotes the C-node associated with v. Otherwise, $bcproper(v)$ denotes the B-node of the unique block containing v. It is easy to see that, knowing $\mu_1 = bcproper(v_1)$ and $\mu_2 = bcproper(v_2)$, we can determine in $O(1)$

time whether v_1 and v_2 are in the same block of G [9]: namely the block associated with node μ contains vertices v_1 and v_2 if and only if the undirected path of $\mathcal{B}$ between μ_1 and μ_2 contains μ but no other B-node. This is equivalent to testing whether one of the following eight mutually exclusive conditions is verified:

- $\mu = \mu_1 = \mu_2$ (neither v_1 nor v_2 are cutvertices).
- $\mu = \mu_1$ and μ_2 is the parent of μ (v_1 is not a cutvertex, and v_2 is a cutvertex).
- $\mu = \mu_2$ and μ_1 is the parent of μ (v_1 is a cutvertex, and v_2 is not a cutvertex).
- $\mu = \mu_1$ and μ is the parent of μ_2 (v_1 is not a cutvertex, and v_2 is a cutvertex).
- $\mu = \mu_2$ and μ is the parent of μ_1 (v_1 is a cutvertex, and v_2 is not a cutvertex).
- μ is the parent of μ_1 and μ_2 (both v_1 and v_2 are cutvertices).
- μ is the parent of μ_1, and μ_2 is the parent of μ (both v_1 and v_2 are cutvertices).
- μ is the parent of μ_2, and μ_1 is the parent of μ (both v_1 and v_2 are cutvertices).

4.2. *Updates*. We add to our repertory of updates, operation *AttachVertex*. The restructuring of the BC-tree in consequence of operation *AttachVertex*(v, u) consists of adding a new B-node for the trivial block (u, v), and a new C-node for vertex u if it was not formerly a cutvertex.

We now examine the structural changes of the BC-tree when operation *InsertEdge*(v_1, v_2) is performed on G. If v_1 and v_2 are in the same block B of G, then the primary structure of the BC-tree stays unchanged, and we process the insertion in the secondary structure (SPQR-tree) of the B-node of B. Otherwise, the effect of *InsertEdge* is to merge the "old blocks" corresponding to the B-nodes on the path Π between nodes $\mu_1 = bcproper(v_1)$ and $\mu_2 = bcproper(v_2)$ into a "new block." The primary structure of $\mathcal{B}$ is updated by means of a sequence of merge operations. To update the secondary structure, we need to construct efficiently the SPQR-tree of the new block from the SPQR-trees of the old blocks, as follows:

1. Create a new Q-node λ for the newly inserted edge (v_1, v_2).
2. For each B-node μ of Π, let B be the block of μ (B is an old block), and let u and v be the vertices of B associated with the C-nodes neighbors of μ in Π. If $\mu = \mu_i$, then let $u = v_i$, and let v be the vertex associated with the unique neighbor C-node of μ in Π ($i = 1, 2$). We add a fictitious edge e_B between u and v in B, and update the SPQR-tree of B accordingly. Let μ_B be the Q-node of the fictitious edge e_B.
3. Let B^* be the old block of maximum size (number of vertices). For each old block $B \neq B^*$, we reroot the SPQR-tree of B at Q-node μ_B.
4. Replace Q-node μ_{B^*} with a new S-node ν, and connect it to λ and to node μ_B for each old block $B \neq B^*$. The order of the children of ν is given by the sequence of B-nodes along Π.
5. Absorb into tree edges the degree-2 Q-nodes of the fictitious edges.

Informally speaking, the correctness of the above algorithm is justified by the fact that the old blocks plus the newly inserted edge form a "ring," which is represented by an S-node. The reason for rerooting all the old blocks but the largest one is that each vertex is involved in no more than a logarithmic number of rerootings, since after each rerooting the size of its block at least doubles.

Rerooting the SPQR-tree of a block B can be easily done in time proportional to

the number of vertices of B. Clearly, the number of R-nodes and the degrees of the P-nodes is not affected by the rerooting so that the potential of the SPQR-tree of block B, as defined in the proof of Lemma 8, stays unchanged. Note that although the allocation nodes of each vertex remain the same, the proper allocation node of each vertex in general changes. This implies a rebuilding of the union-find and split-find data structures, which can also be done in time proportional to the number of vertices of B.

In the following we determine the time complexity of maintaining the BC-tree in a sequence of *ThreePaths*, *InsertEdge*, *InsertVertex*, and *AttachVertex* operations. We perform a separate amortized analysis for the number N of elementary operations performed, and the time t to rebuild the SPQR-trees of the blocks.

We denote the old blocks with $B_1, \ldots, B_k$, where $B_k = B^*$, and let $n_i + 1$ be the number of vertices of block B_i. Note that the new block has $\sum_{i=1}^{k} n_i + 1$ vertices.

The number N of elementary operations performed in operation *InsertEdge* is

$$N = \sum_{i=1}^{k} N_i,$$

where N_i is the number of elementary operations performed in block B_i.

Let Φ_{B_i} be the potential of the SPQR-tree of block B_i, as defined in the proof of Lemma 8. By the proof of Lemma 8, we have

$$N_i + \Delta\Phi_{B_i} \leq d,$$

where d is a constant.

We define potential function Φ_1 for graph G as follows:

$$\Phi_1 = \sum_{\text{all blocks } B} \Phi_B + d|\mathcal{B}|,$$

where $|\mathcal{B}|$ denotes the number of edges of the BC-tree $\mathcal{B}$ and Φ_B is the potential of the SPQR-tree of block B, as defined in the proof of Lemma 8. Note that $\Phi_1 = O(n)$.

Since $k - 1$ edges of the BC-tree $\mathcal{B}$ are contracted as a consequence of operation *InsertEdge*, we conclude that the amortized number A of elementary operations performed in *InsertEdge* is

$$A = T + \Delta\Phi_1 = O(1).$$

The amortized analysis of the remaining operations is straightforward. Thus, in a sequence of l *ThreePaths*, *InsertEdge*, *InsertVertex*, and *AttachVertex* operations, $O(l)$ elementary operations are performed.

Now we turn to the amortized analysis of the rebuilding time. We define the potential function Φ_2 of G as

$$\Phi_2 = \sum_{\text{all blocks } B} n_B \log \frac{1}{n_B},$$

where $n_B + 1$ is the number of vertices of block B.

We have that $\Phi_2 \leq 0$ and $|\Phi_2| = O(n \log n)$. With an appropriate choice of the time unit, the rebuilding time t in operation *InsertEdge* is given by

$$t = 2(n_1 + \cdots + n_{k-1}).$$

The variation of potential caused by *InsertEdge* is given by

$$\Delta\Phi_2 = (n_1 + \cdots + n_k)\log\frac{1}{n_1 + \cdots + n_k} - \sum_{i=1}^{k} n_i \log\frac{1}{n_i}.$$

LEMMA 9. *Consider the function* $f(x) = x\log(1/x)$, *and let* $1 \le x_1 \le \cdots \le x_k$. *We have*

$$f(x_1 + \cdots + x_k) - (f(x_1) + \cdots + f(x_k)) \le -2(x_1 + \cdots + x_{k-1}).$$

PROOF. By induction on k. The base case ($k = 2$) is easy to prove by a simple analysis of the binary entropy function $h(x) = f(x) + f(1-x)$ for $0 < x < 1$. For the inductive step, we have

$$\begin{aligned}
&f(x_1 + \cdots + x_k) - (f(x_1) + \cdots + f(x_k)) \\
&\quad = f(x_1 + (x_2 + \cdots + x_k)) - f(x_2 + \cdots + x_k) + f(x_2 + \cdots + x_k) \\
&\qquad - (f(x_1) + \cdots + f(x_k)) \qquad (2)\\
&\quad = (f(x_1 + (x_2 + \cdots + x_k)) - (f(x_1) + f(x_2 + \cdots + x_k))) \\
&\qquad + (f(x_2 + \cdots + x_k) - (f(x_2) + \cdots + f(x_k))) \qquad (3)\\
&\quad \le -2x_1 - 2(x_2 + \cdots + x_{k-1}) \qquad (4)\\
&\quad = -2(x_1 + \cdots + x_{k-1}).
\end{aligned}$$

□

By Lemma 9, we have that the amortized rebuilding time a of operation *InsertEdge* is

$$a = t + \Delta\Phi_2 = 0.$$

The remaining operations can only decrease the potential Φ_2. Hence, recalling that $-\Phi_2 = O(n \log n)$, we have that in a sequence of l *ThreePaths*, *InsertEdge*, *InsertVertex*, and *AttachVertex* operations, the total rebuilding time is $O(n \log n)$. We conclude that a sequence of k operations, each a *ThreePaths*, *InsertEdge*, *InsertVertex*, or *AttachVertex*, takes time $O(n \log n + k\alpha(k, n)) = O(n \log n + k)$.

THEOREM 3. *A data structure for connected graphs exists that supports a sequence of k operations, each a* ThreePaths, InsertEdge, InsertVertex, *or* AttachVertex, *starting from a single vertex, in time* $O(n \log n + k)$, *where n is the number of* InsertVertex *and* AttachVertex *operations. At any time the space requirement is linear in the current number of vertices.*

4.3. *Nonconnected Graphs.* For graphs that are not connected, we add *MakeVertex* to the repertory of update operations, and consider the *BC-forest*, which is the forest of the BC-trees of the connected components. When an *InsertEdge* operation joins two components, we rebuild the BC-tree of the connected component of smaller size, so that it becomes a subtree of the BC-tree of the larger component. We can charge the rebuilding time to the vertices of the graph. Each time a vertex is involved in a rebuilding operation, the size of its connected component at least doubles. Hence, the total rebuilding time is $O(n \log n)$. Recalling Theorem 3, we obtain:

THEOREM 4. *A data structure for general graphs exists that supports a sequence of k operations, each a* ThreePaths, InsertEdge, InsertVertex, AttachVertex, *or* MakeVertex, *starting from the empty graph, in time $O(n \log n + k)$, where n is the number of* InsertVertex, AttachVertex, *and* MakeVertex *operations. At any time the space requirement is linear in the current number of vertices.*

Very recently La Poutré has shown that the time bound of Theorem 4 can be reduced to $O(k \cdot \alpha(k, n))$ [8].

References

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

[2] G. Di Battista and R. Tamassia, On-Line Planarity Testing, *SIAM Journal on Computing*, **25**(5) (1996), to appear.

[3] D. Fussell, V. Ramachandran, and R. Thurimella, Finding Triconnected Components by Local Replacements, *Automata, Languages and Programming* (*Proc*. 16*th ICALP*), Lecture Notes in Computer Science, Vol. 372, Springer-Verlag, Berlin, 1989, pp. 379–393.

[4] H. N. Gabow and R. E. Tarjan, A Linear Time Algorithm for a Special Case of Disjoint Set Union, *J. Comput. Systems Sci*., **30** (1985), 209–221.

[5] J. Hopcroft and R. E. Tarjan, Dividing a Graph into Triconnected Components, *SIAM J. Comput*., **2** (1973), 135–158.

[6] H. Imai and T. Asano, Dynamic Orthogonal Segment Intersection Search, *J. Algorithms*, **8** (1987), 1–18.

[7] A. Kanevsky, A Characterization of Separating Pairs and Triplets in a Graph, *Congress. Numer.*, **74** (1990), 213–232.

[8] J. A. La Poutré, Maintenance of Triconnected Components of Graphs, *Automata, Languages and Programming* (*Proc*. 19*th ICALP*), Lecture Notes in Computer Science, Vol. 623, Springer-Verlag, Berlin, 1992, pp. 354–365.

[9] R. Tamassia, On-Line Planar Graph Embedding, *J. Algorithms* (to appear). (Preliminary version in *Proc*. 15*th ICALP*, Lecture Notes in Computer Science, Vol. 317, Springer-Verlag, Berlin, 1988, pp. 576–590.)

[10] R. E. Tarjan, Amortized Computational Complexity, *SIAM J. Algebraic Discrete Methods*, **6** (1985), 306–318.

[11] R. E. Tarjan and J. van Leeuwen, Worst-Case Analysis of Set-Union Algorithms, *J. Assoc. Comput. Mach*., **31** (1984), 245–281.

[12] W. T. Tutte, *Graph Theory*, Encyclopedia of Mathematics and Its Applications,Vol. 21, Addison-Wesley, Reading, MA, 1984.

[13] J. Westbrook and R. E. Tarjan, Maintaining Bridge-Connected and Biconnected Components On-Line, *Algorithmica*, **7** (1992), 433–464.