On-Line Maintenance of the Four-Connected Components of a Graph*

(Extended Abstract)

Arkady Kanevsky
Department of Computer Science
Texas A&M University
College Station, TX 77843

Giuseppe Di Battista
Dip. Informatica e Sistemistica
Università di Roma "La Sapienza"
Roma, Italy 00198

Roberto Tamassia
Department of Computer Science
Brown University
Providence, R.I. 02912-1910

Jianer Chen
Department of Computer Science
Texas A&M University
College Station, TX 77843

Abstract

Given a graph G with n vertices and m edges, a k-connectivity query for vertices v' and v'' of G asks whether there exist k disjoint paths between v' and v''. Answering such queries has important applications to network reliability. In this paper we consider the problem of performing k-connectivity queries for $k \leq 4$. First, we present a static data structure that answers such queries in $O(1)$ time. Next, we consider the problem of performing queries intermixed with on-line updates that insert vertices and edges. For triconnected graphs we give a dynamic data structure that supports queries and updates in time $O(\alpha(\ell, n))$ amortized, where n is the current number of vertices of the graph and ℓ is the total number of operations performed ($\alpha(\ell, n)$ denotes the slowly growing Ackermann's function inverse). For general graphs, a sequence of ℓ operations takes total time $O(n \log n + \ell)$. All of the above data structures use space $O(n)$, proportional to the number of vertices of the graph. Our results also yield an efficient algorithm for testing whether graph G is four-connected that runs in $O(n \cdot \alpha(n, n) + m)$ time using $O(n + m)$ space.

*Research supported in part by Cadre Technologies Inc., by the ESPRIT II Basic Research Actions Program of the EC under Contract No. 3075 (project ALCOM), by the National Science Foundation under grant CCR-9007851, by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-83-K-0146 and ARPA order 6320, amendment 1, by the Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo of the Italian National Research Council, by Texas A&M TEES Excellence grants, and by the U.S. Army Research Office under grant DAAL03-91-G-0035.

1 Introduction and Overview

Connectivity is a fundamental property of graphs and much effort has been devoted to devise efficient algorithms for connectivity problems. A graph G is k-connected if between any two vertices there are k disjoint paths. Equivalently, G remains connected after removing any $k-1$ vertices or edges [22]. Assume that G is k-connected but not $(k+1)$-connected; we say that G has connectivity k. A $(k+1)$-connected component C of G is defined as a maximal subset of vertices of G such that any pair of vertices in C can be linked by $k+1$ disjoint paths. A separating k-set S of G is a subset of k vertices or edges whose removal disconnects G. The concepts of $(k+1)$-connected components and separating k-sets are of central importance in fault tolerance of communication networks. In the rest of this paper we shall denote with n and m the number of vertices and edges of the graph currently being considered, respectively.

In a static environment it is possible to determine the connectivity of a graph using a reduction to a collection of network flow problems. The fastest algorithm runs in time $O(\max(k, n^{1/2})kmn^{1/2})$ [10, 13], where n and m denote the number of vertices and edges of the graph, respectively. For small values of k, more efficient algorithms are available. Namely, testing 1-, 2- or 3-connectivity takes time $O(n+m)$ [14, 23, 28], and testing 4-connectivity takes time $O(n^2)$ [19].

We consider the problem of maintaining on-line the k-connected components of a graph that is updated by inserting vertices and edges. Namely, we investigate

how to perform a sequence of intermixed updates and k-connectivity queries, where a k-connectivity query for vertices v' and v'' asks whether v' and v'' are in the same k-connected component, i.e., whether there exist k disjoint paths between v' and v''.

Previously, this problem was efficiently solved only for $k \leq 3$. For $k = 1$, the classical union-find data structure [29] supports a sequence of ℓ operations in time $O(\ell \cdot \alpha(\ell, n))$, where n is the total number of vertex insertions. (Here, $\alpha(\ell, n)$ denotes the slowly growing Ackermann's function inverse [29].) The space requirement is $O(n)$, which does not depend on the number of inserted edges. This result is extended to the on-line maintenance of the biconnected components of a general graph in [30] and of the triconnected components of a biconnected graph in [4]. It is also shown in [4] that the triconnected components of a general graph can be maintained on-line using $O(n)$ space such that a sequence of ℓ operations takes time $O(n \log n + \ell)$.

The main results of this paper are summarized as follows. First, we present an $O(n)$-space static data structure that answers four-connectivity queries in $O(1)$ time. Next, we consider the problem of performing four-connectivity queries intermixed with on-line updates that insert vertices and edges. For triconnected graphs we give a dynamic data structure that supports a sequence of ℓ operations (each a query or an update) in time $O(\ell \cdot \alpha(\ell, n))$, where n is the current number of vertices of the graph. For general graphs, a sequence of ℓ operations takes time $O(n \log n + \ell)$. All of the above data structures use space $O(n)$, proportional to the number of vertices of the graph. Our results also yield an efficient algorithm for testing whether graph G is four-connected that runs in $O(n \cdot \alpha(n, n) + m)$ time using $O(n + m)$ space.

Our dynamic technique uses and extends the combinatorial results of [16, 17, 18] and provides new insights on the structure of four-connected components. Namely, we show that the four-connected components interact by means of two fundamental structures, the "wheel" and the "flower", and can be associated with the nodes of a decomposition tree whose edges represent "canonical" separating triplets. The amortized analysis of the update algorithm shows that the evolution of the four-connected components exhibits "entropic" properties, where the amount of work performed by the algorithm is compensated by a corresponding simplification of the structure.

Other previous results on dynamic graph connectivity are as follows. Fully dynamic data structures for maintaining k-connected components under both insertions and deletions of vertices and edges are presented in [6] and [12]: the connected components of a general graph can be maintained in $O(\sqrt{n})$ time per operation using $O(n + m)$ space [6]; the biconnected components of a planar graph can be maintained in $O(n^{2/3})$ time using $O(n)$ space [12]. Data structures that maintain the connected components of a graph under a sequence of vertex and edge deletions are presented in [5, 27]. The related problems of maintaining the 2- and 3-edge-connected components are studied in [7, 11, 30].

Open problems include the on-line maintenance of k-connected components for all values of k, and developing algorithms for testing k-connectivity that are based upon them.

2 Decomposition of a Triconnected Graph

Let G be a k-connected graph. A *separating k-set* of G is a k-tuple $\tau = \{t_1, \cdots, t_k\}$, where each t_i is a vertex or an edge, such that removing τ disconnects G. Separating 1-sets, 2-sets and 3-sets are called separating elements, pairs, and triplets, respectively.

A *bridge* of a separating k-set τ is either a single edge connecting two vertices of τ (called a *trivial bridge*), or a maximal subgraph B of G such that no edge of B connects two vertices of τ, and any two vertices of B can be joined by a path with no edges or intermediate vertices in τ. We establish the convention that if a vertex t_i of τ has a unique incident edge e in bridge B, then B contains e but not t_i. Hence, we allow a bridge to have *hanging edges* with only one endpoint in the bridge. The *attachments* of bridge B are the vertices of τ in B and the hanging edges connecting B to τ. Note that τ can have more than two nontrivial bridges only if it consists of three vertices.

In the following, we provide some structural properties of the separating triplets of a triconnected graph G. (Some of the concepts can be extended to the separating k-sets of a k-connected graph.) Let B be a bridge of a separating triplet of G. Since G is triconnected, bridge B is biconnected (disregarding the hanging edges) unless it has less than three vertices. Therefore, we have the following facts:

Fact 1 *Let t be a vertex of a separating triplet τ, and let B be a bridge with attachments t, a', and a''. There are disjoint paths in B between t and a' and between t and a''.*

Fact 2 *Let t' and t'' be vertices of a separating triplet τ, and let B be a bridge containing t' and t''. There*

are at least two disjoint paths in B between t' and t''.

Note that Facts 1–2 do not apply when a vertex of a separating triplet τ, say vertex t', has a unique edge e connecting it to a bridge B. Indeed, by our definition of bridge, B contains edge e but not vertex t'.

A separating triplet τ is said to be *redundant* if it has only two nontrivial bridges and there is another separating triplet τ' that is obtained from τ by replacing a vertex of τ with an edge incident on it. Clearly, if any two vertices v' and v'' are separated by a triplet, then they are also separated by an nonredundant triplet. Note that the attachments of a bridge form an nonredundant triplet. Redundant separating pairs and elements can be similarly defined.

Let τ be an nonredundant separating triplet of G, and assume that τ has two bridges B' and B''. A *cross separating triplet* with respect to τ is a separating triplet σ with one vertex in $B' - \tau$ and one vertex in $B'' - \tau$. The following theorem, which simplifies and unifies some results of [17], provides a structural characterization of the cross separating triplets.

Theorem 1 *All the nonredundant cross separating triplets of an nonredundant separating triplet τ have a vertex of τ in common.*

By Theorem 1, a nonredundant triplet τ and its nonredundant cross separating triplets form a *wheel*, which decomposes G into *sectors*. Each sector is delimited by a nonredundant triplet that has no nonredundant cross separating triplets. The vertex common to τ and its cross separating triplets is called *center* of the wheel.

Now, we describe a recursive decomposition of a triconnected graph with respect to a set of "canonical" separating triplets. The decomposition is described by a tree with four types of nodes. Each node μ of the tree is associated with a subgraph of G, called the *skeleton* of μ. Each edge of the tree from a node μ to its parent ν is associated with an nonredundant separating triplet τ that contains precisely the vertices and edges common to the skeletons of nodes μ and ν. The elements of triplet τ are called the *poles* of node μ.

In the following, we allow graphs to have "hanging" edges, i.e., edges without one of the endpoints, and "suspended" edges, i.e., edges without both endpoints. Consider a graph G and a triplet $\tau = \{t_1, t_2, t_3\}$, where each t_i is a vertex or a hanging edge of G. We denote with $\bar{\tau}$ the triplet obtained by replacing each hanging edge of τ with its (unique) endpoint vertex. We assume that the graph obtained from G by connecting the vertices of $\bar{\tau}$ to a new vertex is triconnected.

The *decomposition tree* $\mathcal{T}$ of G with respect to τ is an ordered rooted tree. Let ρ be the root of $\mathcal{T}$. The poles of ρ are the elements of τ. The type and the subtrees of node ρ are recursively defined as follows (see Figs. 1–4).

Terminal Decomposition:
If τ consists of three edges and has a unique bridge consisting of a single vertex v, then ρ is a *T-node* and has no children.

Flower Decomposition:
If the triplet $\bar{\tau}$ has two or more nontrivial bridges, then ρ is an *F-node*. There is a subtree of ρ for each bridge S of $\bar{\tau}$, recursively defined as the decomposition tree of S with respect to σ, where σ is the triplet of attachments of S to $\bar{\tau}$. Also, ρ has a child T-node for every vertex of $\bar{\tau}$ with degree 3.

Wheel Decomposition:
If the former cases do not apply, let B be the (unique) nontrivial bridge of τ. If $\bar{\tau}$ has a vertex c that forms one or more separating pairs of B, then all such pairs have only two nontrivial bridges. Let C be the set consisting of

- vertex c;
- the edges forming with c a separating pair of B;
- the vertices forming with c a separating pair of B, excluding the endpoints of the above edges; and
- the two elements of τ distinct from c or from the hanging edge of τ incident on c.

The nontrivial bridges of B with respect to C and the elements of $C - \{c\}$ are linearly ordered "around" vertex c. In this case, ρ is a *W-node*. For each nontrivial bridge S of B with respect to C there is a subtree of ρ, recursively defined as the decomposition tree of S with respect to σ, where σ is the nonredundant triplet that separates S. The *wheel* W of μ is defined as the graph with vertex set C and with edge set $\{(c,t)|t \neq c\} \cup \{(t',t'')|t'$ and t'' are consecutive in $C\}$. Hence, c is the center of the wheel and each subtree of ρ is associated with a sector of wheel W. Note that there are cases where more than one vertex of $\bar{\tau}$ can be the center c of the wheel.

Rigid Decomposition:
If none of the previous cases applies, then the root ρ of $\mathcal{T}$ is an *R-node*. Consider all the nonredundant separating triplets of G that separate a maximal subgraph of G. For each such separating triplet σ and subgraph S there is a subtree of σ,

recursively defined as the decomposition tree of S with respect to σ.

The skeleton of node ρ consists of the vertices and edges of G that are not in the subgraphs associated with the subtrees of ρ, plus the poles of the children of ρ. For a W-node the order of the children is given by the ordering of the corresponding subgraphs in the wheel. For R- and F-nodes the order is arbitrary. Examples of decomposition trees are given in Figs. 1, 2, 3, and 4.

Note that the skeleton of a node may have hanging edges (e.g., edges e_1, e_2, e_3, e_4, e_5, and e_6 in Fig. 1, and edges e_1, e_2, e_4, e_8, e_{10}, and e_{12} in Fig. 4) or "suspended" edges, i.e., edges without both endpoints (e.g., edge e_1 in Fig. 3, and edges e_2, e_5, e_6, e_7, e_{10}, and e_{11} in Fig. 4).

The *allocation nodes* of a vertex or edge x of G are the nodes whose skeleton contains x. In the example of Fig. 1 vertex v_{12} has three allocation nodes, while vertex v_7 has a unique allocation node.

Lemma 1 *The decomposition tree $\mathcal{T}$ of G with respect to a triplet τ has $O(n)$ nodes. Also, the total size of the skeletons of the nodes of $\mathcal{T}$ is $O(n+m)$, where n and m denote the number of vertices and edges of G.*

Lemma 2 *The least-common ancestor of any two allocation nodes of a vertex or edge x is itself an allocation node of x. Also, the least-common ancestor μ of all the allocation nodes of x is the unique allocation node of x of which x is not a pole, unless μ is the root.*

By Lemma 2, we have that the allocation nodes of x form a subgraph of $\mathcal{T}$ that is a rooted tree. The root of such tree, i.e., the least-common ancestor μ of the allocation nodes of x is called the *proper* allocation node of x, denoted $\mu = proper(x)$. In the examples of Figs. 1–4 the vertices and edges for which μ is the proper allocation node are placed either next to μ or next to an edge between μ and a child of μ. In Fig. 1 the proper allocation node of vertex v_{12} is the W-node, the proper allocation node of vertex v_7 is the R-node parent of the F-node, and the vertices and edges properly allocated at the W-node are v_{11}, v_{12}, e_2, e_3, e_4, e_5, and e_6. In Fig. 3, the proper allocation node of edge e_1 is the top R-node, while the proper allocation nodes of vertices v_6 and v_9 (the endpoints of e_1) are the two bottom R-nodes.

Given elements a' and a'', the set of nodes of $\mathcal{T}$ whose skeleton contains both a' and a'' is given by the intersection of two rooted trees subgraphs of $\mathcal{T}$, which is also a rooted tree (possibly empty). As shown in the following lemma, the root of this tree can be easily found knowing the proper allocation nodes of a' and a'' and their poles.

Lemma 3 *Elements a' and a'' of G are in the skeleton of the same node of $\mathcal{T}$ if and only if one of the following conditions is verified for some node μ of $\mathcal{T}$:*

- $\text{proper}(a') = \text{proper}(a'') = \mu$; *or*
- $\text{proper}(a') = \mu$ *and a'' is a pole of μ; or*
- $\text{proper}(a'') = \mu$ *and a' is a pole of μ.*

Also, node μ is the highest node of $\mathcal{T}$ whose skeleton contains both a' and a''.

3 Static Data Structure

Let G be a triconnected graph with n vertices and m edges. In this section we consider the problem of determining whether two given vertices of G are in the same four-connected component, i.e., we want to support the following 4-connectivity query operation on G.

FourPaths(v', v''): determine whether G has four disjoint paths between v' and v''. This operation returns a yes/no answer.

We describe an $O(n)$-space data structure for G that supports operation *FourPaths*(v', v'') in $O(1)$ time.

The following three lemmas, based on Facts 1–2, show how the decomposition tree $\mathcal{T}$ is related to the the four-connected components of G.

Lemma 4 *If v' and v'' are vertices of the skeleton of an F-node, then query operation* FourPaths(v', v'') *returns "yes".*

Lemma 5 *If v' and v'' are vertices of the skeleton of an R-node, then query operation* FourPaths(v', v'') *returns "yes".*

Lemma 6 *If v' and v'' are vertices of the skeleton of a W-node μ, then query operation* FourPaths(v', v'') *returns "yes" if and only if v' and v'' are adjacent vertices of the wheel of μ.*

Theorem 2 *Let v' and v'' be vertices of a triconnected graph G, and let $\mathcal{T}$ be the decomposition tree of G. Query operation* FourPaths(v', v'') *returns "yes" if and only if*

1. *v' and v'' are in the skeleton of the same node μ of $\mathcal{T}$; and*
2. *if μ is a W-node, then v' and v'' are adjacent in the wheel of μ.*

Sketch of Proof: Assume that vertices v' and v'' are not in the skeleton of the same node, and let $\mu' = proper(v')$ and $\mu'' = proper(v'')$. By Lemma 3, v'

and v'' are separated by the poles of μ' if μ' is not an ancestor of μ'', and are separated by the poles of μ'' if μ'' is not an ancestor of μ'. If v' and v'' are in the skeleton of the same node μ, then we apply one of Lemmas 4, 5, or 6, depending on the type of μ (note that μ cannot be a T-node). □

Now, we illustrate Theorem 2 by means of examples. In Fig. 1, operation *FourPaths* returns "yes" for pair (v_{11}, v_{12}) and returns "no" for pair (v_3, v_5) and for a pair (u', u'') with u' in S_5 and u'' in S_6. In Fig. 3, operation *FourPaths* returns "yes" for pair (v_8, v_9) and returns "no" for pairs (v_1, v_6) and (v_1, v_9). In Fig. 4, operation *FourPaths* returns "yes" for pair (v_6, v_9) (an interesting case) and returns "no" for pair (v_5, v_9).

The data structure for operation *FourPaths* uses the decomposition tree $\mathcal{T}$ of G, where each node of $\mathcal{T}$ stores the triplet of its poles. Also, each W-node stores its wheel by means of adjacency lists. The data structure is accessed from the outside by pointers from each vertex v to its proper allocation node μ in $\mathcal{T}$. If μ is a W-node, then v points to the representative of v in the wheel of μ. Note that we do not store the skeletons of the nodes of $\mathcal{T}$. Since the size of the wheel of a W-node μ is proportional to the number of children of μ, by Lemma 1 the data structure uses $O(n)$ space. In fact, only $O(n)$ edges of G, i.e., those that are poles, are represented in the data structure.

The algorithm for operation *FourPaths*(v', v'') is based on Theorem 2. We access the proper allocation nodes μ' of v' and μ'' of v'' by following pointers in $O(1)$ time. We determine whether Condition 1 of Theorem 2 is verified using Lemma 3. If Condition 1 is not verified, we return the answer "no". If Condition 1 is verified, then if μ is an R-node or an F-node, we return the answer "yes", else (μ is a W-node) we test Condition 2 using the wheel of μ. Since only one vertex of the wheel (the center) can have degree > 3 (in the wheel), we can test adjacencies in the wheel in $O(1)$ time. The construction of the static data structure is incrementally performed using the dynamic techniques described in the next sections.

Theorem 3 *Let G be a triconnected graph with n vertices and m edges. There exists an $O(n)$-space data structure that supports query operation* FourPaths(v', v'') *in $O(1)$ time and can be constructed in $O(n \cdot \alpha(m, n) + m)$ time.*

Our data structure can be easily modified so that, if the answer to *FourPaths*(v', v'') is "no", it also reports a triplet that separates v' from v''. This is accomplished simply by labeling the nodes of $\mathcal{T}$ with integers such that the label of a node is larger than the label of its parent. Let μ' and μ'' be the proper allocation nodes of v' and v'', respectively. If *FourPaths*(v', v'') returns "no", then we report the poles of the node among μ' and μ'' with the larger label. For example, in Fig. 1, operation *FourPaths*(v_3, v_9) returns "no" and the triplet $\{v_{11}, e_5, e_6\}$.

It is interesting to observe that, although the number of separating triplets of G is $O(n^2)$ [17], only $O(n)$ such triplets are stored in the data structure, namely the "canonical" triplets consisting of the poles of the nodes of $\mathcal{T}$. If v' and v'' are separated by a triplet, then they are also separated by a canonical triplet. The canonical triplets are nonredundant and do not have nonredundant cross separating triplets. The quadratic bound on the number of triplets is caused by the wheels and their cross-separating triplets [17].

For a general graph G, we consider the K-connectivity query operation *K-Paths*(v', v'') that asks whether there exist K disjoint paths between vertices v' and v''. Combining the data structure of Theorem 3 with the ones for 2- and 3-connected components of [30] and [4], we obtain the following result.

Corollary 1 *Let G be a graph with n vertices and m edges. There exists an $O(n)$-space data structure that supports a query operation* K-Paths(v', v''), *for $K \leq 4$, in $O(1)$ time and can be constructed in $O(n \cdot \alpha(m, n) + m)$ time.*

4 Incremental Construction of a Triconnected Graph

We consider a dynamic environment where a triconnected graph G is incrementally constructed by inserting vertices and edges using operations of the following repertory.

I-insertion: add an edge e between existing vertices u' and u''.

T-insertion: insert a vertex v along an existing edge a, which becomes split into two edges a' and a'', and add an edge e between the newly created vertex v and an existing vertex u.

H-insertion: insert vertices v' and v'' along existing edges a' and a'', and add an edge e between the newly created vertices v' and v''.

Is is easy to see that after each of the above operations the graph remains triconnected. As shown in the following theorem, the above repertory of update operations is complete. We denote with K_4 the complete graph on four vertices.

Theorem 4 *Let G be a triconnected graph with n vertices and m edges. Graph G can be assembled from the*

complete graph K_4 using a sequence of $O(n+m)$ operations from the above repertory.

In order to prove Theorem 4 we need to introduce further definitions. An *ear decomposition* $D = (P_0, P_1, \cdots, P_r)$ of graph G is a partition of the edges of G into an ordered collection of edge-disjoint simple paths P_0, P_1, $\cdots$, P_r such that P_0 is a simple cycle, each endpoint of P_i, for $i \geq 1$, is contained in some P_j, $j < i$, and none of the internal vertices of P_i are contained in any P_j, $j < i$. The paths in D are called *ears*. D is an *open ear decomposition* if, for $i \geq 1$, ear P_i is not a cycle. A graph has an open ear decomposition if and only if it is biconnected [31]. Moreover, all intermediate graphs $D_i = P_0 + P_1 + \cdots + P_i$ of an open ear decomposition of a biconnected graph are biconnected [31]. Ear decomposition has been recently used in designing several efficient algorithms for graph connectivity and planarity [2, 8, 19, 20, 21, 23, 25, 26].

We say that an *ear decomposition* $D = (P_0, P_1, \cdots, P_r)$ is *triconnected* if the subgraph $D_2 = P_0 + P_1 + P_2$ is homeomorphic to the complete graph K_4, and for $i \geq 3$, the subgraph $D_i = P_0 + P_1 + \cdots + P_i$ is homeomorphic to a triconnected graph. The following theorem is analogous to a structural characterization of Chen and Gross [1].

Theorem 5 *A graph with no degree-2 vertices is triconnected if and only if it has a triconnected ear decomposition.*

Sketch of Proof: The if-part is immediate. For the only-if part, inductively suppose that we have constructed the subgraph $D_i = P_0 + \cdots + P_i$, for $i \geq 1$. Since G is triconnected, we can always find an ear P_{i+1} whose endpoints are in D_i and are not both in a maximal path of D_i which is an induced subgraph of D_i. Note that with such a P_2, the subgraph $D_1 + P_2$ is homeomorphic to the complete graph K_4. By Menger's theorem [22], if D_i is homeomorphic to a triconnected graph, then the subgraph $D_{i+1} = D_i + P_{i+1}$ is also homeomorphic to a triconnected graph. Therefore, we obtain a triconnected ear decomposition of G. On the other hand, if at some stage during the above process, the next ear P_{i+1} satisfying the above condition cannot be found, then we conclude that G is not triconnected. □

Theorem 4 follows from Theorem 5.

5 Dynamic Data Structure

In this section we show how to perform T-insertions. First, we analyze the structural changes in the decomposition tree caused by this operation, and then give the dynamic data structure. The other two update operations, H-insertions and I-insertions, are performed using similar algorithms. A detailed description is not provided in this extended abstract and special cases are not discussed.

The algorithm for T-inserting a new edge e between a vertex u and an edge a starts by finding nodes $\mu' = \mathit{proper}(u)$, $\mu'' = \mathit{proper}(a)$, and their least-common ancestor μ. Next, it performs a sequence of *transformations* along the paths from μ' to μ and from μ'' to μ. Informally, the effect of these transformations is to *merge* all R-nodes and to *split* all the W-nodes encountered on these paths. When we merge two R-nodes we also merge their skeletons and sets of children. When we split a wheel, we separate it skeleton and children set. Also, a new T-node is created for the new degree-3 vertex created by the operation. We show in Fig. 5 an example of T-insertion.

The number of transformation can be $O(n)$ in the worst case. Let R, F, and W denote the sets of R-, F-, and W-nodes of $\mathcal{T}$. Also, let edges(μ) be number of edges that are poles of node μ. We define the *potential* Φ of G as

$$\Phi = |R| + 2\sum_{\mu \in W} \deg(\mu) + 3\sum_{\nu \in F} \deg(\nu) + 2\sum_{\xi \in \mathcal{T}} \text{edges}(\xi)$$

By Lemma 1, we have that $\Phi = O(n)$. Let N be the number of transformations performed in a T-insertion, and let $\Delta\Phi$ be the corresponding variation of potential in $\mathcal{T}$. We show that $N + \Delta\Phi = O(1)$, so that the amortized number of transformations is $O(1)$.

We represent tree $\mathcal{T}$ using a union-find structure [29] for the children of an R-node μ and the vertices properly allocated at μ. We use a split-find structure [9, 15] for the children of a W-node ν, the vertices and edges properly allocated at ν, and the wheel of ν. No special structure is needed for T-nodes and F-nodes: we store their entire skeleton. Note that we do not store the edges properly allocated at R-nodes, so that the space requirement is $O(n)$ and does not depend on the number of edges. With these data structures a merge operation takes $O(\alpha(\ell, n))$ amortized time, while a split operation takes $O(1)$ amortized time. Each transformation takes $O(1)$ time plus the time to perform a merge or a split operation.

Theorem 6 *Let G be a triconnected graph. There exists an $O(n)$-space dynamic data structure for G that supports query operation* FourPaths *and incremental updates (I-, T- and H-insertion) in time $O(\alpha(\ell, n))$ amortized, where n is the current number of vertices of G, and ℓ is the number of operations performed.*

For general graphs we extend the repertory of update operations by allowing the insertion of an isolated vertex and the insertion of a vertex along an existing edge. We use a combination of our data structure and existing techniques for maintaining triconnected, biconnected, and connected components [4, 30, 29].

Theorem 7 *Let G be a graph that is updated on-line by inserting vertices and edges, and let n be the current number of vertices in G. There exists an $O(n)$-space data structure that supports query operation* K-Paths(v', v''), *for $K \leq 4$, and insertion of vertices and edges such that a sequence of ℓ intermixed query and update operations takes total time $O(n \log n + \ell)$.*

6 Testing Four-Connectivity

In this section we present an algorithm that tests if a triconnected graph G is four-connected in time $O(n \cdot \alpha(n,n)+m)$ using $O(n+m)$ space, where n and m denote the number of vertices and edges of G, respectively. It is based on Theorem 4, our technique for maintaining four-connected components, and the sparse certificates for connectivity [3, 24].

The algorithm consists of the following steps:

1. Find a sparse certificate S for the four-connectivity of G: S is a spanning subgraph of G with $O(n)$ edges such that G is four-connected if and only if S is four-connected.
2. Find a triconnected ear decomposition $D = (P_0, P_1, \cdots, P_r)$ of S. Note that $r = O(n)$.
3. Starting from K_4 (given by $P_0 + P_1 + P_2$), incrementally construct S by means of $r-2$ update operations of our repertory, where the i-th update corresponds to adding ear P_{i+2}.
4. If the decomposition tree of S consists of a single R-node, then report that graph G is four-connected, and otherwise report that it is not.

Step 1 can be performed in time $O(n+m)$ [3, 24]. Step 3 consists of performing a sequence of $O(n)$ update operations, each an I-, T-, or H-insertion, and hence by Theorem 6 takes time $O(n \cdot \alpha(n,n))$. Step 4 trivially takes $O(1)$ time.

The algorithm for Step 2, based on Theorems 4 and 5, is a variation of the depth-first-search algorithm of [14] for testing triconnectivity. Informally speaking, we construct the triconnected ear decomposition based on the ordering of the depth-first-search tree. Each ear consists of a (possibly empty) sequence of tree edges followed by a back edge. Therefore, we check for every back edge if the corresponding ear adjoins two different edges of the current graph. If it does, then we add it immediately to the current graph. Otherwise, we delay the addition of the ear until it becomes incident upon two different edges. An informal sketch of the algorithm is given below.

1. Construct a depth-first-search tree T for S.
2. Find back edges e_1, e_2, and e_3 such that the graph $T+e_1+e_2+e_3$ contains a subgraph homeomorphic to K_4, which gives the first three ears.
3. Perform the following variation of depth-first search. For each vertex v encountered during the search, check all back edges ending at v. For each back edge $\{u, v\}$, trace backwards from vertex u along the tree until a vertex w is encountered that is already on the current graph. Check if both v and w are on a chain of the current graph that is an induced subgraph. If so, then we delay the ear placing it on a stack and keep a record of its starting and ending vertices. Otherwise, we add the ear to the current graph and add other ears that were previously delayed.

Since each edge of S is contained in exactly one ear, and each ear can be delayed and added at most once, the above algorithm runs in $O(n)$ time.

Theorem 8 *Let G be a graph with n vertices and m edges. There exists an algorithm for testing whether G is four-connected that runs in time $O(n \cdot \alpha(n,n)+m)$ and uses $O(n+m)$ space.*

Acknowledgements

The first author is in debt to R. Thurimella for suggesting the use of sparse certificates. The authors would also like to thank Vijaya Ramachandran and Ioannis Tollis for useful discussions and comments on early versions of this paper.

References

[1] J. Chen and J.L. Gross. Extensions of Whitney's theorem synthesizing 2-connected graphs. Submitted for publication, 1990.

[2] J. Cheriyan and S.N. Maheshwari. Finding nonseparating induced cycles and independent spanning trees in 3-connected graphs. *Journal of Algorithms*, 9:507–537, 1988.

[3] J. Cheriyan and R. Thurimella. Algorithms for parallel k-vertex connectivity and sparse certificates. *Proc. 23th ACM Symp. on Theory of Computing*, pages 391–401, 1991.

[4] G. Di Battista and R. Tamassia. On-line graph algorithms with SPQR-trees. *Automata, Languages and*

Programming (Proc. 17th ICALP), LNCS 442:598–611, 1990.

[5] S. Even and Y. Shiloach. An on-line edge deletion problem. *J. ACM*, 28:1–4, 1981.

[6] G.N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Computing*, 14(4):781–798, 1985.

[7] G.N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *Proc. 32th IEEE Symp. on Foundations of Computer Science*, 1991.

[8] D. Fussell, V. Ramachandran, and R. Thurimella. Finding triconnected components by local replacements. *Automata, Languages and Programming (Proc. 16th ICALP)*, LNCS 372:379–393, 1989.

[9] H.N. Gabow and R.E. Tarjan. A linear time algorithm for a special case of disjoint set union. *J. Computer Systems Sciences*, 30, 1985.

[10] Z. Galil. Finding the vertex connectivity of graphs. *SIAM J. Computing*, 9:197–199, 1980.

[11] Z. Galil and G.F. Italiano. Fully dynamic algorithms for edge-connectivity problems. *Proc. 23th ACM Symp. on Theory of Computing*, pages 317–327, 1991.

[12] Z. Galil and G.F. Italiano. Maintaining biconnected components of dynamic planar graphs. *Automata, Languages and Programming (Proc. 18th ICALP)*, 1991.

[13] M. Girkar and M. Sohoni. On finding the vertex connectivity of graphs. Technical Report ACT-77, Coordinated Science Laboratory, Univ. of Illinois at Urbana-Champaign, 1987.

[14] J. Hopcroft and R.E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Computing*, 2(3):135–158, 1973.

[15] H. Imai and T. Asano. Dynamic orthogonal segment intersection search. *J. Algorithms*, 8:1–18, 1987.

[16] A. Kanevsky. *Vertex Connectivity of Graphs: Algorithm and Bounds.* PhD thesis, University of Illinois at Urbana-Champaign, Coordinated Science Laboratory, 1988. Technical Report ACT-97.

[17] A. Kanevsky. A characterization of separating pairs and triplets in a graph. *Congressus Numerantium*, 74:213–232, 1990.

[18] A. Kanevsky. On the number of minimum size separating vertex sets in a graph and how to find all of them. *Proc. 1st Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 411–421, 1990.

[19] A. Kanevsky and V. Ramachandran. Improved algorithms for graph four-connectivity. *Journal of Computer Science and Systems*, 42:288–306, 1991.

[20] L. Lovasz. Computing ears and branchings in parallel. *Proc. 26th IEEE Symp. on Foundations of Computer Science*, pages 464–467, 1985.

[21] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (eds) and st-numbering in graphs. *Theoretical Computer Science*, 47:277–298, 1986.

[22] K. Menger. Zur allgemeinen kurventheorie. *Fund. Math.*, 10:96–115, 1927.

[23] G.L. Miller and V. Ramachandran. A new graph triconnectivity algorithm and its parallelization. *Proc. 19th ACM Symp. on Theory of Computing*, pages 335–344, 1987.

[24] H. Nagamochi and T. Ibaraki. Linear time algorithms for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 1991, to appear.

[25] V. Ramachandran and J.H. Reif. An optimal parallel algorithm for graph planarity. *Proc. IEEE Symp. on Foundations of Computer Science*, pages 282–293, 1989.

[26] V. Ramachandran and U. Vishkin. Efficient parallel triconnectivity in logarithmic time. *VLSI Algorithms and Architectures (Proc. 3rd AWOC)*, LNCS 319:33–42, 1988.

[27] J.H. Reif. A topological approach to dynamic graph connectivity. *Information Processing Letters*, 25:65–70, 1987.

[28] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1:146–160, 1972.

[29] R.E. Tarjan and J. van Leeuwen. Worst-case analysis of set-union algorithms. *J. ACM*, 31:245–281, 1984.

[30] J. Westbrook and R.E. Tarjan. Maintaining bridge-connected and biconnected components on-line. Technical Report CS-TR-228-89, Dept. Computer Science, Princeton Univ., 1989. To appear in *Algorithmica*.

[31] H. Whitney. Non-separable and planar graphs. *Trans. Amer. Math. Soc.*, 34:339–362, 1932.

Figure 1: (a) A triconnected graph G. (b) Decomposition tree $\mathcal{T}$ of G with respect to triplet $\{t_1, t_2, t_3\}$. The poles of a node μ are shown next to the edge from μ to its parent. The vertices and edges of the skeleton of μ are shown next to μ and its incident edges. (c) The wheel of the W-node of tree $\mathcal{T}$.

Figures 2, 3, 4: (a) A triconnected graph G. (b) Decomposition tree $\mathcal{T}$ of G with respect to triplet $\{v_1, v_2, v_3\}$.

Figure 5: Example of T-insertion of an edge. (a) Graph with the newly inserted edge drawn with a thick dashed line. (b-c) Decomposition trees before and after the insertion. The proper allocation nodes of the vertex and edge joined by the new edge, and the path between them, are shown with thick lines.

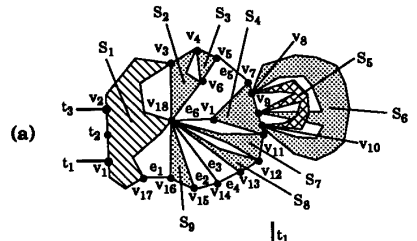
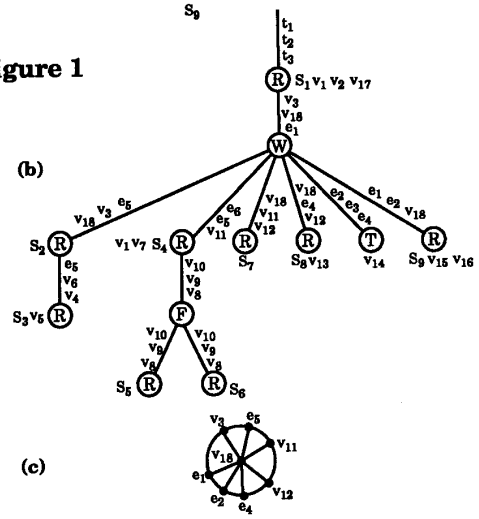


Figure 1



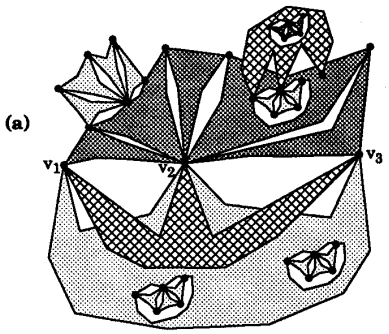
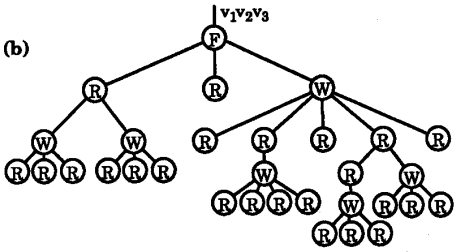


Figure 2



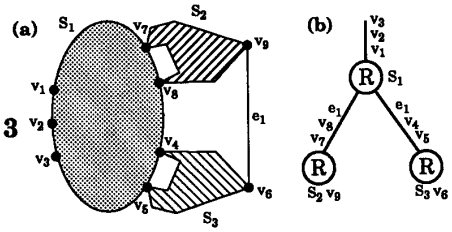


Figure 3

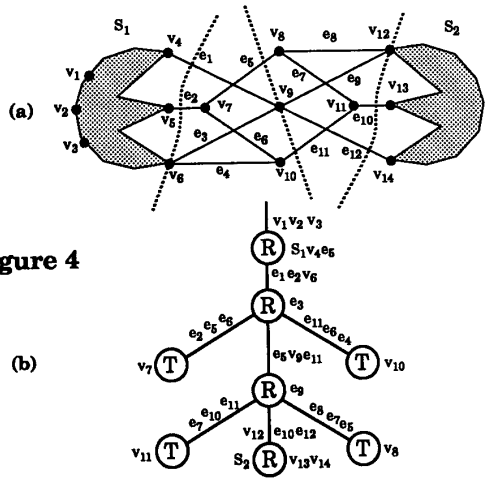
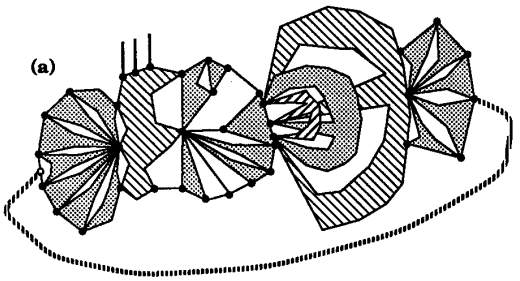


Figure 4



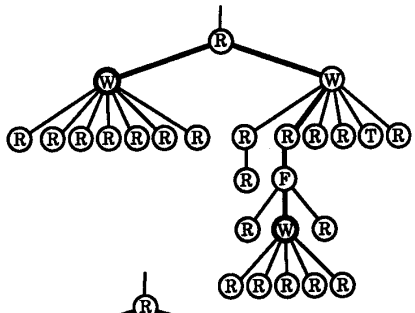


Figure 5