Figure 5

Figure 4

Figure 3

Figure 2

Figure 1

[32] R. Tamassia, and F. P. Preparata, "Dynamic maintenance of planar digraphs, with applications", *Algorithmica* 5 (1990), 509–527.

[33] R. Tamassia, and I. G. Tollis, "Reachability in planar digraphs", manuscript 1990.

[34] R. E. Tarjan, "Depth-first search and linear graph algorithms", *SIAM J. Comput.* 1 (1972), 146–160.

[35] R. E. Tarjan, and J. van Leeuwen, "Worst case analysis of set union algorithms", *J. Assoc. Comput. Mach.* 31 (1984), 245–281.

[36] R. E. Tarjan, and U. Vishkin, "An efficient parallel biconnectivity algorithm", *SIAM J. Comput.* 14 (1985), 862–864.

[37] J. Westbrook, "Algorithms and data structures for dynamic graph problems", Ph. D. Dissertation, Tech. Rep. CS-TR-229-89, Department of Computer Science, Princeton University, October 1989.

[38] J. Westbrook, and R. E. Tarjan, "Maintaining bridge-connected and biconnected components on-line", Tech. Rep. CS-TR-228-89, Department of Computer Science, Princeton University, August 1989. To appear in *Algorithmica*.

[39] D. M. Yellin, "A dynamic transitive closure algorithm", Research Report 13535, IBM Research Division, T. J. Watson Research Center, 1988.

[16] F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[17] D. Harel, "On-line maintenance of the connected components of dynamic graphs", Unpublished manuscript, 1982.

[18] J. Hopcroft, and R. E. Tarjan, "Dividing a graph into triconnected components", *SIAM J. Comput.* 2 (1973), 135–158.

[19] N. Horspool, "Incremental generation of LR parsers", Technical Report, Department of Computer Science, University of Victoria, 1988.

[20] T. Ibaraki, and N. Katoh, "On-line computation of transitive closure for graphs", *Inform. Process. Lett.* 16 (1983), 95–97.

[21] G. F. Italiano, "Amortized efficiency of a path retrieval data structure", *Theoret. Comput. Sci.* 48 (1986), 273–281.

[22] G. F. Italiano, "Finding paths and deleting edges in directed acyclic graphs", *Inform. Process. Lett.* 28 (1988), 5–11.

[23] G. F. Italiano, A. Marchetti Spaccamela, U. Nanni, "Dynamic data structures for series parallel digraphs", *Proc. Workshop on Algorithms and Data Structures* (WADS 89), *Lecture Notes in Computer Science* vol. 382, Springer-Verlag, Berlin, 1989, 352–372.

[24] J. A. La Poutré, and J. van Leeuwen, "Maintenance of transitive closure and transitive reduction of graphs", *Proc. Workshop on Graph-Theoretic Concepts in Computer Science*, *Lecture Notes in Computer Science*, vol. 314, Springer-Verlag, Berlin, 1988, 106–120.

[25] C. C. Lin, and R. C. Chang, "On the dynamic shortest path problem", *Proc. Int. Workshop on Discrete Algorithms and Complexity*, 1989, 203–212.

[26] J. H. Reif, "A topological approach to dynamic graph connectivity", *Inform. Process. Lett.* 25 (1987), 65–70.

[27] H. Rohnert, "A dynamization of the all pairs least cost path problem", *Proc. 2nd Annual Symp. on Theoretical Aspects of Computer Science*, *Lecture Notes in Computer Science*, vol. 182, Springer-Verlag, Berlin, 1985, 279–286.

[28] D. D. Sleator, and R. E. Tarjan, "A data structure for dynamic trees", *J. Comput. System Sci.* 24 (1983), 362–381.

[29] D. D. Sleator, and R. E. Tarjan, "Self-adjusting binary search trees", *J. Assoc. Comput. Mach.* 32 (1985), 652–686.

[30] P. M. Spira, and A. Pan, "On finding and updating spanning trees and shortest paths", *SIAM J. Comput.* 4 (1975), 375–380.

[31] R. Tamassia, "A dynamic data structure for planar graph embedding", *Proc. 15th Int. Colloquium on Automata, Languages and Programming*, *Lecture Notes in Computer Science*, vol. 317, Springer-Verlag, Berlin, 1988, 576–590.

References

[1] G. Ausiello, G. F. Italiano, A. Marchetti Spaccamela, U. Nanni, "Incremental algorithms for minimal length paths", *Proc. 1st Annual ACM-SIAM Symp. on Discrete Algorithms*, 1990, 12–21. A journal version to appear in *J. Algorithms*.

[2] G. Ausiello, A. Marchetti Spaccamela, and U. Nanni, "Dynamic maintenance of paths and path expressions in graphs", *Proc. Int. Symp. on Symbolic and Algebraic Computation*, *Lecture Notes in Computer Science* 358, Springer-Verlag, Berlin, 1989, 1–12.

[3] F. Chin, and D. Houk, "Algorithms for updating minimum spanning trees", *J. Comput. System. Sci.* 16 (1978), 333–344.

[4] R. F. Cohen, and R. Tamassia, "Dynamic expression trees and their applications", *Proc. 2nd Annual ACM-SIAM Symp. on Discrete Algorithms*, 1991, 52–61.

[5] G. Di Battista, and R. Tamassia, "Incremental planarity testing", *Proc. 30th Annual Symp. on Foundations of Computer Science*, 1989, 436–441.

[6] G. Di Battista, and R. Tamassia, "On-line graph algorithms with SPQR-trees", *Proc. 17th Int. Colloquium on Automata, Languages and Programming*, *Lecture Notes in Computer Science*, vol. 443, Springer-Verlag, Berlin, 1990, 598–611.

[7] 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 Annual ACM-SIAM Symp. on Discrete Algorithms*, 1990, 1–11.

[8] S. Even, and H. Gazit, "Updating distances in dynamic graphs", *Methods of Operations Research* 49 (1985), 371–387.

[9] S. Even, and Y. Shiloach, "An on-line edge deletion problem", *J. Assoc. Comput. Mach.* 28 (1981), 1–4.

[10] G. N. Frederickson, "Data structures for on-line updating of minimum spanning trees", *SIAM J. Comput.* 14 (1985), 781–798.

[11] H. Gabow, and R. E. Tarjan, "A linear-time algorithm for a special case of disjoint set union", *J. Comput. Syst. Sci.* 30 (1985), 209–221.

[12] Z. Galil, and G. F. Italiano, "Reducing edge connectivity to vertex connectivity", *SIGACT News*, 1991.

[13] Z. Galil, and G. F. Italiano, "Fully dynamic algorithms for edge connectivity problems", *Proc. 23rd Annual ACM Symp. on Theory of Computing*, 1991.

[14] Z. Galil, and G. F. Italiano, "Maintaining biconnected components of dynamic planar graphs", *Proc. 18th Int. Colloquium on Automata, Languages and Programming*, 1991.

[15] Z. Galil, and G. F. Italiano, "Fully dynamic algorithms for 3-edge connectivity", in preparation.

InsertEdge(x, y, A): Insert a new edge between vertices x and y and call A the new (if any) 3-edge-connected component created by (x, y).

We have been able to achieve an $O(m^{2/3})$ bound per operation for the fully dynamic maintenance of the 3-edge-connected components of a graph [15] by using techniques similar to the ones used for the fully dynamic maintenance of the 2-edge-connected components [13].

One might ask whether there is any efficient partially dynamic algorithm for maintaining on-line the k-edge-connected components of a graph, $k \geq 4$.

Acknowledgments

We are grateful to Dany Breslauer for many useful discussions.

of vertices in B_i. Since each vertex of G is an at most one 2-edge-connected component, we have that $\sum_i n_i = n + 1$. By Theorem 3.1, the tree of 3-edge-connected components of B_i has at most $O(n_i)$ nodes and edges. As a result, all the trees in our data structure require

$$O(\sum_i n_i) = O(n)$$

space.

Each *Same3EdgeBlock* operation can be performed in $O(\alpha(q,n))$ amortized time as the following argument shows. Finding B_u and B_v, the 2-edge-connected components of G containing u and v, can be done in $O(\alpha(q,n))$ amortized time [38]. If $B_u \neq B_v$ then *Same3EdgeBlock* returns false and no further computation is required. If $B_u = B_v = B$, performing *Same3EdgeBlock* on $\mathcal{T}_2(B)$ requires $O(\alpha(q,n))$ amortized time by Theorem 3.1.

As for an *InsertEdge*(u,v) operation, the bridge-block tree can be updated in $O(\alpha(q,n))$ amortized time [38]. The update of the trees of 3-edge-connected components $\mathcal{T}_2(B_i)$, $0 \leq i \leq \ell$, requires

$$O(\ell + \sum_i |\pi_i|)$$

merge, *split* and *parent* primitives. By applying exactly the same argument used in Theorem 3.1, we obtain that the total number of such primitives during any sequence of *InsertEdge* operations is $O(n)$. As a result, the total time required to perform any sequence of *InsertEdge* operations is $O((q+n)\alpha(q,n))$. Finally, each *AddVertex* operation can be performed in $O(1)$ time. □

Our data structure can be used even if we start with a non-empty graph $G_0 = (V_0, E_0)$ and we allow $O(|V_0| + |E_0|)$ preprocessing.

Lemma 4.2 *Given a graph $G_0 = (V_0, E_0)$ our data structure can be initialized in $O(|V_0| + |E_0|)$ time.*

Proof: Compute in $O(|V_0| + |E_0|)$ the 2-edge-connected components of G_0 by using the algorithm of Tarjan [34]. This gives also $\mathcal{T}_1$, the bridge-block tree of G_0, which can be initialized in $O(|\mathcal{T}_1|) = O(|V_0|)$ as a condensible and expandable nodes tree. For each 2-edge-connected component B of G_0, compute the 3-edge-connected components of B and initialize the tree of 3-edge-connected components $\mathcal{T}_2(B)$. This requires $O(|B|)$ time [12], and sums up to a total time of $O(|V_0| + |E_0|)$. □

5 Concluding Remarks

In this paper we have studied the on-line maintenance of the 3-edge-connected components of an undirected graph during edge and vertex insertions. We have proposed algorithms that support any sequence of q *InsertEdge*, *AddVertex*, and *Same3EdgeBlock* operations on an initially connected graph with n vertices in a total of $O((n+q)\alpha(q,n))$ time.

We remark that the same $O(\alpha(q,n))$ amortized bound holds also if we wish to maintain the names of the 3-edge-connected components of a graph subject to the following operations.

3EdgeBlock(x): Return the name of the 3-edge-connected component containing vertex x.

the proof of Lemma 3.3, it can be shown that all and only the vertices of $graph(B_i)$ contained in $V'_{i,1}, V'_{i,2}, \ldots, V'_{i,q}$ become 3-edge-connected. □

Let $G^{new} = G \cup \{(u,v)\}$, and let $(G^{new})'$ be the graph obtained by shrinking the 3-edge-connected components of G^{new} into one vertex. Once again, Lemma 4.1 gives a way to compute $(G^{new})'$ from G'. Consider the 2-edge-connected components of G' obtained by shrinking the 3-edge-connected components of $graph(B_i)$, $0 \leq i \leq \ell$. Let us refer to them as B'_i, $0 \leq i \leq \ell$. For each one of them we can repeat the same argument given after Lemma 3.3. Fix i, $0 \leq i \leq \ell$. By definition of G' all the paths between $V'_{i,1}$ and $V'_{i,q}$ in G' are given by $q-1$ edge-disjoint simple cycles $\rho_{i,1}, \rho_{i,2}, \ldots, \rho_{i,q-1}$ in B'_i such that $V'_{i,1} \in \rho_{i,1}$, $V'_{i,q} \in \rho_{i,q-1}$, and $\rho_{i,j-1} \cap \rho_{i,j} = \{V'_{i,j}\}$, $2 \leq j \leq q-1$. Each $\rho_{i,j}$ is composed of two disjoint paths between $V'_{i,j}$ and $V'_{i,j+1}$, say $\rho^{(1)}_{i,j}$ and $\rho^{(2)}_{i,j}$. Furthermore, the insertion of (u,v) in G implies that all and only the vertices in $V'_{i,s}$, $1 \leq s \leq p$, become 3-edge-connected in G. Therefore in $(G^{new})'$ the vertices $V'_{i,1}, V'_{i,2}, \ldots, V'_{i,q}$ of G' have to be shrunk into one vertex V'_i. This destroys the previous simple cycles $\rho_{i,j}$, $1 \leq j \leq q-1$, and creates at most $2(q-1)$ new simple cycles $\rho^{(1)}_{i,j}$ and $\rho^{(2)}_{i,j}$, $1 \leq j \leq q-1$, that are all incident to V'_i. Furthermore, there is a new simple cycle in G' consisting of all the newly created vertices $V'_1, V'_2, \ldots, V'_\ell$ and edges $(x_0,y_0), (x_1,y_1), \ldots, (x_{\ell-1}, y_{\ell-1}), (u,v)$.

Because of Lemma 4.1 we have to perform the following updates in our data structure. First, all the nodes B_i, $0 \leq i \leq \ell$, have to be merged into one node B in $\mathcal{T}_1$ because all the vertices in $graph(B_0), graph(B_1), \ldots, graph(B_\ell)$ become now 2-edge-connected. This can be done by using the algorithm of Westbrook and Tarjan [38]. Second, we have to compute the tree $\mathcal{T}_2(B)$ of 3-edge-connected components of the new node $B = B_0 \cup B_1 \cup \ldots \cup B_\ell$. We compute $\mathcal{T}_2(B)$ starting from the trees $\mathcal{T}_2(B_0)$, $\mathcal{T}_2(B_1)$, ..., $\mathcal{T}_2(B_\ell)$ as follows. For $0 \leq i \leq \ell$, we perform $CompressPath(start(B_i), end(B_i), \mathcal{T}_2(B_i))$. This is correct because of condition (ii) of Lemma 4.1, and produces the new trees $\mathcal{T}_2^{new}(B_i)$, $0 \leq i \leq \ell$. Denote by σ_i the square node of $\mathcal{T}_2^{new}(B_i)$ which is the result of the path compression that took place in $\mathcal{T}_2(B_i)$. Notice that σ_i, $0 \leq i \leq \ell$, corresponds to the new vertex V'_i in $(G^{new})'$. We recall that B_λ is the least common ancestor of B_u and B_v in $\mathcal{T}_1$. As said before, all the new vertices V'_i are now in a new simple cycle of $(G^{new})'$: we create a new round node ρ, make ρ a child of σ_λ and make $\sigma_{\lambda+1}, \ldots, \sigma_\ell, \sigma_0, \ldots, \sigma_{\lambda-1}$ children of ρ in $\mathcal{T}_2(B)$ in this order. This preserves the same order as in the new simple cycle created in $(G^{new})'$. Since σ_i, $i \neq \lambda$, is the root of $\mathcal{T}_2^{new}(B_i)$, we do not need to re-root any of these trees.

An $AddVertex(u,v)$ operation inserts a new vertex u and connects it to v. We update our data structure as follows. Let B_v be the 2-edge-connected component containing v. We create a new 2-edge-connected component B_u containing only vertex u and make it child of B_v in $\mathcal{T}_1$. We initialize $\mathcal{T}_2(B_u)$ to be a tree with a singleton square root node, and initialize a new 3-edge-connected component containing only vertex u.

Theorem 4.1 *There exists a data structure that supports any sequence of* q Same3EdgeBlock *and* InsertEdge *and* n AddVertex *operations in a total of* $O((q+n)\alpha(q,n))$ *time. The space required is* $O(n)$.

Proof: We first analyze the space complexity of our data structure. Notice that we start with a graph containing a single vertex. At the end of the sequence of operations, we end up with a graph G with at most $n+1$ vertices. Since G can have at most $O(n)$ 2-edge-connected components and bridges, $\mathcal{T}_1$ has size $O(n)$. For each 2-edge-connected component B_i of G, denote by n_i the number

represented by a simple edge (u,v) in $\mathcal{T}$, while there is a square node σ and edges (u,σ) and (σ,v) in the block tree of G'. Finally, we remark that Lemma 3.1 still holds for G'. As far as Lemma 3.2 is concerned, we have now that each biconnected component of G' consists of either a unique edge or a simple cycle.

We show how to perform the three operations. A *Same3EdgeBlock*(u,v) is carried out as follows. Find the 2-edge-connected components of G containing u and v, say B_u and B_v by using the algorithm of Westbrook and Tarjan [38]. If $B_u \neq B_v$ then u and v are not even in a same 2-edge-connected component of G, and therefore they cannot be in the same 3-edge-connected component of G and we return *false*. Otherwise, u and v are in a same 2-edge-connected component $B = B_u = B_v$ of G. We now perform *Same3EdgeBlock*(u,v) on a 2-edge-connected graph as explained in the previous section.

To support an *InsertEdge*(u,v), we first find the 2-edge-connected components of G containing vertices u and v, say B_u and B_v. If $B_u = B_v = B$, then we perform *CompressPath*$(u,v,\mathcal{T}_2(B))$ as shown in the previous section.

If $B_u \neq B_v$, let $B_u = B_0, B_1, \ldots, B_\ell = B_v$ be the nodes in the path $\pi_{u,v}$ in $\mathcal{T}_1$ between B_u and B_v as they are met while going from B_u to B_v. Each B_i corresponds to the 2-edge-connected component $graph(B_i)$ of G. For each B_i let (x_i, y_i) be the edge in G between $graph(B_i)$ and $graph(B_{i+1})$, $0 \leq i \leq \ell - 1$. For $1 \leq i \leq \ell - 1$ define $start(B_i) = y_{i-1}$ and $end(B_i) = x_i$. Notice that for each 2-edge-connected component $graph(B_i)$, $1 \leq i \leq \ell - 1$, $start(B_i)$ [$end(B_i)$] is the vertex to which the edge in $\pi_{u,v}$ between $graph(B_{i-1})$ and $graph(B_i)$ [between $graph(B_i)$ and $graph(B_{i+1})$] is incident to. For sake of completeness, define $start(B_0) = u$, $end(B_0) = x_0$, $start(B_\ell) = y_{\ell-1}$, and $end(B_\ell) = v$. Denote by B_λ the least common ancestor of B_u and B_v in $\mathcal{T}_1$ (see Figure 5).

[Figure 5]

The following lemma explains the updates needed in our data structure.

Lemma 4.1 *Let π_i be the path in $\mathcal{T}_2(B_i)$ between $C(start(B_i))$ and $C(end(B_i))$, $0 \leq i \leq \ell$. After the insertion of edge (u,v) in G, we have the following changes:*

(i) All and only the vertices in $\cup_{0 \leq i \leq \ell} \{graph(B_i)\}$ become 2-edge-connected in G.

(ii) Fix i, $0 \leq i \leq \ell$. Let $u_1^{(i)}$, $u_2^{(i)}$, ..., $u_{q_i}^{(i)}$ be the square nodes in π_i. All and only the vertices in the 3-edge-connected components of $graph(B_i)$ corresponding to $u_1^{(i)}$, $u_2^{(i)}$, ..., $u_{q_i}^{(i)}$ become 3-edge-connected in G.

Proof: Edge (u,v) introduces cycles in G that contain at least one vertex in $graph(B_0)$, $graph(B_1)$, ..., $graph(B_\ell)$. Since these are the only new cycles introduced, all and only the vertices in those 2-edge-connected components become 2-edge-connected as stated in proposition (i).

The proof of proposition (ii) can be carried out along the same lines as the proof of Lemma 3.3. Let $V_{i,1}, V_{i,2}, \ldots, V_{i,p}$ be the 3-edge-connected components of $graph(B_i)$ before the insertion of edge (u,v). Let $V'_{i,1}, V'_{i,2}, \ldots V'_{i,q}$, $1 \leq q \leq p$ be the 3-edge-connected of $graph(B_i)$ corresponding to the square nodes of $\mathcal{T}_2(B_i)$ in π_i, as they are met while going from $C(start(B_i))$ to $C(end(B_i))$ in $\mathcal{T}_2(B_i)$. Notice that $C(start(B_i)) = V'_{i,1}$ and $C(end(B_i)) = V'_{i,q}$. The insertion of edge (u,v) causes a new path outside $graph(B_i)$ between $V'_{i,1}$ and $V'_{i,q}$. By repeating the same argument given in

at the beginning G can have at most n 3-edge-connected components, and each time we perform *InsertEdge*(x, y) we decrease this number by at least $\left\lceil \frac{|\pi_{x,y}|}{2} \right\rceil$,

$$T(n) \leq \sum_{(x,y)} 3|\pi_{x,y}| = O(n)$$

.

As for the space complexity, we have to bound the size of the set-union data structures used to represent the 3-edge-connected components, and the size of the tree $\mathcal{T}$. Since each 3-edge-connected component of G consists of at least one vertex, and any two 3-edge-connected components are disjoint, the total space required by the set-union data structures used to store the 3-edge-connected components is $O(n)$. As for the size of the tree $\mathcal{T}$, we bound the number of square and round nodes in $\mathcal{T}$. Each square node of $\mathcal{T}$ uniquely corresponds to a vertex of G' (i.e., to a 3-edge-connected component of G), while each round node of $\mathcal{T}$ corresponds to a biconnected component of G'. Since we have at most $O(n)$ 3-edge-connected components in G, there can be at most $O(n)$ vertices and $O(n)$ biconnected components in G'. Therefore there can be at most $O(n)$ square and round nodes in $\mathcal{T}$. Since $\mathcal{T}$ has at most $O(n)$ nodes, and each *merge* primitive implies an $O(1)$ extra space, $\mathcal{T}$ can be implemented using condensible and expandable nodes in $O(n)$ space as shown in [38]. □

4 Maintaining 3-Edge-Connected Components of Connected Graphs

We now extend the previous result to the case of connected (but not necessarily 2-edge-connected) graphs. We start with a singleton vertex, and we would like to perform any intermixed sequence of *Same3EdgeBlock*, *InsertEdge*, and *AddVertex* operations.

In this case, we maintain the bridge-block tree $\mathcal{T}_1$ of G. We recall that each node B of $\mathcal{T}_1$ corresponds to a 2-edge-connected component of G, referred to as $graph(B)$, and each edge of $\mathcal{T}_1$ correspond to a bridge of G. Since $graph(B)$ is a 2-edge-connected component of G, it is 2-edge-connected. We maintain for each node B of $\mathcal{T}_1$ the tree $\mathcal{T}_2(B)$ of 3-edge-connected components of $graph(B)$ as defined in the previous section.

Thus, we maintain information about the graph G by using a tree of trees. Each of these trees is rooted as follows. $\mathcal{T}_1$ is rooted arbitrarily at any node C, and $\mathcal{T}_2(C)$ is rooted arbitrarily at any square node. For each non-root node B of $\mathcal{T}_1$, let (x_B, y_B) be the bridge in G corresponding to the edge $(B, parent(B))$ in $\mathcal{T}_1$, $x_B \in B$, $y_B \notin B$. We root $\mathcal{T}_2(B)$ at $C(x_B)$ (i.e., the square node corresponding to the 3-edge-connected component containing vertex x_B). Besides this data structure, we maintain also the 2-edge-connected components of G by using the algorithm of Westbrook and Tarjan [38].

As in the previous section, let G' be the graph obtained by shrinking each 3-edge-connected component of G into a super-vertex. Notice that this time G is not necessarily 2-edge-connected and therefore also G' is not necessarily 2-edge-connected. However, by Lemma 2.2 there is a one-to-one correspondence between bridges of G' and bridges of G, and between 2-edge-connected components of G' and 2-edge-connected components of G. Consequently, $\mathcal{T}_1$ is the bridge-block tree of G' as well, and given any 2-edge-connected component B of G', $\mathcal{T}_2(B)$ is the block tree of B. We notice that combining $\mathcal{T}_1$ and all the trees $\mathcal{T}_2(B)$ yields a tree $\mathcal{T}$ that resembles the block tree of G'. Indeed, denote by $\mathcal{T}$ the tree obtained by plugging each tree $\mathcal{T}_2(B)$ in place of the corresponding node in $\mathcal{T}_1$: the only difference between $\mathcal{T}$ and the block tree of G' is that each bridge (u, v) of G' is

perform the corresponding *split* operation in $\mathcal{SF}(\rho)$: this creates two new round nodes ρ_1 and ρ_2 and destroys ρ. To make ρ_1 and ρ_2 children of v we union $\mathcal{UF}(v)$ with two singleton sets. Finally we make v child of u by inserting an edge from u to the item of $\mathcal{UF}(u)$ that round node ρ was pointing to. Therefore splitting a round node ρ corresponds to performing a *split* operation in $\mathcal{SF}(\rho)$ and at most two union operations involving $\mathcal{UF}(v)$. Finding the parent of a square node corresponds to a *find* operation in a set-splitting data structure.

Notice that due to the structure of $\mathcal{T}$, in which after each update every path alternates between square and round nodes, there is no interaction between condensible and expandable nodes, since only round nodes are subject to splits, and only square nodes are subject to merges. Each *merge* requires unioning two condensible nodes, while each *split* can be performed by splitting an expandable node and by performing a constant number of unions on condensible nodes. Each *parent* requires a *find* operation in either a set-union or a set-splitting data structure. Therefore by using condensible and expandable nodes we are able to perform each sequence of p *merge*, *split* and *parent* primitives on a tree with n nodes in a total of $O(p\alpha(p,n))$ worst-case time. □

Lemma 3.4 allows us to prove the following theorem:

Theorem 3.1 *The data structure above supports any sequence of* q Same3EdgeBlock *and* Insert-Edge *operations on an initially 2-edge-connected graph with* n *vertices in a total of* $O((q+n)\alpha(q,n))$ *time. The space required is* $O(n)$.

Proof: A *Same3EdgeBlock* operation consists of performing two *find* operations in disjoint sets and therefore can be performed in $O(\alpha(q,n))$ amortized time. As for an *InsertEdge*(x,y) operation, it requires (i) finding the path $\pi_{x,y}$ in $\mathcal{T}$, (ii) performing the merging and splitting of square and round nodes in $\pi_{x,y}$, and (iii) unioning the disjoint sets corresponding to the square nodes (3-edge-connected components) in $\pi_{x,y}$. Given $\mathcal{T}$, we can locate the path $\pi_{x,y}$ by first performing *find*(x) and *find*(y). This returns $C(x)$ and $C(y)$, the two square nodes corresponding to the 3-edge-connected components containing x and y respectively. Then we trace the paths from $C(x)$ to the root of $\mathcal{T}$ and from $C(y)$ to the root of $\mathcal{T}$, alternating among them one edge at the time. We stop when we reach a node already visited. This requires $2|\pi_{x,y}|$ *parent* primitives in the worst case. Then we have to perform at most $|\pi_{x,y}|$ merges and splits along this path. We recall that *merge* operations union all the 3-edge-connected components in $\pi_{x,y}$. Henceforth, the whole operation can be implemented by using no more than $3|\pi_{x,y}|$ *merge*, *split*, and *parent* primitives. As a consequence of Lemma 3.4, it can be done in a total of $O(|\pi_{x,y}|\alpha(q,n))$ time.

The total time spent during a sequence of at most q *Same3EdgeBlock* and *InsertEdge* operations is therefore $O((q+T(n))\alpha(q,n))$, where

$$T(n) \leq \sum_{(x,y)} 3|\pi_{x,y}|$$

is the total number of *merge*, *split*, and *parent* primitives performed during *InsertEdge* operations.

We prove the claimed time bound by showing that $T(n) = O(n)$. Notice that the path $\pi_{x,y}$ in $\mathcal{T}$ is a path between square node $C(x)$ and square node $C(y)$. Since any path in $\mathcal{T}$ alternates between square and round nodes, $\pi_{x,y}$ contains at least $\left\lceil \frac{|\pi_{x,y}|}{2} \right\rceil + 1$ square nodes (i.e., 3-edge-connected components of G). After we perform path compression on $\pi_{x,y}$ we are left with only 1 square node. Consequently the number of 3-edge-connected components of G decreases by at least $\left\lceil \frac{|\pi_{x,y}|}{2} \right\rceil$. Since

If γ is a round node, then set $e = (\beta, \gamma)$ and perform *split*(γ, e). If γ is a square node, then perform *merge*(β). In both cases set $\gamma =$ *parent*(β).

Denote by β_1 the square node child of λ at which the preceding step stops. Now set $\beta = C(y)$ and repeat the same splits and merges in the path from $C(y)$ to λ. Similarly, denote by β_2 the square node child of λ in this path. If λ is a round node, then split it as shown in Figure 4 (a). Otherwise, λ is a square node; merge β_1, β_2 and λ as shown in Figure 4 (b). Notice that those two operations can be implemented by using a constant number of *merge* and *split* primitives. In the following, we refer to the update of $\mathcal{T}$ after an *InsertEdge*(u, v) operation as *CompressPath*$(u, v, \mathcal{T})$.

[Figure 4]

The following lemma shows how to perform efficiently any sequence of *merge*, *split* and *parent* operations.

Lemma 3.4 *Any sequence of* p merge, split *and* parent *primitives on a tree with* n *nodes requires a total of* $O(p\alpha(p, n))$ *worst-case time.*

Proof: To support efficiently these primitives, we use different data structures for each node of $\mathcal{T}$. We represent square nodes as *condensible nodes* [38], which enable one to merge two square nodes in $O(\alpha(p, n))$ amortized time at the price of spending $O(\alpha(p, n))$ amortized time to find the parent of a round node. The idea behind a condensible square node is to group all the sibling round nodes into disjoint sets and to maintain those disjoint sets under *union* and *find* operations. We do this as follows. By definition, the children of each square node in $\mathcal{T}$ are all round nodes. Let v be any square node in $\mathcal{T}$, with children $\rho_1, \rho_2, \ldots, \rho_\ell$, $\ell \geq 1$. We maintain a set $\mathcal{UF}(v)$ such that the name of $\mathcal{UF}(v)$ returns a pointer to v. Let ρ_i be a child of v: instead of having an edge from ρ_i to v, we have an edge from ρ_i to an item in the set $\mathcal{UF}(v)$.

We refer to [38] for the details of this method. It suffices to say here that a *merge*(v) operation corresponds to a *union* between disjoint sets $\mathcal{UF}(v)$ and $\mathcal{UF}($*parent*$(v))$, and to a *union* of the 3-edge-connected components corresponding to v and *parent*(v). We remark here that unioning disjoint sets $\mathcal{UF}(v)$ and $\mathcal{UF}($*parent*$(v))$ implies that the space used by the item corresponding to v in $\mathcal{UF}($*parent*$(v))$ is left unused. This implies an $O(1)$ extra space per *merge* operation.

Finding the parent of a round node corresponds to a *find* operation. If for each square node $v \in \mathcal{T}$ we implement $\mathcal{UF}(v)$ by means of a fast set-union data structure [35], each *merge* and *parent* primitive can be supported in $O(\alpha(p, n))$ amortized time.

For splitting round nodes efficiently we use a similar data structure, which we call *expandable node*, and which allows one to split a round node and to find the parent of a square node in constant amortized time. The expandable node data structure has much the same flavor as condensible nodes, but it is based upon set-splitting data structures. We recall here that set-splitting data structures are able to perform *find* and *split* operations on disjoint sets in constant amortized time [11]. Now we group all the sibling square nodes into disjoint sets and we maintain those disjoint sets under sequences of *split* and *find* operations. Let ρ be any round node in $\mathcal{T}$, with children $v_1, v_2, \ldots, v_\ell$, $\ell \geq 1$. We maintain a set $\mathcal{SF}(\rho)$ containing ℓ items and such that the name of $\mathcal{SF}(\rho)$ returns a pointer to ρ. Let v_i be any child of ρ: instead of having an edge from v_i to ρ, we have an edge from v_i to the corresponding item in the set $\mathcal{SF}(\rho)$. We implement *split*(ρ, e) as follows. Let v be the square node child of ρ such that $e = (v, \rho)$, and let u be the square node parent of ρ. We

adjacent to ρ_i. Without loss of generality let V'_{i+1} be the parent of ρ_i in $\pi_{x,y}$. Let $a_1^{(i)}, \ldots, a_s^{(i)}$, $V'_i, b_1^{(i)}, \ldots, b_t^{(i)}$, $s, t \geq 0$, be the children of ρ_i sorted from left to right. Split ρ_i into two round nodes $\rho_i^{(1)}$ and $\rho_i^{(2)}$: make $a_1^{(i)}, \ldots, a_s^{(i)}$ children of $\rho_i^{(1)}$ and $b_1^{(i)}, \ldots, b_t^{(i)}$ children of $\rho_i^{(2)}$. Make $\rho_i^{(1)}$ and $\rho_i^{(2)}$ children of V'. If $s = 0$ [$t = 0$] delete $\rho_i^{(1)}$ [$\rho_i^{(2)}$].

(iii) If λ is a round node, again denote by V'_i and V'_{i+1} the two square nodes in $\pi_{x,y}$ adjacent to λ. Without loss of generality let $a_1^{(i)}, \ldots, a_s^{(i)}$, V'_i, $b_1^{(i)}, \ldots, b_t^{(i)}$, V'_{i+1} $c_1^{(i)}, \ldots, c_u^{(i)}$, $s, t, u \geq 0$, be the children of λ sorted from left to right. Split λ into two round nodes $\lambda^{(1)}$ and $\lambda^{(2)}$: make $a_1^{(i)}, \ldots, a_s^{(i)}, V', c_1^{(i)}, \ldots, c_u^{(i)}$ children of $\lambda^{(1)}$ and $b_1^{(i)}, \ldots, b_t^{(i)}$ children of $\lambda^{(2)}$. Make $\lambda^{(1)}$ child of the previous parent of λ and $\lambda^{(2)}$ child of V'. If $t = 0$ delete $\lambda^{(2)}$. Notice that since $\mathcal{T}$ is rooted at a square node, λ cannot be the root of $\mathcal{T}$ and thus must have a parent.

The sequence of above transformations, plus the actual merging of the 3-edge-connected components, is referred to in the sequel as *path compression*. Figure 2 shows the updates needed in the graph of Figure 1 as a result of the insertion of a new edge.

[Figure 2]

To implement the above rules, we need to support the following primitives on the tree $\mathcal{T}$ during an *InsertEdge* operation:

split(ρ, e): Given a round node ρ and an edge $e = (x, \rho)$, where x is a square node child of ρ, split ρ into two round nodes ρ_1 and ρ_2: the edges previously to the left [right] of e are now incident to ρ_1 [ρ_2]. Make ρ_1 and ρ_2 children of x, and x child of the former parent of ρ (see Figure 3 a). If either ρ_1 or ρ_2 has no children, then delete it. Since ρ is a round node it cannot be the root and therefore the parent of ρ is always defined.

merge(v): Given a square node v, whose parent is a square node itself, contract the edge $(v, parent(v))$ (see Figure 3 b), and union the corresponding 3-edge-connected components.

parent(v): Given any (round or square) node v, return its parent.

Notice that the primitive *merge*(v) is not applicable to the tree $\mathcal{T}$ itself since in $\mathcal{T}$ each square node has a round parent. However, a *split*(ρ, e) produces two adjacent square nodes as shown in Figure 3 and therefore allows a primitive *merge* to be applied in the resulting tree.

[Figure 3]

We are now able to describe our implementation of an *InsertEdge*(x, y) operation. First we perform a *Same3EdgeBlock*(x, y) operation: if it returns true, then x and y are in the same 3-edge-connected component and by Lemma 3.3 nothing need to be done. Otherwise, x and y are in different 3-edge-connected components. We proceed as follows. Given $\mathcal{T}$, we first locate the path $\pi_{x,y}$. Then we perform path compression on $\pi_{x,y}$ by doing *merge* operations on the square nodes of $\pi_{x,y}$ and *split* operations on the round nodes of $\pi_{x,y}$. We do this as follows. Denote by λ the least common ancestor of $C(x)$ and $C(y)$ in $\mathcal{T}$. Set $\beta = C(x)$ and $\gamma = parent(\beta)$. While $\gamma \neq \lambda$ repeat the following step:

change in the 3-edge-connected components of G. In this case $\pi_{x,y} = \{C(x)\}$ and the lemma is trivially true.

Assume now $C(x) \neq C(y)$. By Corollary 3.1, all the simple paths between $C(x) = V_1'$ and $C(y) = V_q'$ in G' are given by $q-1$ simple cycles $\rho_1, \rho_2, \ldots, \rho_{q-1}$ in G' such that $V_1' \in \rho_1$, $V_p' \in \rho_{q-1}$, $\rho_{i-1} \cap \rho_i = \{V_i'\}$, $2 \leq i \leq q-1$, and $\rho_i \cap \rho_j = \emptyset$, $i \leq j-2$ or $i \geq j+2$. Each ρ_i, $1 \leq i \leq q-1$, is composed of two disjoint paths between V_i' and V_{i+1}', say $\rho_i^{(1)}$ and $\rho_i^{(2)}$. Then there are three edge-disjoint paths between V_i' and V_j', $1 \leq i < j \leq q$, in $\varphi(G^{new})$. The first is given by $\rho_i^{(1)}$, $\rho_{i+1}^{(1)}$, $\ldots$, $\rho_{j-1}^{(1)}$; the second by $\rho_i^{(2)}$, $\rho_{i+1}^{(2)}$, $\ldots$, $\rho_{j-1}^{(2)}$; and finally the third is given by $\rho_{i-1}^{(1)}$, $\rho_{i-2}^{(1)}$, $\ldots$, $\rho_1^{(1)}$, $(C(x), C(y))$, $\rho_{q-1}^{(1)}$, $\rho_{q-2}^{(1)}$, $\ldots$, $\rho_j^{(1)}$. Thus $V_i' \equiv_3 V_j'$.

We now show that only the vertices V_i', $1 \leq i \leq q$, become 3-edge-connected in $\varphi(G^{new})$. Let $\widetilde{V}$ be any vertex of G' different from V_i', $1 \leq i \leq q$, and let V_i, $1 \leq i \leq p$, be any vertex of G' different from $\widetilde{V}$. Since vertices of G' correspond to 3-edge-connected components of G, $\widetilde{V} \not\equiv_3 V_i$ in G'. We show that also $\widetilde{V} \not\equiv_3 V_i$ in $\varphi(G^{new})$. By Corollary 3.1, all the simple paths between $\widetilde{V}$ and V_i in G' are given by $p \geq 1$ simple cycles $\rho_1', \rho_2', \ldots, \rho_p'$ in G' such that $\widetilde{V} \in \rho_1'$, $V_i \in \rho_p'$. Consequently, if e_1 and e_2 are the two edges of ρ_1' incident to $\widetilde{V}$, $\{e_1, e_2\}$ separates $\widetilde{V}$ and V_i in G'. Therefore the removal of $\{e_1, e_2\}$ breaks G' into two pieces: G_1' and G_2' such that G_1' contains $\widetilde{V}$ and G_2' contains V_i. We claim that the removal of $\{e_1, e_2\}$ must leave $C(x)$ and $C(y)$ on the same side (i.e., either both in G_1' or both in G_2'). Indeed assume by contradiction that $C(x)$ and $C(y)$ are left in different sides by the removal of $\{e_1, e_2\}$. Without loss of generality, let $C(x)$ be in G_1' and $C(y)$ be in G_2'. This implies that $\{e_1, e_2\}$ is an edge-cut for $C(x)$ and $C(y)$ in G'. Since e_1 and e_2 are both incident to $\widetilde{V}$ in G', also the removal of $\widetilde{V}$ together with its incident edges disconnects $C(x)$ and $C(y)$ in G'. This implies that $\widetilde{V}$ is an articulation point whose removal disconnects $C(x)$ and $C(y)$ in G'. But by Corollary 3.1, this gives $\widetilde{V} = V_i'$, for some i, $1 \leq i \leq q$, contradicting our assumption. Since $C(x)$ and $C(y)$ must be either both in G_1' or both in G_2', $\{e_1, e_2\}$ still separates $\widetilde{V}$ and V_i in $\varphi(G^{new}) = G' \cup \{C(x), C(y)\}$. Consequently, $\widetilde{V} \not\equiv_3 V_i$ in $\varphi(G^{new})$.

By Theorem 1.2, all and only the vertices $V_1', V_2', \ldots, V_q'$ become 3-edge-connected in $\varphi(G^{new})$, while all the remaining vertices of $\varphi(G^{new})$ stay 2-edge-connected. By Lemma 2.2 all and only the vertices of G contained in $V_1', V_2', \ldots, V_q'$ become 3-edge-connected. □

Lemma 3.3 gives a way to compute $(G^{new})'$ from G'. As shown in the lemma, all the paths between V_1' and V_q' in G' are given by $q-1$ edge-disjoint simple cycles $\rho_1, \rho_2, \ldots, \rho_{q-1}$ in G' such that $V_1' \in \rho_1$, $V_q' \in \rho_{q-1}$, and $\rho_{i-1} \cap \rho_i = \{V_i'\}$, $2 \leq i \leq q-1$. Each ρ_i is composed of two edge-disjoint paths between V_i' and V_{i+1}', say $\rho_i^{(1)}$ and $\rho_i^{(2)}$. Furthermore, the insertion of (x, y) in G implies that all and only the vertices in V_i', $1 \leq i \leq p$, become 3-edge-connected in G. Therefore $(G^{new})'$ can be computed from G' by simply shrinking all the vertices $V_1', V_2', \ldots, V_q'$ into one vertex V'. This destroys the previous simple cycles ρ_i, $1 \leq i \leq q-1$, and creates at most $2(q-1)$ new simple cycles $\rho_i^{(1)}$ and $\rho_i^{(2)}$, $1 \leq i \leq q-1$, that are all incident to V' in $(G^{new})'$.

The changes in G' induce updates in $\mathcal{T}$, which is the block tree of G'. We denote by λ the least common ancestor of $C(x)$ and $C(y)$ in $\mathcal{T}$. The update causes the following transformations on the path $\pi_{x,y}$ of $\mathcal{T}$, to obtain $\mathcal{T}^{new}$, the block tree of $(G^{new})'$.

(i) Merge together all the square nodes $V_1', V_2', \ldots, V_q'$ in $\pi_{x,y}$ into a new square node V'.

(ii) For each round node $\rho_i \neq \lambda$ in $\pi_{x,y}$, denote by V_i' and V_{i+1}' the two square nodes in $\pi_{x,y}$

We root the tree $\mathcal{T}$ arbitrarily at any square node. Edges of $\mathcal{T}$ are considered directed from children to parent. Furthermore, for each simple cycle in G', we maintain the same cyclic order of vertices in $\mathcal{T}$ (see figure 1). In other words, let $\sigma_1, \sigma_2, \ldots, \sigma_p$ be the vertices of G' encountered in a counterclockwise walk around around a cycle ρ. Then in $\mathcal{T}$ there is one square node σ_ℓ, $1 \leq \ell \leq p$, that is the parent of ρ, and all the children of ρ from left to right are $\sigma_{\ell+1}, \sigma_{\ell+2}, \ldots, \sigma_p, \sigma_1, \sigma_2, \ldots, \sigma_{\ell-1}$. The following corollary follows easily from Lemma 3.2 and the definition of block tree of a graph.

Corollary 3.1 *Let x and y two any vertices in G, $x \not\equiv_3 y$. Let $C(x)$ and $C(y)$ be the 3-edge-connected components containing x and y respectively. Let $\pi_{x,y}$ be the path in $\mathcal{T}$ between $C(x)$ and $C(y)$ containing vertices $C(x) = \sigma_1, \rho_1, \sigma_2, \ldots, \sigma_{q-1}, \rho_{q-1}, \sigma_q = C(y)$, $q \geq 2$ (σ_i are square nodes and ρ_i are round nodes). Then $\sigma_2, \sigma_3, \ldots, \sigma_{q-1}$ are all the articulation points separating $C(x)$ and $C(y)$ in G', and all the simple paths between $C(x)$ and $C(y)$ in G' are given by the $q-1$ simple cycles ρ_1, ρ_2, ..., ρ_{q-1}.*

Besides maintaining $\mathcal{T}$, we maintain the actual 3-edge-connected components of G as disjoint sets subject to *union* and *find* operations. The name of any such set gives a pointer to the corresponding square node in $\mathcal{T}$. For each vertex x, *find*(x)=$C(x)$ (i.e., *find*(x) returns the 3-edge-connected component containing vertex x in G). With this data structure, a *Same3EdgeBlock*(x, y) operation can be simply performed by checking whether *find*(x) = *find*(y). The *union* operations will be used to update efficiently the 3-edge-connected components during *InsertEdge* operations. We recall that using the fast set-union data structures of [35] yields that any sequence of q *union* and *find* operations on a collection of n elements can be performed in $O(q\alpha(q, n))$ worst-case time.

Let x and y be any two vertices in G such that $x \not\equiv_3 y$. Denote by $C(x)$ and $C(y)$ the square nodes in $\mathcal{T}$ corresponding to the 3-edge-connected components of x and y respectively, and let $\pi_{x,y}$ be the path in $\mathcal{T}$ between $C(x)$ and $C(y)$. We now describe the updates needed in $\mathcal{T}$ because of the insertion of a new edge (x, y). As the following lemma shows, the path in $\mathcal{T}$ between $C(x)$ and $C(y)$ plays a crucial role in the update.

Lemma 3.3 *Let G be a 2-edge-connected graph. After the insertion of (x, y), all the vertices of the 3-edge-connected components corresponding to square nodes in $\pi_{x,y}$ become 3-edge-connected, while there is no change in the other 3-edge-connected components of G.*

Proof: Let $V_1, V_2, \ldots, V_p$ be the 3-edge-connected components of G before the insertion of edge (x, y). Let $V'_1, V'_2, \ldots V'_q$, $1 \leq q \leq p$, be the 3-edge-connected of G corresponding to the square nodes of $\mathcal{T}$ in $\pi_{x,y}$, as they are met while going from $C(x)$ to $C(y)$ in $\mathcal{T}$. Notice that $C(x) = V'_1$ and $C(y) = V'_q$. Let $G^{new} = G \cup \{(x, y)\}$. Similarly, let $(G^{new})'$ be the graph obtained from G^{new} by contracting its 3-edge-connected components.

Consider the graph $G'_{temp} = G' \cup \{(C(x), C(y))\}$. Notice that G'_{temp} is not necessarily $(G^{new})'$. However, since any two vertices that were 3-edge-connected in G are still 3-edge-connected in G^{new}, $V_1, V_2, \ldots, V_q$ gives a partition of the vertices of G^{new} such that if $u, v \in V_i$, then $u \equiv_3 v$. Therefore, G'_{temp} can be derived from G^{new} by applying a 3-edge-connected obeying mapping φ to G^{new}: namely, $G'_{temp} = \varphi(G^{new})$.

Assume first that $C(x) = C(y)$. Since by definition no two vertices of G' are 3-edge-connected, then no two vertices of $\varphi(G^{new})$ are 3-edge-connected either. Therefore by Lemma 2.2 there is no

are in the same 3-edge-connected component of G, and it returns false otherwise. We consider first 2-edge-connected graphs and then generalize the results to connected graphs.

Let G be a 2-edge-connected graph with n vertices subject to *Same3EdgeBlock* and *InsertEdge* operations. In this section we do not consider *AddVertex* operations since they do not preserve G 2-edge-connected; we show how to deal with them in the next section.

For any vertex x in G, denote by $C(x)$ the 3-edge-connected component containing x. We show now how to maintain information about 3-edge-connected components of a 2-edge-connected graph G during edge insertions. Let G' be the graph obtained by contracting each 3-edge-connected component of G into a super-vertex. Notice that there can be multiple edges in G'. We use interchangeably the terms 3-edge-connected component of G and super-vertex in G'. By hypothesis G is 2-edge-connected, and by Lemma 2.2 G' is 2-edge-connected too. As a result, any two super-vertices of G' lie on a common cycle. The following lemma states that G' consists of simple cycles such that any two of them share at most one super-vertex.

Lemma 3.1 *Two simple cycles of G' can intersect in at most one super-vertex.*

Proof: We proceed by contradiction. Assume there exist two different simple cycles Λ_1 and Λ_2 of G' intersecting at two super-vertices σ_1 and σ_2. Then there must be a subpath of Λ_2 between two vertices σ' and σ'' of Λ_1 (not necessarily σ_1 and σ_2) that is edge-disjoint from Λ_1. As a result, there are three edge-disjoint paths between σ' and σ'' in G'. By Theorem 1.2 $\sigma' \equiv_3 \sigma''$ in G'. By Lemma 2.2, any two vertices x and y of G such that $x \in \sigma'$ and $y \in \sigma''$ are 3-edge-connected. Consequently, σ' and σ'' cannot be 3-edge-connected components of G, a contradiction. □

Lemma 3.2 *Each biconnected component of G' is a simple cycle.*

Proof: We proceed by contradiction. Let B be a biconnected component of G' that is not a simple cycle. This implies that there are three edges e_1, e_2, and e_3 in B that do not lie in a simple cycle. However, since two edges belong to the same biconnected component if and only if they lie in a same simple cycle, there must be a simple cycle Λ_1 containing both e_1 and e_2, and a simple cycle Λ_2 containing both e_2 and e_3. Since e_2 belongs to both Λ_1 and Λ_2, Λ_1 and Λ_2 must intersect in at least two vertices, which contradicts Lemma 3.1. □

Define $\mathcal{T}$ as the block tree of G'. We recall here that given a graph G, the block tree of G is defined as follows (see for instance reference [16]). The nodes of the block tree are partitioned into *square* and *round* nodes: square nodes correspond to the vertices of G, while round nodes correspond to the biconnected components of G. There is an edge between a square node u and a round node ρ if and only if vertex u of G is in biconnected component ρ. Also, there is an edge between a square vertex u and round vertices ρ_1 and ρ_2 if and only if u is an articulation point separating biconnected components ρ_1 and ρ_2 in G.

In the following, we refer to $\mathcal{T}$ as the *tree of 3-edge-connected components* of G. Notice that each 3-edge-connected component of G corresponds uniquely to a super-vertex of G' and to a square node in $\mathcal{T}$. In the remainder of this section we use interchangeably the terms 3-edge-connected component of G, super-vertex of G' and square node of $\mathcal{T}$.

Figure 1 shows a graph G, the graph G' and the block tree $\mathcal{T}$.

[Figure 1]

corresponding edge in $\varphi(G)$. Notice that $\varphi(G)$ can contain multiple edges and self-loops. For any $E' \subseteq E$, let $\varphi(E') = \{\varphi(e) | e \in E'\}$. As consequence of this definition, $|\varphi(E')| = |E'|$ for any edge set $E' \subseteq E$. Similarly, if $x \in V_i$ we define $\varphi(x) = \varphi(V_i)$. In the sequel such a φ will be referred to as a *k-edge-connectivity obeying mapping.*

The following lemmas characterize some combinatorial properties of the k-edge-connected components of a graph.

Lemma 2.1 *Let φ be a k-edge-connectivity obeying mapping. A set of edges E', $|E'| \leq k-1$, is an edge-cut for x and y in G if and only if $\varphi(E')$ is an edge-cut for $\varphi(x)$ and $\varphi(y)$ in $\varphi(G)$.*

Proof: Assume $\varphi(E')$ is not an edge-cut for $\varphi(x)$ and $\varphi(y)$ in $\varphi(G)$. Thus, there is a path $\varphi(\pi)$ in $\varphi(G)$ between $\varphi(x)$ and $\varphi(y)$ containing no edges in $\varphi(E')$. Let $\varphi(e_1), \varphi(e_2), \ldots, \varphi(e_p)$ be the edges in $\varphi(\pi)$, and let $e_1, e_2, \ldots, e_p$ be the corresponding edges in G. Clearly, $e_i \notin E'$, $1 \leq i \leq q$. However, $e_1, e_2, \ldots, e_p$ does not give a path in G. Let $e_i = (u_i, v_i)$, $1 \leq i \leq q$. Denote x by v_0 and y by u_{p+1}. By definition of $\varphi(G)$, $v_i \equiv_k u_{i+1}$, $0 \leq i \leq p$. By Theorem 1.2, there are at least k edge-disjoint paths in G between v_i and u_{i+1}, $0 \leq i \leq p$. Since $|E'| \leq k-1$, there is at least one path π_i between v_i and u_{i+1}, $0 \leq i \leq p$, containing no edges in E'. As a result, $\pi_0 \cdot e_1 \cdot \pi_1 \cdot e_2 \cdot \ldots \cdot e_p \cdot \pi_p$ gives a path in G between x and y avoiding edges of E'. Hence E' is not an edge-cut for x and y in G.

Conversely, assume E' is not an edge-cut for x and y in G. That is, there is a path $\pi = \{x = v_0, v_1, \ldots, v_{q-1}, v_q = y\}$ in G containing no edges of E'. Then $\varphi(\pi) = \{\varphi(x), \varphi(v_1), \ldots, \varphi(v_{q-1}), \varphi(y)\}$ gives a path between $\varphi(x)$ and $\varphi(y)$ in $\varphi(G)$ that uses no edges of $\varphi(E')$. Therefore $\varphi(E')$ cannot be an edge-cut for $\varphi(x)$ and $\varphi(y)$ in $\varphi(G)$. □

Lemma 2.2 *Let φ be a k-edge-connectivity obeying mapping. Let $\varphi(V_i)$ and $\varphi(V_j)$ be any two vertices in $\varphi(G)$, and let x and y be any two vertices of G, $x \in V_i$, $y \in V_j$. Given any integer h, $1 \leq h \leq k$, $\varphi(V_i)$ and $\varphi(V_j)$ are h-edge-connected in $\varphi(G)$ if and only if x and y are h-edge-connected in G.*

Proof: If $V_i = V_j$ the lemma is trivial.

Assume $i \neq j$ and let V_i and V_j be such that $\varphi(V_i) \not\equiv_h \varphi(V_j)$ in the contracted graph $\varphi(G)$. This means that a there is a min-edge-cut for $\varphi(V_i)$ and $\varphi(V_j)$ in $\varphi(G)$ containing at most $h-1$ edges. Let A be such such a min-edge-cut, $|A| \leq h-1$. Let x be any vertex in V_i and let y be any vertex in V_j. Let $\bar{E} = \{e \in E | \varphi(e) \in A\}$. By Lemma 2.1, $\bar{E}$ is an edge-cut for x and y in G. Since $|\bar{E}| = |A| \leq h-1$, $x \not\equiv_h y$ in G.

Let x and y be vertices in G such that $x \not\equiv_h y$. Let $x \in V_i$, $y \in V_j$, $i \neq j$. Let E', $|E'| \leq h-1$, be an edge-cut for x and y in G. Then by Lemma 2.1, $\varphi(E')$, $|\varphi(E')| = |E'| \leq h-1$, is an edge-cut for $\varphi(V_i)$ and $\varphi(V_j)$ in $\varphi(G)$. Therefore, $\varphi(V_i) \not\equiv_h \varphi(V_j)$ in $\varphi(G)$. □

3 Maintaining 3-Edge-Connected Components of 2-Edge-Connected Graphs

In this section we show how to maintain efficiently the 3-edge-connected components of a graph G under any sequence of *InsertEdge*, *AddVertex* and *Same3EdgeBlock* operations. We recall here that *InsertEdge*(x, y) adds a new edge between vertices x and y, *AddVertex*(u, v) inserts a new vertex u and connects it to vertex v, and *Same3EdgeBlock*(u, v) returns true if vertices u and v

In the last decade there has been a growing interest in dynamic problems on graphs. In particular, much attention has been devoted to (among others) the dynamic maintenance of connected components [9, 10, 26, 28], 2- and 3-connectivity [6, 14, 37, 38], transitive closure [2, 20, 21, 22, 24, 32, 33, 39], planarity [5, 6, 31], shortest paths [1, 4, 8, 25, 27, 30, 39], and minimum spanning trees [3, 7, 10, 30]. In these problems one would like to answer queries on graphs that are undergoing a sequence of updates, for instance, insertions and deletions of edges and vertices. A problem is said to be *fully dynamic* if the update operations include both insertions and deletions of edges. On the other hand, a problem is called *partially dynamic* if only one type of update, i.e., either insertions or deletions, is allowed. The goal of a (fully or partially) dynamic graph algorithm is to update efficiently the solution of a problem after dynamic changes, rather than having to recompute it from scratch each time. Given their powerful versatility, it is not surprising that dynamic algorithms and dynamic data structures are usually more difficult to design than their static counterparts.

In this paper we study the partially dynamic problem of maintaining the 3-edge-connected components of a connected graph during edge insertions. We wish to maintain the graph under an intermixed sequence of operations of the following kinds.

> *Same3EdgeBlock*(u, v): Return *true* if vertices u and v are in the same 3-edge-connected component. Return *false* otherwise.
>
> *InsertEdge*(x, y): Insert a new edge between the two vertices x and y.
>
> *AddVertex*(u, v): Add a new vertex u to the graph, and connect it to G by means of the edge (u, v).

We give algorithms that support any sequence of q *InsertEdge*, *AddVertex*, and *Same3EdgeBlock* operations on an initially connected graph with n vertices in a total of $O((n+q)\alpha(q, n))$ time. Notice that the best known bound for the twin problem of maintaining in a partially dynamic fashion the triconnected components of a graph is $O(q + n \log n)$ [6], and the algorithms given in [6] achieve the $O(q\alpha(q, n))$ bound only in the case of an initially biconnected graph.

Our techniques combine a variety of graph properties, and data structures. We represent the 3-edge-connected components of a graph by means of a tree structure, and update and query this tree instead of the original graph. Indeed, we represent a graph G by a tree of 2-edge-connected components, each of which is itself represented as a tree of 3-edge-connected components. We then develop efficient data structures for updating this representation during edge and vertex insertions.

The remainder of the paper consists of four sections. Section 2 gives some graph-theoretical properties of k-edge-connectivity. Section 3 gives our bounds for 2-edge-connected graphs, and Section 4 extends these bounds to connected graphs. In Section 5 we list some concluding remarks.

2 Some Properties of k-edge Connectivity

Let $G = (V, E)$ be an undirected graph. Let $V_1, V_2, \ldots, V_p$ be a partition of the vertices of $G = (V, E)$ such that if $x, y \in V_i$, $1 \leq i \leq p$, then $x \equiv_k y$. Notice that $V_1, V_2, \ldots, V_p$ are not necessarily the k-edge-connected components of G since we can have $u \equiv_k v$ for $u \in V_i$ and $v \in V_j$, $i \neq j$. Let $\varphi(G) = (\varphi(V), \varphi(E))$ be the graph obtained from G by shrinking each V_i, $1 \leq i \leq p$, into one super-vertex $\varphi(V_i)$. If $e = (u, v)$ is an edge of G, $u \in V_i$, $v \in V_j$, let $\varphi(e) = (\varphi(V_i), \varphi(V_j))$ be the

1 Introduction

Given an undirected graph $G = (V, E)$, and an integer $k \geq 2$, a pair of vertices $\langle u, v \rangle$ is said to be *k-edge-connected* if the removal of any $k-1$ edges in G leaves u and v connected. This is an equivalence relationship, and we denote it by $\equiv_k$, i.e., if a pair of vertices $\langle x, y \rangle$ is k-edge-connected we write $x \equiv_k y$. The vertices of a graph G are partitioned by this relationship into equivalence classes called *k-edge-connected components*. An edge set $E' \subseteq E$ is an *edge-cut* for vertices x and y if the removal of all the edges in E' disconnects x and y. The cardinality of an edge-cut E', denoted by $|E'|$, is given by the number of edges in E'. An edge-cut E' for x and y is said to be a *minimum cardinality edge-cut* or in short a *min-edge-cut* if there is no other edge-cut E'' for x and y such that $|E''| < |E'|$. Then $x \equiv_k y$ if and only if a min-edge-cut for x and y contains at least k edges. A graph G is said to be *k-edge-connected* if all its pairs of vertices are k-edge-connected. A min-edge-cut of cardinality 1 is called a *bridge*. Similarly, a vertex set $V' \subseteq V - \{x, y\}$ is a *vertex-cut* for vertices x and y if the removal of all the vertices in V' disconnects x and y. The cardinality of a vertex-cut V', denoted by $|V'|$, is given by the number of vertices in V'. A vertex-cut V' for x and y is said to be a *minimum cardinality vertex-cut* or in short a *min-vertex-cut* if there is no other vertex-cut V'' for x and y such that $|V''| < |V'|$. Then x and y are k-vertex-connected if and only if a min-vertex-cut for x and y contains at least k vertices. A graph G is said to be *k-vertex-connected* if all its pairs of vertices are k-vertex-connected. A min-vertex-cut of cardinality 1 is called an *articulation point*.

The following theorems due to Menger, Ford and Fulkerson, and Elias, Feinstein and Shannon (see for instance [16]) give another characterization of k-vertex and k-edge connectivity.

Theorem 1.1 [Menger] *Given a graph G and two vertices x and y in G, x and y are k-vertex-connected if and only if there are at least k vertex-disjoint paths between x and y.*

Theorem 1.2 [Ford and Fulkerson, Elias, Feinstein and Shannon] *Given a graph G and two vertices x and y in G, x and y are k-edge-connected if and only if there are at least k edge-disjoint paths between x and y.*

Vertex connectivity and edge connectivity problems arise naturally in many applications and have been extensively studied. Much attention has been given to the 2-vertex-connected (or biconnected) components and 2-edge-connected (or bridge-connected) components of a graph. Tarjan [34] gives optimal linear-time sequential algorithms. Tarjan and Vishkin [36] give logarithmic-time parallel algorithms. Recently, Westbrook and Tarjan [38] considered the problem of maintaining biconnected components and bridge-connected components undergoing any sequence of edge and vertex insertions. They presented algorithms that run in a total of $O(q\alpha(q, n))$ time, where q is the total number of operations, n is the number of vertices and α is a functional inverse of Ackermann's function. As for 3-vertex-connectivity, Hopcroft and Tarjan [18] showed how to compute the triconnected components of a graph in linear time. Di Battista and Tamassia [6] showed how to maintain the triconnected components of a graph during edge insertions in a total of $O(q + n \log n)$ time [1], where q is the total number of operations performed and n is the number of vertices. Their algorithm achieves an $O(q\alpha(q, n))$ bound only in the case of an initially biconnected graph.

[1] In this paper all the logarithms are assumed to be base two.

Maintaining the 3-Edge-Connected Components of a Graph On-Line*

Zvi Galil
Columbia University, and
Tel-Aviv University

Giuseppe F. Italiano†
Columbia University, and
Università di Roma, Italy

Abstract

We study the problem of maintaining the 3-edge-connected components of a graph undergoing repeated dynamic modifications, such as edge and vertex insertions. We show how to answer whether two vertices belong to the same 3-edge-connected component of a connected graph that is undergoing only edge insertions. Any sequence of q query and updates on an n-vertex graph can be performed in $O((n+q)\alpha(q,n))$ time.

Key Words: Analysis of algorithms, dynamic data structures, edge connectivity, vertex connectivity.

*Work partially supported by NSF Grants CCR-8814977, CCR-9014605, by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (Project ALCOM), and by the Italian MURST Project "Algoritmi e Strutture di Calcolo". Portions of this paper appear in [*Proc. 23rd Annual ACM Symp. on Theory of Computing*].

†Partially supported by an IBM Graduate Fellowship. Part of this work was done while visiting the IBM T. J. Watson Research Center and Universität des Saarlandes, Saarbrücken, Germany.