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

[61] D. M. Yellin and R. Strom. INC: a language for incremental computations. In *Proc. ACM SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 115–124, 1988.

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

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

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

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

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

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 Unpublished manuscript, 1990.

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

[51] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. Syst. Sci.*, 18:110–127, 1979.

[52] R. E. Tarjan. *Data structures and network algorithms*, volume 44. CBMS-NSF Regional Conference Series in Applied Mathematics, SIAM, 1983.

[53] R. E. Tarjan. Amortized computational complexity. *SIAM J. Alg. Disc. Meth.*, 6:306–318, 1985.

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

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

[56] J. Westbrook. *Algorithms and data structures for dynamic graph problems.* PhD thesis, Department of Computer Science, Princeton University, October 1989. Technical Report CS-TR-229-89.

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

[58] M. Yannakakis. Graph theoretic methods in database theory. In *Proc. ACM Conf. on Principles of Database Systems*, pages 230–242, 1990.

[59] A. C. Yao. Should tables be sorted? *J. Assoc. Comput. Mach.*, 31:245–281, 1984.

[28] N. Horspool. Incremental generation of LR parsers. Technical report, Department of Computer Science, University of Victoria, 1988.

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

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

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

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

[33] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. on Database Systems*, 1990.

[34] D. B. Johnson. A priority queue in which initialization and queue operations take $O(\log\log D)$ time. *Math. Syst. Th.*, 15:295–309, 1982.

[35] S. Kaplan. Incremental attribute evaluation on graphs. Technical Report UIUC-DCS-86-18309, University of Illinois at Urbana-Champaign, 1986.

[36] E. L. Lawler. *Combinatorial optimization: networks and matroids.* Holt, Rinehart and Winston, 1976.

[37] K. Mehlhorn. Personal communication, 1990.

[38] K. Mehlhorn, S. Näher, and H. Alt. A lower bound for the complexity of the union-split-find problem. *SIAM J. Comput.*, 17:1093–1102, 1988.

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

[40] J. H. Reif. A topological approach to dynamic graph connectivity. *Inform. Process. Lett.*, 25:65–70, 1987.

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

[42] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Comput.*, 17:1253–1262, 1988.

[43] A. Schönhage. Storage modification machines. *SIAM J. Comput.*, 9:490–508, 1980.

[11] M. D. Carrol and B. G. Ryder. Incremental data flow analysis via dominator and attribute updates. In *Proc. 15th Annual ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages*, pages 274–284, 1988.

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

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

[14] Charles Elkan. A decision procedure for conjunctive query disjointness. In *Proc. ACM Conf. on Principles of Database Systems*, pages 134–139, 1989.

[15] 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. In *Proc. First Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 1–11, 1990.

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

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

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

[19] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. Assoc. Comput. Mach.*, 34:596–615, 1987.

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

[21] Z. Galil and G. F. Italiano. Fully dynamic algorithms for 3-edge-connectivity. In preparation.

[22] M. R. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of NP-Completeness.* W. H. Freeman, 1979.

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

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

[25] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13:338–355, 1984.

[26] R. Hoover. *Incremental Graph Evaluation.* PhD thesis, Department of Computer Science, Cornell University, 1987. Technical Report 87-836.

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

Bibliography

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, 1989.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[3] B. Alpern, R. Hoover, B. Rosen, P. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. In *Proc. 1st Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 32–42, 1990.

[4] G. Ausiello and G. F. Italiano. On-line algorithms for polynomially solvable satisfiability problems. *J. of Logic Programming*, 10(1):69–90, 1991.

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

[6] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange networks and PRAM. *IEEE Trans. on Computers*, C-36:1258–1263, 1987.

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

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

[9] A. L. Buchsbaum, P. C. Kanellakis, and J. S. Vitter. A data structure for arc insertion and regular path finding. In *Proc. 1st Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 22–31, 1990.

[10] M. Burke and B. G. Ryder. Incremental iterative data flow analysis algorithms. Technical Report LCSR-TR-96, Department of Computer Science, Rutgers University, 1987.

operations on an initially connected graph with n vertices in a total of $O((n+q)\alpha(q,n))$ time. We have recently 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 [21]. One might ask whether there is any efficient (either partially or fully) dynamic algorithm for maintaining the k-edge-connected components of a graph, $k \geq 4$.

We notice that for many graph theoretical problems there is no known efficient partially dynamic algorithm for handling edge deletions only. For instance, the best bound known for many dynamic problems with edge deletions only (including maintaining a minimum spanning tree, the connected components, or the 2-edge-connected components of a graph) is the same as the bound known for the fully dynamic versions of these problems. Since, as shown in Chapter 1, dynamic problems with edge insertions only can be solved much more efficiently than the corresponding fully dynamic problems, it seems quite natural to ask whether more efficient techniques for handling edge deletions only can be designed.

Chapter 6

Conclusions and Further Research

In this dissertation we have introduced several new dynamic algorithms and new dynamic data structures for graph theoretical problems, such as edge connectivity, vertex connectivity and shortest paths. The solutions proposed share many common features, and combine a variety of graph properties and novel algorithmic techniques.

We have described fast algorithms for maintaining a solution to the All Pairs Shortest Paths Problem under edge insertions and edge costs decreases on a directed graph with integer edge costs in $[1 \ldots C]$. The total time required to perform a sequence of at most $O(Cn^2)$ edge insertions and edge cost decreases is $O(Cn^3 \log(nC))$, and the space complexity is $O(n^2)$. We have generalized this result to longest paths in directed acyclic graphs. There are several related problems which seem worth further investigation. First, in our approach questions about the solution are answered quickly while updating the data structure may be expensive. Is there any trade-off between these two kinds of operations? Moreover, given non-integer edge costs, is it possible to do better than previously known algorithms? Finally, the impact of deletions on this problem deserves further investigation.

Next, we have described efficient algorithms for maintaining the 2-edge-connected components of an undirected graph under insertions and deletions of edges. We have shown how to support each operation in $O(m^{2/3})$ for general graphs, and in $O(\sqrt{n \log\log n})$ for planar graphs. Then we have presented a new data structure that is able to maintain the biconnected components of a planar graph under insertions and deletions of edges. Each operation is supported in $O(n^{2/3})$ time in the worst case, where n is the total number of vertices in the graph. The problem of maintaining the biconnected components of a general graph during edge insertions and deletions deserves further study.

Finally, we have studied the partially dynamic 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*

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 [57]. 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 14.

As for an *InsertEdge*(u,v) operation, the bridge-block tree can be updated in $O(\alpha(q,n))$ amortized time [57]. 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 14, 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 38 *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 [50]. 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 by Corollary 2, and sums up to a total time of $O(|V_0| + |E_0|)$. □

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.

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).

them as B'_i, $0 \leq i \leq \ell$. For each one of them we can repeat the same argument given after Lemma 35. 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 37 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 [57]. 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 37, 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 15 *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 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 14,

Figure 5.5: Tree of 2-edge-connected components of a graph

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 5.5).

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

Lemma 37 *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 35. 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 the proof of Lemma 35, 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 37 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

5.5 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 [57].

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 30 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 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 33 still holds for G'. As far as Lemma 34 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 [57]. 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

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 36, 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 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 [57]. □

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 [20]. 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 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 36 allows us to prove the following theorem:

Theorem 14 *The data structure above supports any sequence of* q Same3EdgeBlock *and* InsertEdge *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

Figure 5.4: Dealing with vertex λ

in the same 3-edge-connected component and by Lemma 35 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:

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

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 5.4(a). Otherwise, λ is a square node; merge β_1, β_2 and λ as shown in Figure 5.4(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 $CompressPath(u, v, \mathcal{T})$.

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

Lemma 36 *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* [57], 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 [57] 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 [54], each *merge* and *parent* primitive can be supported in $O(\alpha(p, n))$ amortized time.

Figure 5.3: Merge and split primitives

Figure 5.2: Updates needed after inserting a new edge

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 $\pi_{x,y}$ 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 5.2 shows the updates needed in the graph of Figure 5.1 as a result of the insertion of a new edge.

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 5.3a). 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 5.3b), 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 5.3 and therefore allows a primitive *merge* to be applied in the resulting tree.

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

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 $\rho_i^{(1)}$, $\rho_{i+1}^{(1)}$, ..., $\rho_{j-1}^{(1)}$; the second by $\rho_i^{(2)}$, $\rho_{i+1}^{(2)}$, ..., $\rho_{j-1}^{(2)}$; and finally the third is given by $\rho_{i-1}^{(1)}$, $\rho_{i-2}^{(1)}$, ..., $\rho_1^{(1)}$, $(C(x),C(y))$, $\rho_{q-1}^{(1)}$, $\rho_{q-2}^{(1)}$, ..., $\rho_j^{(1)}$. 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, 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, 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 12, 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 30 all and only the vertices of G contained in $V_1', V_2', \ldots, V_q'$ become 3-edge-connected. □

Lemma 35 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

$C(x)$ 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 [54] 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 35 *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 30 there is no 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, 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 $i \geq j+2$.

Figure 5.1: A graph G, the graph G' and the block tree $\mathcal{T}$

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 12 $\sigma' \equiv_3 \sigma''$ in G'. By Lemma 30, 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 34 *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 33. □

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 [23]). 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 5.1 shows a graph G, the graph G' and the block tree $\mathcal{T}$.

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 5.1). 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 34 and the definition of block tree of a graph.

Corollary 3 *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*

arc-vertices $\varphi(e_1), \varphi(e_2), \ldots, \varphi(e_p)$, a contradiction.

Consequently, the graph G is k-edge-connected if and only if $\varphi_k(G)$ is k-vertex-connected. □

The following corollary is an immediate consequence of Theorem 13 and of the linear-time algorithm of Hopcroft and Tarjan [27] for 3-vertex-connectivity.

Corollary 2 *A graph G can be checked for 3-edge-connectivity in $O(m+n)$ time. Its 3-edge-connected components can be computed in the same bound.*

5.4 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 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.

The main idea behind our data structure is to 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.

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 30 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 33 *Two simple cycles of G' can intersect in at most one super-vertex.*

On the other hand, consider a simple path $\pi_{\varphi_k(G)}$ between node-vertices $\varphi(v^1)$ and $\varphi(v^p)$ in $\varphi_k(G)$. Let $\varphi(v^1), \varphi(v^2), \ldots, \varphi(v^p)$ be the node-vertices in the path $\pi_{\varphi_k(G)}$. For $1 \leq i \leq p-1$, let $\pi^i_{\varphi_k(G)}$ be the subpath of $\pi_{\varphi_k(G)}$ between $\varphi(v^i)$ and $\varphi(v^{i+1})$. Since there are no edges in $\varphi_k(G)$ between any two node-vertices, each subpath $\pi^i_{\varphi_k(G)}$, $1 \leq i \leq p-1$, contains only arc-vertices. Fix any $\pi^i_{\varphi_k(G)}$, $1 \leq i \leq p-1$. By construction, $\pi^i_{\varphi_k(G)}$ must consist of pieces of wheels of $\varphi_k(G)$, say wheels $W^1_i, W^2_i, \ldots, W^{\ell_i}_i$, $\ell_i \geq 1$, such that $\varphi(v^i)$ belongs to wheel W^1_i and $\varphi(v^{i+1})$ belongs to wheel $W^{\ell_i}_i$. Namely, $\pi^i_{\varphi_k(G)}$ can be broken up into ℓ_i pieces $\pi^{i,1}_{\varphi_k(G)}, \pi^{i,2}_{\varphi_k(G)}, \ldots, \pi^{i,\ell_i}_{\varphi_k(G)}$ such that piece $\pi^{i,j}_{\varphi_k(G)}$ consists only of edges in wheel W^j_i, $1 \leq j \leq \ell_i$, and pieces $\pi^{i,j}_{\varphi_k(G)}$ and $\pi^{i,j+1}_{\varphi_k(G)}$, $1 \leq j \leq \ell_i - 1$, share a common endpoint. This common endpoint is an arc-vertex of $\varphi_k(G)$ that belongs to wheels W^i_j and W^i_{j+1}, and we denote it by $\varphi(e^{i,j})$. In other words, the subpath $\pi^i_{\varphi_k(G)}$ starts with edges of wheel W^1_i ending at arc-vertex $\varphi(e^{i,1})$, then continues with edges of wheel W^2_i ending at arc-vertex $\varphi(e^{i,2})$, and so on until it ends with edges of wheel $W^{\ell_i}_i$. We now show that there is a path π^i_G in G going through the vertices corresponding to the hubs of the wheels $W^1_i, W^2_i, \ldots, W^{\ell_i}_i$.

Indeed, let v^i be the vertex in G corresponding to node-vertex $\varphi(v^i)$, $1 \leq i \leq p$, and let $e^{i,j}$ be the edge of G corresponding to arc-vertex $\varphi(e^{i,j})$, $1 \leq i \leq p$, $1 \leq j \leq \ell_i - 1$. We construct a path π^i_G between vertices v^i and v^{i+1} in G that corresponds to subpath $\pi^i_{\varphi_k(G)}$ in $\varphi_k(G)$ as follows. If $v^i = v^{i+1}$, then $\pi^i_G = \emptyset$. If $v^i \neq v^{i+1}$ then let $W^1_i, W^2_i, \ldots, W^{\ell_i}_i$ be the wheels of $\varphi_k(G)$ in the subpath $\pi^i_{\varphi_k(G)}$. Let $u^{i,j}$ be the vertex of G corresponding to the hubs of the wheel W^j_i, $1 \leq j \leq \ell_i$. Notice that $v^i = u^{i,1}$ and $v^{i+1} = u^{i,\ell_i}$. Then by construction the edge $e^{i,j}$ in G is between vertices $u^{i,j}$ and $u^{i,j+1}$, $1 \leq j \leq \ell_i - 1$. Thus, the path π^i_G consists of vertices $v^i, u^{i,2}, \ldots, u^{i,\ell_i - 1}, v^{i+1}$. Notice that if $\pi^i_{\varphi_k(G)}$ avoids arc-vertex $\varphi(e)$ in $\varphi_k(G)$ then π^i_G avoids edge e in G. Putting together all the subpaths π^i_G, $1 \leq i \leq p-1$, gives a path $\pi_G = \pi^1_G \cdot \pi^2_G \cdot \ldots \cdot \pi^{p-1}_G$ between vertices v^1 and v^p in G. Indeed, since $\pi_{\varphi_k(G)}$ is a simple path, it has no repetition of vertices; consequently, π_G cannot have repetition of edges and therefore it is a path. We define π_G as the path of G corresponding to $\pi_{\varphi_k(G)}$. Again, if $\pi_{\varphi_k(G)}$ avoids arc-vertex $\varphi(e)$ in $\varphi_k(G)$, then π_G avoids edge e in G.

Let x and y be two vertices which are not k-edge-connected in G. Then there exist an edge-cut E' in G of size at most $k-1$ such that the removal of edges in E' separates x and y. Let $E' = \{e_1, e_2, \ldots, e_p\}$, $p \leq k-1$. We claim that $\varphi(E') = \{\varphi(e_1), \varphi(e_2), \ldots, \varphi(e_p)\}$ is a vertex-cut for $\varphi(x_1)$ and $\varphi(y_1)$ in $\varphi_k(G)$. Indeed if it is not, then there is a simple path $\pi_{\varphi_k(G)}$ between $\varphi(x_1)$ and $\varphi(y_1)$ in $\varphi_k(G)$ avoiding vertices in $\varphi(E')$, and the corresponding path π_G in G between x and y avoids edges in E', a contradiction. Conversely, let $\varphi(x_i)$ and $\varphi(y_j)$ be two node-vertices which are not k-vertex-connected in $\varphi_k(G)$. Since by Lemma 32 no node-vertex can be in a min-vertex-cut in $\varphi_k(G)$ then there exists a min-vertex-cut $\varphi(V') = \{\varphi(e_1), \varphi(e_2), \ldots, \varphi(e_p)\}$, $p \leq k-1$, separating $\varphi(x_i)$ and $\varphi(y_j)$ and consisting only of arc-vertices. Then $\{e_1, e_2, \ldots, e_p\}$ is an edge-cut for x and y in G: again, if it is not, then there is a simple path between x and y in G avoiding edges $e_1, e_2, \ldots, e_p$, and therefore a path $\pi_{\varphi_k(G)}$ between $\varphi(x_i)$ and $\varphi(y_j)$ in $\varphi_k(G)$ avoiding

hubs $\varphi(v_1), \varphi(v_2), \ldots, \varphi(v_{k-2})$, and vertices $\varphi(e_0)$, $\varphi(e_1)$, ..., $\varphi(e_{d-1})$. This implies that $\varphi_k(G)$ has $O(m+kn)$ vertices and $O(km)$ edges and can be computed in time $O(k(m+n))$ from G.

Lemma 31 *Let v be any vertex of G and let e_i, $0 \leq i \leq d-1$, be the edges of G incident to v. Given any two arc-vertices $\varphi(e_r)$ and $\varphi(e_s)$, $0 \leq r, s \leq d-1$, in $\varphi_k(G)$, there are k vertex-disjoint paths between $\varphi(e_r)$ and $\varphi(e_s)$.*

Proof: By definition of $\varphi_k(G)$ there are $k-2$ paths going through vertices $\varphi(v_j)$, $1 \leq j \leq k-2$. Furthermore, the cycle $\varphi(e_0), \varphi(e_1), \ldots, \varphi(e_{d-1})$ gives rise to two other paths between $\varphi(e_r)$ and $\varphi(e_s)$, and these k paths are vertex-disjoint. □

Lemma 32 *No node-vertex can be in a min-vertex-cut of size at most $k-1$ in $\varphi_k(G)$.*

Proof: Assume by contradiction that there is a min-vertex-cut V', $|V'| \leq k-1$, in $\varphi_k(G)$ containing a node-vertex $\varphi(v_j)$, $1 \leq j \leq k-2$. This implies that the subgraph of $\varphi_k(G)$ induced by $\varphi(V) - V'$ consists of $\ell \geq 2$ connected components, say $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_\ell$.

Let $V'' = V' - \{\varphi(v_j)\}$. Since V' is a min-vertex-cut, the subgraph of $\varphi_k(G)$ induced by $\varphi(V) - V''$ must be connected. Thus, the vertex $\varphi(v_j)$ is adjacent to every connected component $\mathcal{A}_i$, for $1 \leq i \leq \ell$. Let v the vertex in G corresponding to $\varphi(v_j)$, and let $e_0, e_1, \ldots, e_{d-1}$ be the edges of G incident to v. By construction, $\varphi(v_j)$ is adjacent only to arc-vertices $\varphi(e_0), \varphi(e_1), \ldots, \varphi(e_{d-1})$ in $\varphi_k(G)$. Notice that all the arc-vertices $\varphi(e_0), \varphi(e_1), \ldots, \varphi(e_{d-1})$ cannot be in the min-vertex-cut V', otherwise $\varphi(V) - V''$ would not be connected. By Lemma 31 and Theorem 11, any two arc-vertices $\varphi(e_r)$ and $\varphi(e_s)$, $0 \leq r, s \leq d-1$, are k-vertex-connected in $\varphi_k(G)$, and therefore all the arc-vertices $\varphi(e_0), \varphi(e_1), \ldots, \varphi(e_{d-1})$ that are not in the min-vertex-cut V' must belong to only one component $\mathcal{A}_i$, a contradiction. □

Theorem 13 *G is k-edge-connected if and only if $\varphi_k(G)$ is k-vertex-connected.*

Proof: We first show that there is a correspondence between paths in G and paths in $\varphi_k(G)$. That is, given a simple path π_G between vertices x and y in G, there is a corresponding path $\pi_{\varphi_k(G)}$ between node-vertices $\varphi(x_1)$ and $\varphi(y_1)$ in $\varphi_k(G)$. Conversely, given a simple path $\pi_{\varphi_k(G)}$ between node-vertices $\varphi(x_i)$ and $\varphi(y_j)$ in $\varphi_k(G)$, there is a corresponding path π_G between x and y in G.

Consider a simple path π_G between vertices $v^{(1)}$ and $v^{(p)}$ in G containing vertices $v^{(1)}, v^{(2)}, \ldots, v^{(p)}$ and edges $e^{(1)}, e^{(2)}, \ldots, e^{(p-1)}$, where $e^{(i)} = (v^{(i)}, v^{(i+1)})$, $1 \leq i \leq p-1$. By construction, there is a path $\pi_{\varphi_k(G)}$ in $\varphi_k(G)$ containing vertices $\varphi(v_1^{(1)})$, $\varphi(e^{(1)})$, $\varphi(v_1^{(2)})$, $\varphi(e^{(2)})$, ... , $\varphi(v_1^{(p-1)})$, $\varphi(e^{(p-1)})$, $\varphi(v_1^{(p)})$. Define $\pi_{\varphi_k(G)}$ as the path of $\varphi_k(G)$ corresponding to π_G. Notice that if π_G avoids edge e in G, then $\pi_{\varphi_k(G)}$ avoids arc-vertex $\varphi(e)$ in $\varphi_k(G)$.

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 30 *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 29, $\bar{E}$ 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 29, $\varphi(E')$, $|\varphi(E')| = |E'| \leq h-1$, 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 $\varphi(G)$. □

5.3 A Linear-time Algorithm for 3-Edge Connectivity

In this subsection we show how to reduce k-edge connectivity to k-vertex connectivity. Using this reduction, we achieve an $O(m+n)$ bound for the problem of deciding whether a graph is 3-edge-connected and for the problem of computing the 3-edge-connected components of a graph. Despite its simplicity, we have not found this result in the literature.

We show how to reduce k-edge connectivity to k-vertex connectivity, for $k \geq 3$. Given an undirected graph $G = (V, E)$, define the graph $\varphi_k(G) = (\varphi(V), \varphi(E))$ as follows. For every vertex v of G, there are $k-2$ vertices $\varphi(v_1), \varphi(v_2), \ldots, \varphi(v_{k-2})$ in $\varphi_k(G)$. We say that $\varphi(v_1), \varphi(v_2), \ldots, \varphi(v_{k-2})$ are *node-vertices* of $\varphi_k(G)$. Furthermore, for every edge e of G, there is a vertex $\varphi(e)$ in $\varphi_k(G)$, referred to as an *arc-vertex* of $\varphi_k(G)$.

Let v be any vertex of G and let $u_0, u_1, \ldots, u_{d-1}$ be the vertices adjacent to v. Let $e_i = (v, u_i)$, $0 \leq i \leq d-1$. Then there are edges $(\varphi(e_i), \varphi(e_{(i+1) \bmod d}))$, $0 \leq i \leq d-1$, and $(\varphi(e_i), \varphi(v_j))$, $0 \leq i \leq d-1$, $1 \leq j \leq k-2$, in $\varphi_k(G)$. Notice that $\varphi_k(G)$ is obtained from G by replacing a vertex v and its incident edges e_0, e_1, …, e_{d-1}, by a wheel with $k-2$

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 $O(q + n \log n)$ [8], and the algorithms given in [8] 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 chapter consists of four sections. Section 2 gives some graph-theoretical properties of k-edge-connectivity. Section 3 gives a linear-time algorithm to compute the 3-edge-connected components of a graph. Section 4 gives our bounds for 2-edge-connected graphs, and Section 5 extends these bounds to connected graphs.

5.2 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 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 29 *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 12, there are at least k edge-disjoint paths in G

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 [23]) give another characterization of k-vertex and k-edge connectivity.

Theorem 11 [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 12 [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 [50] gives optimal linear-time sequential algorithms. Tarjan and Vishkin [55] give logarithmic-time parallel algorithms. Recently, Westbrook and Tarjan [57] 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 [27] showed how to compute the triconnected components of a graph in linear time. Di Battista and Tamassia [8] showed how to maintain the triconnected components of a graph during edge insertions in a total of $O(q + n\log n)$ time, 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.

In this chapter 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 $O((n+q)\alpha(q,n))$

Chapter 5

Partially Dynamic Algorithms for 3-Edge Connectivity

This chapter deals with 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.

5.1 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

4.5 Concluding Remarks

We remark that our data structure is also able to return an articulation point separating two given vertices x and y (if one exists) in the same $O(n^{2/3})$ time bound. Furthermore, insertions of new vertices and deletions of disconnected vertices can be supported in the same time bound in the worst case. However, the choice of the integer k depends now on the number n of current vertices, which may change due to insertions and deletions of disconnected vertices. We circumvent this by recomputing the topological partition of G every time n gets either twice larger or twice smaller than before. Since recomputing the topological partition as well as the full and the abridged representations of the new clusters can be done in linear time, this can be amortized against the number of vertices inserted and deleted after the last time we recomputed the topological partition. As a result, the times for our operations become amortized. Other standard techniques (as the ones shown in Chapter 3) allow us to make these times worst-case.

We remark that our algorithm does not need to recompute the possible change of planar embedding induced by the insertion of new edges. Indeed, we do not even know of any algorithm that checks in $o(n)$ time per operation whether a graph remains planar during sequences of both edge insertions and deletions.

Finally we observe that the $O(n^{2/3})$ bound given in Theorem 10 applies to a larger family of graphs, that is sparse graphs whose sparseness is preserved under contractions of edges. Unlike planarity, this property can be easily checked throughout any sequence of operations.

and abridged representations of the clusters involved in a *Delete* operation is $O(k^2)$.

Theorem 9 *Our data structure supports each* Insert, Delete *and* SameBlock *operation in* $O(n^{2/3})$ *worst-case time on a connected planar graph, where* n *is the number of vertices in the graph.*

Proof: As shown before, each *Insert* and *Delete* operation can be performed in $O(k^2)$ time, while *SameBlock* requires $O(k + \frac{n}{k})$ time. Choosing $k = \lceil n^{1/3} \rceil$ gives the claimed bounds. □

4.4 Dealing with Unconnected Planar Graphs

We show how to deal with unconnected planar graphs within the same time bounds. Let G be an unconnected planar graph. If G consists of $q \geq 2$ connected components, we augment G by inserting $q - 1$ *dummy* edges which make it connected, such that there is no cycle containing two dummy edges. Notice that the size of the augmented graph is still $O(n)$. Furthermore, this transformation does not change the biconnected components of G, and two vertices x and y are in the same biconnected component in the augmented graph if and only if they are in the same biconnected component in the original graph. Furthermore, we assign cost 1 to each dashed edge of G, cost 2 to each solid edge and cost 3 to each dummy edge, and we maintain a topological partition of a minimum spanning tree of the augmented graph.

To insert an edge $e = (x, y)$, we first check whether there is already a dummy edge between x and y. If so, we decrease its cost from 3 to 2. Otherwise, we insert the edge (x, y) of cost 2 and update the minimum spanning tree in $O(\sqrt{n}\,)$ time using the algorithm by Frederickson [18]. If (x, y) enters the minimum spanning tree T and a dummy edge e' leaves T, we delete e'. This can be accomplished within the $O(n^{2/3})$ time bound.

To delete an edge $e = (x, y)$, we first check whether e is in the minimum spanning tree T. If e is not in T, we simply delete e. If e is in T, we check by using the algorithm by Frederickson [18] whether there is a replacement edge for e. If there is such a replacement, we simply delete e. Otherwise, removing e will disconnected the augmented graph and therefore we make e dummy by increasing its cost from 2 to 3. Once again, this does not change the $O(n^{2/3})$ time bound.

This gives us the following theorem:

Theorem 10 *Our data structure supports each* Insert, Delete *and* SameBlock *operation in* $O(n^{2/3})$ *time in the worst case on unconnected planar graphs, where* n *is the total number of vertices.*

new abridged representation of $\mathcal{C}$ can be computed in $O(k)$ time as shown in Lemma 23. However, because of the insertion of extra vertices to keep a degree no greater than three, it may happen that the size of $\mathcal{C}$ has now become greater than $3k-2$. In this case, the cluster $\mathcal{C}$ is split into two clusters $\mathcal{C}'$ and $\mathcal{C}''$. Since $\mathcal{C}$ has size $O(k)$, this splitting may be performed in $O(k)$ time. Furthermore, the full and abridged representations of the new clusters $\mathcal{C}'$ and $\mathcal{C}''$ can be computed in $O(k)$. However, because of this splitting, the full and abridged representations of all the clusters previously adjacent to $\mathcal{C}$ might change. Since there are at most $O(k)$ edges incident to a cluster, at most $O(k)$ clusters can be adjacent to $\mathcal{C}$. Consequently, the full and abridged representations of the clusters that were previously adjacent to $\mathcal{C}$ can be computed in a total of $O(k^2)$ time.

If $\mathcal{C}_{x'} \neq \mathcal{C}_{y'}$, then we have to insert an inter-cluster edge (x,y). Besides the possible updates due to the transformation to keep the vertex degrees no greater than three, we have now to recompute the full and abridged representations of clusters $\mathcal{C}_{x'}$ and $\mathcal{C}_{y'}$. By Lemma 23, this can be done in $O(k)$ time. If the edge (x,y) caused any cluster size to exceed $3k-2$, then we split the cluster as described above.

The total time required to perform an *Insert* operation is therefore $O(k^2)$.

Delete(x,y)

Let (x',y') be the edge of G' corresponding to edge (x,y) in G. Again we denote by $\mathcal{C}_{x'}$ and $\mathcal{C}_{y'}$ the clusters containing respectively x' and y'. When the edge (x,y) is being deleted, we may have to apply the reverse of the transformation used to keep the vertex degrees no greater than three. This can be easily taken care of, since it deletes only a constant number of vertices and edges and it involves only a constant number of clusters. If because of this either $\mathcal{C}_{x'}$ or $\mathcal{C}_{y'}$ has size less than k, then it is merged with an adjacent cluster. If this adjacent cluster has now more than $3k-2$ vertices, it must be split in two clusters. Merging two clusters $\mathcal{C}'$ and $\mathcal{C}''$ into one requires updating the full and abridged representations of all the clusters previously adjacent to $\mathcal{C}'$ and $\mathcal{C}''$. As shown before for the case of cluster splits, this can be done in $O(k^2)$ time. Therefore the total time required to determine whatever splits and merges are needed as well as to perform them along with the re-organization of the full and abridged representation of the clusters involved is $O(k^2)$.

If $\mathcal{C}_{x'} \neq \mathcal{C}_{y'}$, then we have to recompute the full and abridged representations of the two clusters. As said before, this can be done in $O(k)$ time. If $\mathcal{C}_{x'} = \mathcal{C}_{y'} = \mathcal{C}$, then we have the following two cases. If the deletion of (x,y) leaves the cluster $\mathcal{C}$ connected, then we have the compute full and abridged representation of cluster $\mathcal{C}$. This can be done in $O(k)$ time by lemma 23. Otherwise, the deletion of (x,y) disconnects $\mathcal{C}$ and $\mathcal{C}$ can no longer be a cluster. Denote by $\mathcal{C}'$ and $\mathcal{C}''$ the two connected components into which the deletion of (x,y) splits cluster $\mathcal{C}$. As shown before, this splitting may be performed in $O(k^2)$ time. If either $\mathcal{C}'$ or $\mathcal{C}''$ has size less than k, then we merge it with an adjacent cluster. If this adjacent cluster has now more than $3k-2$ vertices, it must be split in two clusters. It follows that the total time required to determine whatever splits and merges are needed as well as to perform them along with the re-organization of the full

the planar graph into $O(\frac{n}{k})$ clusters such that each cluster contains a connected subgraph having between k and $3k-2$ vertices. Furthermore, we maintain for each cluster in the partition both its full and abridged representation. The following lemma is immediate.

Lemma 28 *Any cluster $\mathcal{C}$ can be replaced with its full representation in $O(k)$ time, and with its abridged representation in $O(d(\mathcal{C}))$ time.*

4.3 Implementing the Operations

We now show how to perform *SameBlock*(x,y), *Insert*(x,y), and *Delete*(x,y) operations. As explained earlier, each vertex of G corresponds to vertices of G' that can possibly be in different clusters. In order to execute any operation involving vertices x and y in G, we arbitrarily choose in G' vertices x' and y', corresponding to x and y respectively. This arbitrary choice is possible because all the vertices x' of G' corresponding to vertex x in G are merged into the same vertex during the contraction of dashed edges in the abridged graph.

SameBlock(x,y)

In order to test whether two vertices belong to the same biconnected component, we first locate the two cluster $\mathcal{C}_{x'}$ and $\mathcal{C}_{y'}$ containing respectively x' and y'. We then compute $\mathcal{G}_{x,y}(G)$, the abridged graph induced by $\{x,y\}$. Replacing $\mathcal{C}_{x'}$ and $\mathcal{C}_{y'}$ with their full representation requires $O(k)$ time and replacing the remaining clusters with their abridged representations requires a total of $O(\frac{n}{k})$ time because of Lemma 28. Once $\mathcal{G}_{x,y}(G)$ has been computed in $O(k+\frac{n}{k})$ time, we check whether there is a square node that is an articulation point separating x and y in $\mathcal{G}_{x,y}(G)$. If at least one such articulation point is found, we return *false*. Otherwise we return *true*. This gives us a correct answer because of Lemma 27. By Lemma 26, it can be accomplished in a total of $O(k+\frac{n}{k})$ time by using an off-line algorithm for finding the biconnected components of $\mathcal{G}_{x,y}(G)$ [50].

Insert(x,y)

When the new edge (x,y) is inserted, then the degrees of x and y in the original graph increase by one. If either degree becomes four, then the transformation given earlier has to be applied. If the degree becomes larger than four then the transformation has been already applied but now must be updated. In both cases, this introduces at most a constant number of extra vertices and edges and can be easily handled. We now distinguish two cases depending on whether (x,y) is an internal edge or an inter-cluster edge. As usual we denote by $\mathcal{C}_{x'}$ and $\mathcal{C}_{y'}$ the clusters containing x' and y' respectively.

If $\mathcal{C}_{x'} = \mathcal{C}_{y'} = \mathcal{C}$, then (x,y) is an internal edge. As a result, the insertion of (x,y) changes the interior of $\mathcal{C}$, that is the full and the abridged representation of $\mathcal{C}$. The change in the full representation is simply the new edge (x,y). On the other hand, the

(i) there is an articulation point separating x and y in G if and only if there is a square node that is an articulation point separating x and y in $\widehat{G}$.

We then complete the proof by showing that

(ii) there is square node that is an articulation point separating x and y in $\widehat{G}$ if and only if there is at least one square node that is an articulation point separating x and y in $\mathcal{G}_{x,y}(G)$.

$\widehat{G}$ can be obtained from G by first making all the vertices of G square nodes and then by repeatedly applying the following transformation.

(T1) Biconnected subgraph replacement: Given a biconnected subgraph $\widetilde{G}$ of G, remove all the edges in $\widetilde{G}$ and introduce a round node ρ. Connect to ρ all the vertices in $\widetilde{G}$.

If there is an articulation point separating x and y in G, then (T1) preserves this articulation point. It is possible to re-obtain G from $\widehat{G}$ by means of the inverse transformation of (T1), which again preserve articulation points. This proves (i).

To prove (ii) we notice that $\mathcal{G}_{x,y}(G)$ can be obtained from $\widehat{G}$ by means of a sequence of the following two transformations.

(T2) Leaf elimination: Given a (round or square) node v ($v \neq x$ and $v \neq y$) of degree one, delete v together with its incident edge.

(T3) Round node contraction: Given a round node ρ of degree two that is not adjacent to x and to y, denote by u and v the two square nodes adjacent to ρ. Delete round node ρ and merge u and v into a new square node (i.e., all the edges previously incident to u and v are now incident to the new node).

Indeed transformations (T2) and (T3) correspond to the operations performed while computing the abridged representation of a cluster $\mathcal{C}$ starting from its full representation: (T2) corresponds to rule (R1), and (T3) to rule (R2). If there is a square node that is an articulation point separating x and y in $\widehat{G}$, then transformations (T2) and (T3) preserve at least one square node that is an articulation point separating x and y. Indeed during (T2) v cannot be an articulation point separating x and y, while in (T3) if u is an articulation point separating x and y, also v must be an articulation point separating x and y and therefore the new square node will be an articulation point separating x and y too. Similarly, the inverse transformations of (T2) and (T3) preserve articulation points separating x and y from $\mathcal{G}_{x,y}(G)$ to $\widehat{G}$. □

We now describe our data structure. During any sequence of operations, we maintain a topological partition of order k of the underlying planar graph. That is, we partition

□

Given any vertex x' in G', let us denote by $\mathcal{C}_{x'}$ the cluster containing x'. Notice that, because of the transformation from G to G', a vertex of G may have the corresponding vertices in G' in different clusters. We define a graph $\mathcal{G}_{x,y}(G)$, referred to as the *abridged graph induced by the pair* $\{x, y\}$, by arbitrarily choosing in G' vertices x' and y', corresponding to x and y respectively, and by replacing $\mathcal{C}_{x'}$ and $\mathcal{C}_{y'}$ with their full representations, and each other non-singleton cluster with its abridged representation. The rules for combining full and abridged representations as well as two full representations through solid and dashed edges are the same as described above. Notice that all the dashed edges of G' have been contracted in $\mathcal{G}_{x,y}(G)$. Therefore, a vertex of G corresponds to at most one square node in $\mathcal{G}_{x,y}(G)$. In particular, there is exactly one square node x and one square node y in $\mathcal{G}_{x,y}(G)$. We now characterize some properties of $\mathcal{G}_{x,y}(G)$.

Lemma 26 *Given any two vertices x and y, $\mathcal{G}_{x,y}(G)$, the abridged graph induced by the pair $\{x, y\}$, has at most $O(k + \frac{n}{k})$ vertices and edges and can be computed in $O(k + \frac{n}{k})$ time.*

Proof: The size of $\mathcal{G}_{x,y}(G)$ is at most $O(k + \frac{n}{k})$ because of both Lemma 25 and the fact that the full representation of each cluster has at most $O(k)$ vertices and edges.

The time required to compute the abridged graph induced by x and y can be computed as follows. Since the abridged representation of clusters is available in the adjacency list form, combining the abridged representation of two adjacent clusters $\mathcal{C}'$ and $\mathcal{C}''$ through either a solid or a dashed edge can be done in constant time. As a consequence, the total time spent while combining the abridged representations of adjacent clusters is

$$\sum_{\mathcal{C}} d(\mathcal{C}) = O\left(\frac{n}{k}\right).$$

Since combining the full representation of clusters $\mathcal{C}_{x'}$ and $\mathcal{C}_{y'}$ with the rest of the achieved abridged graph involves examining at most $O(k)$ solid and dashed edges, the total time required to compute $\mathcal{G}_{x,y}(G)$ becomes $O(k + \frac{n}{k})$. □

The abridged graph induced by the pair $\{x, y\}$ satisfies nice properties with respect to the biconnected components of G, as the following lemma shows.

Lemma 27 *There is an articulation point separating x and y in G if and only if there is at least one square node that is an articulation point separating x and y in $\mathcal{G}_{x,y}(G)$.*

Proof: Define $\widehat{G}$ to be the graph obtained from G by replacing each cluster with its full representation and by merging adjacent clusters through solid and dashed edges as explained earlier. Notice that there are two square nodes x and y in both $\widehat{G}$ and $\mathcal{G}_{x,y}(G)$. We first show that

Figure 4.5 (continued)

Figure 4.5: Obtaining the abridged graph of a graph

Figure 4.4: Combining clusters through solid and dashed edges

Proof: First notice that since $\mathcal{C}'$ and $\mathcal{C}''$ are non-singleton clusters, ρ' and ρ'' are always defined. Furthermore, since each cluster is connected, cluster node $\mathcal{C}'$ cannot be an articulation point in $\mathcal{G}'(\mathcal{C}'')$. The same holds for cluster node $\mathcal{C}''$ in $\mathcal{G}'(\mathcal{C}')$. Consequently, ρ' and ρ'' are unique. Let e be an inter-cluster edge whose endpoints are in super-blocks ρ' and ρ'' and assume by contradiction that there is another inter-cluster edge $e' \neq e$ whose endpoints are in ρ_1 and ρ_2, with ρ_1 and ρ_2 being super-blocks in $\mathcal{C}'$ and $\mathcal{C}''$ respectively, and either $\rho_1 \neq \rho'$ or $\rho_2 \neq \rho''$. Since the clusters $\mathcal{C}'$ and $\mathcal{C}''$ are connected, then there is a simple cycle containing both e and e'. As a result, these two edges are in a same biconnected component of both $\mathcal{G}'(\mathcal{C}')$ and $\mathcal{G}'(\mathcal{C}'')$. Therefore, we must have $\rho_1 = \rho'$ and $\rho_2 = \rho''$. □

The abridged representations of all clusters can be combined together to define a new graph $\mathcal{G}(G)$, referred to as the *abridged graph* of G, as follows. For each cluster $\mathcal{C}$, $\mathcal{G}(G)$ contains the vertices and edges of the abridged representation which are internal to $\mathcal{C}$. We handle the inter-cluster edges as follows. Consider any two adjacent non-singleton clusters $\mathcal{C}'$ and $\mathcal{C}''$: we combine them as follows. If there is a cluster node $\mathcal{C}''$ in the abridged representation of $\mathcal{C}'$ (and consequently a cluster node $\mathcal{C}'$ in the abridged representation of $\mathcal{C}''$), as in Lemma 24 denote by ρ' (respectively ρ'') the super-block of $\mathcal{C}'$ (respectively $\mathcal{C}''$) containing cluster node $\mathcal{C}''$ (respectively $\mathcal{C}'$). Then we combine the two abridged representations by merging ρ' and ρ'' into one new round node ρ. The edges previously incident to ρ' and ρ'' are now incident to ρ. Furthermore, the two cluster nodes $\mathcal{C}'$ and $\mathcal{C}''$ are deleted. Lemma 24 justifies the correctness of this operation. We refer to this operation as combining non-singleton clusters $\mathcal{C}'$ and $\mathcal{C}''$ *through a solid edge* (see Figure 4.4a).

Finally, if there is a dashed edge e between $\mathcal{C}'$ and $\mathcal{C}''$, then there is a dashed vertex e in both the abridged representations of $\mathcal{C}'$ and $\mathcal{C}''$. In this case, let us denote by v' and v'' the two square nodes in the abridged representations of $\mathcal{C}'$ and $\mathcal{C}''$ at distance two from e. We delete the dashed vertices e as well as their adjacent round nodes in the abridged representations of $\mathcal{C}'$ and $\mathcal{C}''$. The two abridged representations are now combined by merging v' and v'' into one new square node v, which is adjacent to the previous nodes v' and v'' were adjacent to. We refer to this operation as combining non-singleton clusters $\mathcal{C}'$ and $\mathcal{C}''$ *through a dashed edge* (see Figure 4.4b).

By applying these operations to all the non-singleton clusters, we obtain the *abridged graph* $\mathcal{G}(G)$. Figure 4.5 shows the abridged graph of a given graph.

Lemma 25 *The abridged graph $\mathcal{G}(G)$ of a graph G has at most $O(\frac{n}{k})$ vertices and edges.*

Proof: As usual, let $d(\mathcal{C})$ denote the number of different clusters adjacent to cluster $\mathcal{C}$. By Lemma 23, the abridged representation of $\mathcal{C}$ contains $O(d(\mathcal{C}))$ vertices and edges. Since combining the abridged representations of adjacent clusters does not introduce new vertices and edges, the total number of vertices and edges in $\mathcal{G}(G)$ can be at most

$$\sum_{\mathcal{C}} O(d(\mathcal{C})) = O\left(\frac{n}{k}\right).$$

Figure 4.3: Full and abridged representation of a cluster

square node (i.e., all the edges previously incident to u and v are now incident to the new node).

The resulting tree $\mathcal{T}'$ is called the *abridged representation* of cluster $\mathcal{C}$. The round nodes of $\mathcal{T}'$ are called *super-blocks of* $\mathcal{C}$. Figure 4.3 shows a cluster together with its full and abridged representations.

Consider a cluster $\mathcal{C}$ which is adjacent to only one other cluster $\mathcal{C}'$ by means of solid edges. In this case the abridged representation of $\mathcal{C}$ degenerates into only one square node (corresponding to cluster node $\mathcal{C}'$), and $\mathcal{C}$ is said to be a *singleton* cluster.

For any cluster $\mathcal{C}$ let us denote by $d(\mathcal{C})$ the number of different clusters adjacent to $\mathcal{C}$. We refer to $d(\mathcal{C})$ also as the *external degree* of cluster $\mathcal{C}$. The following lemmas characterize the abridged representation of a cluster.

Lemma 23 *The abridged representation of a cluster $\mathcal{C}$ has $O(d(\mathcal{C}))$ vertices and edges and can be computed in $O(k)$ time starting from the full representation of $\mathcal{C}$.*

Proof: Since there are at most $O(d(\mathcal{C}))$ cluster nodes and dashed vertices in the abridged representation of $\mathcal{G}'(\mathcal{C})$, the number of terminal vertices in $\mathcal{T}$ is at most $O(d(\mathcal{C}))$.

We now show that the size of the abridged representation of cluster $\mathcal{C}$ (i.e., the size of the compressed tree $\mathcal{T}'$) is $O(d(\mathcal{C}))$. The rules used to obtain $\mathcal{T}'$ from $\mathcal{T}$ are (R1) deleting a non-terminal leaf, and (R2) contracting a degree two round node adjacent to non-terminal square nodes. Because of rule (R1) all the leaves of $\mathcal{T}'$ are terminal square nodes. Because of rule (R2) there cannot be degree two round nodes adjacent to two non-terminal square nodes in $\mathcal{T}'$. Consequently, the number of round nodes in $\mathcal{T}'$ is no greater than the number of terminal square nodes. This implies that $|\mathcal{T}'|$ is linear in the number of terminal nodes in $\mathcal{T}$, and therefore $|\mathcal{T}'| = O(d(\mathcal{C}))$.

We now prove that the time required to compute the abridged representation of a cluster $\mathcal{C}$ is $O(k)$. Indeed the abridged representation of cluster $\mathcal{C}$ is computed starting from $\mathcal{T}$, which is of size $O(k)$. Finding the terminal and non-terminal nodes in $\mathcal{T}$ can be done in $O(k)$ by simply visiting $\mathcal{T}$. By representing $\mathcal{T}$ by means of adjacency lists (such a representation can be constructed in $O(k)$ time), any sequence of the two rules used to obtain $\mathcal{T}'$ can be implemented in a total of $O(k)$ time. □

Lemma 24 *Let $\mathcal{C}'$ and $\mathcal{C}''$ be any two adjacent non-singleton clusters in the topological partition. Let ρ' be the super-block in the abridged representation of $\mathcal{C}'$ containing the cluster node $\mathcal{C}''$ (i.e., ρ' is the round node adjacent to cluster node $\mathcal{C}''$ in the abridged representation of $\mathcal{C}'$). Similarly, let ρ'' be the super-block in the abridged representation of $\mathcal{C}''$ containing the cluster node $\mathcal{C}'$. Then all the inter-cluster edges of the graph G' between $\mathcal{C}'$ and $\mathcal{C}''$ are incident to vertices belonging to the super-blocks ρ' and ρ''.*

Figure 4.2: A cluster and its contracted graph

can be accomplished in $O(n)$ time.

Since all the dashed edges will be in T and the intersection of any cluster with T is a tree, there can be be at most $O(\frac{n}{k})$ inter-cluster dashed edges. An edge of G' with both endpoints in a cluster $\mathcal{C}$ is said to be *internal* to $\mathcal{C}$, while an edge of G' with only one endpoint in $\mathcal{C}$ is said to be *incident* to $\mathcal{C}$. For each cluster $\mathcal{C}$, the *subgraph of G' induced by $\mathcal{C}$* is called $\mathcal{G}(\mathcal{C})$ and is defined as follows. $\mathcal{G}(\mathcal{C})$ has one vertex for each cluster in the topological partition adjacent to $\mathcal{C}$ (referred to as a *cluster node*) plus all the vertices in $\mathcal{C}$. The edge set of $\mathcal{G}(\mathcal{C})$ is composed of all the edges of G internal and incident to cluster $\mathcal{C}$. Since $\mathcal{C}$ contains at most $3k-2$ vertices, all of degree no more than three, then $\mathcal{G}(\mathcal{C})$ has at most $O(k)$ vertices and edges. For each cluster $\mathcal{C}$ in the topological partition of T, we maintain a *full representation* and an *abridged representation*, defined as follows.

The *full representation* of a cluster $\mathcal{C}$ is computed as follows. Starting from $\mathcal{G}(\mathcal{C})$, we contract the dashed edges internal to cluster $\mathcal{C}$. As for dashed edges incident to cluster $\mathcal{C}$, in $\mathcal{G}(\mathcal{C})$ they are of the form $(\hat{x},\mathcal{C}')$, with $\hat{x}$ being a vertex internal to $\mathcal{C}$ corresponding to a vertex x in the original graph, and $\mathcal{C}'$ being a cluster different from $\mathcal{C}$. For each such an edge $e=(\hat{x},\mathcal{C}')$ we introduce in its place a new special vertex e, referred to as a *dashed vertex*, and an edge $(\hat{x},e)$. The resulting graph, which we call $\mathcal{G}'(\mathcal{C})$, contains at most $O(k)$ vertices and edges and can be computed in $O(k)$ time. Figure 4.2 shows a graph $\mathcal{G}'(\mathcal{C})$ obtained after these operations.

We then compute the biconnected components of $\mathcal{G}'(\mathcal{C})$, and subsequently construct the block tree $\mathcal{T}$ of $\mathcal{G}'(\mathcal{C})$ (as defined for instance in [23]). We recall here that a block tree of a graph is composed of square nodes (corresponding to vertices in the graph) and round nodes (corresponding to blocks in the graph): whenever a vertex v belongs to a given block, there is a tree edge between the corresponding square and round nodes in the block tree. As a consequence, each path in a block tree alternates between square and round nodes. The construction of the block tree is immediate once the biconnected components of $\mathcal{G}'(\mathcal{C})$ have been found. We define the block tree $\mathcal{T}$ to be the *full representation* of cluster $\mathcal{C}$. Since the size of $\mathcal{G}'(\mathcal{C})$ is $O(k)$, the full representation of $\mathcal{C}$ can be computed in a total of $O(k)$ time.

We define the *abridged representation of a cluster* $\mathcal{C}$ starting from its full representation (block tree) $\mathcal{T}$ as follows. We define the cluster nodes, the dashed vertices and the square nodes at distance two from dashed vertices to be *terminal nodes* in $\mathcal{T}$. All the remaining nodes of $\mathcal{T}$ are defined *non-terminal*. With this definition, round nodes are always non-terminal. Notice that by definition of dashed vertex, for each dashed vertex v in the full representation of $\mathcal{C}$ there is exactly one square node at distance two from v. We compute a succinct representation $\mathcal{T}'$ of the block tree $\mathcal{T}$ by applying any sequence of the following two rules:

(R1) delete any non-terminal leaf (either square or round node);

(R2) if there is any degree-two round node ρ whose two adjacent square nodes, say u and v, are both non-terminal, then delete round node ρ and merge u and v into a new

As mentioned in [57] the fully dynamic maintenance of the connected components of a graph differs substantially from the fully dynamic maintenance of the biconnected components. Indeed in the first problem a single edge insertion can merge at most two components into one. On the other hand, for the second problem a single edge insertion can produce a cycle in the graph that might combine as many as $\Theta(n)$ biconnected components into one. As a result, by simply alternating the insertion and deletion of the same edge, we can generate a sequence of operations that at each step changes the number of components by $\Theta(n)$. This also explains why we do not maintain biconnected components' names: indeed in such a scenario a *Delete* operation would have to specify as many as $\Theta(n)$ different new names.

The remainder of this chapter consists of four sections. Section 2 presents a data structure for connected planar graphs. Section 3 describes the implementation of the operations and analyzes their time complexity. Section 4 deals with unconnected planar graphs, while Section 5 lists some concluding remarks.

4.2 A Data Structure for Connected Planar Graphs

In this section we describe a data structure for maintaining the biconnected components of an undirected planar graph G under an intermixed sequence of *SameBlock*, *Insert*, and *Delete* operations. For sake of simplicity, we start by describing algorithms dealing with connected planar graphs. We will show later how to extend the same algorithms to unconnected planar graphs with the same time bounds.

We describe a data structure that operates on planar graphs with bounded degree. It is well known (see for example [23, page 132]) that each graph $G = (V, E)$ can be transformed into a graph $G' = (V', E')$, whose vertices have degree no greater than three. We use the following transformation. For each vertex u of G of degree $d \geq 4$, where $v_0, v_1, \ldots, v_{d-1}$ are the vertices adjacent to u, we replace u with new vertices $u_0, u_1, \ldots, u_{d-1}$ and replace edge (u, v_i) with a *solid* edge (u_i, v_i), $0 \leq i \leq d-1$. Furthermore, we add new *dashed* edges $(u_i, u_{(i+1)})$, $0 \leq i \leq d-2$. We associate costs with edges in G' as follows: solid edges have cost 2 and dashed edges have cost 1. Using this transformation, each edge (x, y) in G corresponds to a unique solid edge (x_i, y_j) in G' and vice versa. However, this property does not hold for dashed edges. Another property of this transformation is that all the dashed edges are contained in any minimum spanning tree of G'. It is easy to show that given a graph G with m edges and n vertices, G' contains at most $O(m+n)$ edges and vertices and can be generated in $O(m+n)$ time. Furthermore if G is planar, $m = O(n)$ and G' is planar.

Let k be a positive integer to be chosen later. We partition the graph G' into vertex clusters having between k and $3k-2$ vertices, by using a topological partition of order k of a minimum spanning tree T of G', as defined in [18]. We recall that a topological partition of T of order k consists of $O(\frac{n}{k})$ vertex clusters of T such that each cluster contains a connected subgraph with between k and $3k-2$ vertices. As shown in [18], this

Figure 4.1: A graph and its biconnected components

edge belongs exactly to one biconnected component. Figure 4.1 shows a graph together with its biconnected components.

As said in Chapter 1, the problems of computing the biconnected components of undirected graphs arise naturally in many applications and have been extensively studied. Tarjan [50] give optimal linear-time sequential algorithms. Tarjan and Vishkin [55] give logarithmic-time parallel algorithms. Recently, Westbrook and Tarjan [57] considered the problem of maintaining the biconnected components of a graph undergoing a 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 very slowly growing function (a functional inverse of Ackermann's function). As usual, we will refer to a problem in which only insertions are defined as a *partially dynamic* problem, and problems in which both insertions and deletions are allowed are called *fully dynamic*.

In this chapter we are interested in the fully dynamic maintenance of the biconnected components of an undirected planar graph. We would like to perform any sequence of three kinds of operations starting on an initially null graph G (i.e., a graph with n vertices and no edges).

SameBlock(u,v): Return *true* if vertices u and v are in a same biconnected component. Return *false* otherwise.

Insert(x,y): Insert a new edge between the two vertices x and y, provided that the insertion of the new edge leaves the graph planar.

Delete(x,y): Delete the edge between the two vertices x and y.

As mentioned in [57], in such a fully dynamic setting there has been no known solution better than a repeated application of an off-line algorithm, which yields a single operation worst-case time complexity of $O(m)$, where m is the current number of edges in the graph. We show that for planar graphs this problem can be solved in $O(n^{2/3})$ time per operation. We obtain the same bounds even if we are initially given a non-null graph $G_0 = (V_0, E_0)$ and we allow $O(|V_0| + |E_0|)$ preprocessing time. In addition our data structure is able to return an articulation point separating two given vertices x and y (if one exists) in the same $O(n^{2/3})$ time bound. Furthermore, insertions of new vertices and deletions of disconnected vertices can be supported in the same time bound. This is the first sublinear-time algorithm known for this problem, and the case of general (non-planar) graphs remains open. The case of planar graphs, however, seems to be important because a fully dynamic algorithm for maintaining the biconnected components constitutes a first step towards the study of fully dynamic planarity testing.

The gap between the times required to solve the partially and the fully dynamic version of these problems is not surprising. Indeed, as mentioned in Chapter 1, fully dynamic graph problems are usually much more difficult to solve than the corresponding partially dynamic problems.

Chapter 4

Maintaining Biconnected Components of Dynamic Planar Graphs

In this chapter we present algorithms for maintaining the biconnected components of a planar graph undergoing repeated dynamic modifications, such as insertions and deletions of edges and vertices. We show how to test at any time whether two vertices belong to the same biconnected component, and how to insert and delete an edge in $O(n^{2/3})$ time in the worst case, where n is the number of vertices in the graph. The data structure supports also insertions of new vertices and deletions of disconnected vertices in the same time bounds. This is the first sublinear-time algorithm known for this problem.

4.1 Introduction

As already mentioned in Chapter 3, an undirected graph $G = (V, E)$ is said to be *connected* if there is a path from each vertex to every other vertex, and *biconnected* if there is no vertex whose removal disconnects G: such a vertex is called an *articulation point*. It is well known [2] that a graph G is biconnected if and only if there are at least two vertex disjoint paths from each vertex to every other vertex. Given a graph $G = (V, E)$, let $E_1, E_2, \ldots, E_h$ be the partition of E into equivalence classes such that two edges e' and e'' are in the same equivalence class if and only if either (i) $e' = e''$ or (ii) there is a simple cycle (i.e., a cycle with no repetition of edges and vertices) of G containing both e' and e''. For $1 \leq i \leq h$ let V_i be the set of vertices that are endpoints of the edges in E_i. The subgraphs $G_i = (V_i, E_i)$ are called the *biconnected components* (or *blocks*) of G. An articulation point appears in more than one biconnected component. The set of edges of every graph can be partitioned into biconnected components in a unique way and every

clusters that need to be adjusted, the adjustments may be accomplished before a new round of adjustments is initiated. Thus every time an insertion occurs, the clusters can be scanned to find any cluster that is too small and a constant number of these clusters can be combined with a neighbor if needed. Similar operations are performed during an edge deletion. Clearly, the same argument applies to planar graphs. □

3.6 Dealing with Unconnected Graphs

We end this section by showing how to deal with unconnected graphs within the same time bounds. Let G be an unconnected graph. If G consists of q connected components, we augment G by inserting $q-1$ *dummy* edges which make it connected, such that there is no cycle containing two dummy edges. This does not change the 2-edge-connected components of G, and two vertices x and y are in the same 2-edge-connected component in the augmented graph if and only if they are in the same 2-edge-connected component in the original graph. Furthermore, we assign cost 1 to each edge of G and cost 2 to each dummy edge, and we maintain a topological partition of a minimum spanning tree of the augmented graph.

To insert an edge $e=(x,y)$, we first check whether there is already a dummy edge between x and y. If so, we decrease its cost from 2 to 1. Otherwise, we insert the edge (x,y) of cost 1 and update the minimum spanning tree in $O(\sqrt{m}\,)$ time using the algorithm by Frederickson [18]. If (x,y) enters the minimum spanning tree T and a dummy edge e' leaves T, we delete e'. This can be accomplished within the $O(m^{2/3})$ and $O(\sqrt{n\log\log n}\,)$ time bounds.

To delete an edge $e=(x,y)$, we first check whether e is in the minimum spanning tree T. If e is not in T, we simply delete e. If e is in T, we check by using the algorithm by Frederickson [18] whether there is a replacement edge for e. If there is such a replacement, we simply delete e. Otherwise, removing e will disconnected the augmented graph and therefore we make e dummy by increasing its cost from 1 to 2. Once again, this does not change the $O(m^{2/3})$ and $O(\sqrt{n\log\log n}\,)$ time bounds. This gives us the following theorem:

Theorem 8 *The data structure above supports each* InsertEdge, DeleteEdge *and* Same2EdgeBlock *operation in $O(m^{2/3})$ time in the worst case on a general graph, where m is the current number of edges. For planar graphs, this time reduces to $O(\sqrt{n\log\log n}\,)$.*

We notice that the $O(\sqrt{n\log\log n}\,)$ bound applies to a larger family of graphs, that is sparse graphs whose sparseness is preserved under contractions of edges. This property can be checked throughout any sequence of operations. However, we do not know of any algorithm which checks in $o(n)$ time per operation whether a graph remains planar during sequences of edge insertions and deletions.

i.e., integers in the range $[1, 3k-2]$. Similarly, the vertices $\gamma'_1, \gamma'_2, \ldots, \gamma'_r$ have to be sorted according to their preorder numbers, again integers in the range $[1, 3k-2]$. We exploit this by using Johnson's stratified trees [34] instead of balanced search trees to represent the sets $E(\mathcal{C}_i, \mathcal{C}_j)$. Each range query can now be performed more efficiently in $O(\log\log k)$ time, at the the expense of more space usage, that increases from $O(|E(\mathcal{C}_i, \mathcal{C}_j)|)$ to $O(k)$. The initialization of a Johnson's stratified tree containing $O(k)$ items can be still accomplished in $O(k)$ time [37], which implies that the time required to initialize the data structures pertinent to a cluster (as analyzed in Lemma 9) is still $O(k)$. Furthermore, the vertices $\gamma'_1, \gamma'_2, \ldots, \gamma'_r$ can be sorted more efficiently in $O(d(\mathcal{C})\log\log k)$ time at the expense of $O(k)$ space. Consequently, using stratified trees leads to a more efficient implementation of *Same2EdgeBlock* in time $O(k + \frac{n}{k} + \frac{n}{k}\log\log k) = O(k + \frac{n}{k}\log\log k)$. Following exactly the same argument given earlier, it can be shown that during either an *InsertEdge* or a *DeleteEdge* operation the time required to update a cluster, or to merge two clusters into a new cluster, or to split a cluster into two is still $O(k)$. As a result, the time required to perform *InsertEdge* and *DeleteEdge* operations is again $O(k)$. Choosing $k = \lceil \sqrt{n \log\log n} \rceil$, gives an $O(\sqrt{n\log\log n})$ time bound per operation in case of planar graphs.

The total space required by this data structure is $O(n)$ (as shown in Theorem 5) plus the total space required by all the stratified trees. We now bound the total number of stratified trees needed. Consider the super-graph $\widetilde{G}$ obtained (i) by contracting each cluster into a super-vertex, and (ii) by reducing to two the number of multiple edges between any two super-vertices. Notice that the number of edges in $\widetilde{G}$ is an upper bound on the total number of stratified trees needed. But since the original graph G is planar, and contraction preserves planarity, $\widetilde{G}$ is itself planar. Because $\widetilde{G}$ has $O(\frac{n}{k})$ vertices, it has at most $O(\frac{n}{k})$ edges. Therefore we have at most $O(\frac{n}{k})$ stratified trees in our data structure, each of which requires $O(k)$ space. Consequently, the extra space needed is $O(\frac{n}{k}k) = O(n)$. □

Whenever needed, the times given in Theorems 5 and 6 can be made worst-case as the following theorem shows.

Theorem 7 *The data structure above supports each* InsertEdge, DeleteEdge *and* Same2EdgeBlock *operation in $O(m^{2/3})$ time in the worst case, where m is the current number of edges in the connected graph. For planar connected graphs, this time reduces to $O(\sqrt{n\log\log n})$.*

Proof: Let m_t be the number of edges in the graph at time t. We claim that an update at time t for general graphs can be carried out in $O(m_t^{2/3})$ worst-case time. This can be achieved as follows. Let $k_t = \left\lceil m_t^{2/3} \right\rceil$. When the value of k changes due to an *InsertEdge* or *DeleteEdge* operation, there will be at least $\frac{1}{2}\left\lceil m_t^{2/3} \right\rceil$ more updates before k becomes twice larger or twice smaller than before. The idea is to adjust a constant number of clusters each time there is an update. This sums up to a total of $O(m_t^{2/3})$ cluster adjustments. Since there will be no more than

$$\frac{m_t}{k_t} \leq \left\lceil m_t^{1/3} \right\rceil \leq O(m_t^{2/3})$$

3.5 Time Complexity

Our implementation of the *Same2EdgeBlock*, *InsertEdge* and *DeleteEdge* operations achieve the following bounds.

Theorem 5 *The data structure above supports each* InsertEdge, DeleteEdge *and* Same2EdgeBlock *operation in $O(m^{2/3})$ amortized time on a connected graph, where m is the current number of edges in the graph. For planar connected graphs, this time reduces to $O(\sqrt{n \log n})$. The total space required by the data structure is $O(m)$.*

Proof: By Lemmas 18, 20 and 22, each *InsertEdge* and *DeleteEdge* operation can be performed in $O(k)$ time, while *Same2EdgeBlock* requires $O(k + \frac{m^2}{k^2})$ time. Choosing $k = \left\lceil m^{2/3} \right\rceil$ gives the claimed bounds. However, this choice of k depends on the number m of current edges, which may change due to *InsertEdge* and *DeleteEdge* operations. We circumvent this by recomputing the topological partition of G every time m gets either twice larger or twice smaller than before. Since recomputing the topological partition as well as the full and the tree representations and the balanced search trees of the new clusters can be done in linear time, this can be amortized against the number of edges inserted and deleted after the last time we recomputed the topological partition. As a result, the times for *InsertEdge*, *DeleteEdge* and *Same2EdgeBlock* become amortized.

If the underlying graph G is planar and insertions of new edges leave it planar, the time for *Same2EdgeBlock* reduces to $O(k + \frac{n}{k} \log k)$ as shown in Lemma 18. Choosing $k = \lceil \sqrt{n \log n} \rceil$ gives the $O(\sqrt{n \log n})$ amortized bound for planar graphs.

The total space complexity of our data structure can be analyzed as follows. Throughout the sequence of operations, we maintain the super-graph $\widetilde{G}$, which is of size $O(\frac{m^2}{k^2}) = O(m^{2/3})$. Furthermore, by Lemma 9 each cluster $\mathcal{C}$ requires at most $O(k)$ space to store its pertinent data structures. Since there can be at most $O(\frac{m}{k})$ clusters, the total space required by our data structure is $O(m)$. □

The bound for planar graphs can be further reduced, as the following theorem shows.

Theorem 6 *Each* InsertEdge, DeleteEdge *and* Same2EdgeBlock *operation on planar graphs can be supported in $O(\sqrt{n \log \log n})$ amortized time, provided that* InsertEdge *operations leave the graph planar. The space required is $O(n)$.*

Proof: The bottleneck of our algorithm for planar graphs is how to check whether there is a bridge inside a cluster. Indeed, we accomplished this task in a total of $O(\frac{n}{k} \log k)$. The more time consuming tasks were performing range queries in the balanced search trees $E(\mathcal{C}_i, \mathcal{C}_j)$ and sorting the vertices $\gamma'_1, \gamma'_2, \ldots, \gamma'_r$ (as shown in Lemma 17). Notice that for each pair of clusters $\mathcal{C}_i$ and $\mathcal{C}_j$, the set $E(\mathcal{C}_i, \mathcal{C}_j)$ contains preorder numbers in $\mathcal{T}(\mathcal{C}_j)$,

All the data structures pertinent to the new cluster $\mathcal{C}$ can be recomputed from scratch in $O(k)$ by Lemma 9. The other update needed is to compute the balanced search trees $E(\mathcal{C},\mathcal{C}_i)$ for any cluster $\mathcal{C}_i$ adjacent to $\mathcal{C}$. We compute $E(\mathcal{C},\mathcal{C}_i)$ by merging the two old balanced search trees $E(\mathcal{C}',\mathcal{C}_i)$ and $E(\mathcal{C}'',\mathcal{C}_i)$, inserting one item at a time into $E(\mathcal{C},\mathcal{C}_i)$ according to increasing preorder numbers. Because of this, items will always be inserted at the end of $E(\mathcal{C},\mathcal{C}_i)$ and therefore this balanced search tree can be computed in a total of $O(|E(\mathcal{C},\mathcal{C}_i)|)$ time. Since we can have at most $O(k)$ edges incident to cluster $\mathcal{C}$, the total time required to compute all the balanced search trees $E(\mathcal{C},\mathcal{C}_i)$ for clusters $\mathcal{C}_i$ adjacent to $\mathcal{C}$ is $O(k)$. Finally, because of the cluster merging, we may have to merge two super-vertices of $\widetilde{G}$ into one. But this can be easily performed within the $O(k)$ time bound. □

By Lemmas 19 and 21, the total time required to determine whatever splits and merges are needed as well as to perform them along with the re-organization of the data structures involved is $O(k)$. Let $\mathcal{C}_x$ and $\mathcal{C}_y$ denote the two clusters containing x and y after all these possible splits and merges have been performed.

If $\mathcal{C}_x \neq \mathcal{C}_y$, deleting edge (x,y) requires recomputing the full representations of the two clusters and updating the balanced search trees $E(\mathcal{C}_x,C_y)$ and $E(\mathcal{C}_y,C_x)$. Furthermore, we may have to update the edges between $\mathcal{C}_x$ and $\mathcal{C}_y$ in $\widetilde{G}$. As explained before, all this can be done in a total of $O(k)$ time.

If $\mathcal{C}_x = \mathcal{C}_y = \mathcal{C}$, then we have the following two cases. If the deletion of (x,y) leaves the cluster $\mathcal{C}$ connected, then we need to recompute the full and tree representation of $\mathcal{C}$ as well as the balanced search trees $E(\mathcal{C}',\mathcal{C})$ for each $\mathcal{C}'$ neighbor of $\mathcal{C}$. Once again, this requires $O(k)$ time by Lemma 9. Otherwise if the deletion of edge (x,y) disconnects $\mathcal{C}$, it can no longer be a cluster. Denote by $\mathcal{C}'$ and $\mathcal{C}''$ the two connected components into which the deletion of (x,y) splits cluster $\mathcal{C}$. Since $\mathcal{C}$ has size $O(k)$, this splitting may be performed in a total of $O(k)$ time by Lemma 19. If either $\mathcal{C}'$ or $\mathcal{C}''$ has size less than k, then we merge it with an adjacent cluster. If this adjacent cluster has now more than $3k-2$ vertices, it must be split in two clusters. Note that no further merges and splits are needed. By Lemmas 19 and 21, the total time required to determine whatever splits and merges are needed as well as to perform them along with both the re-organization of the data structures involved and the updates needed in $\widetilde{G}$ is $O(k)$.

In summary, deleting an edge (x,y) causes a constant number of cluster splits and merges, and at most a constant number of cluster updates. By Lemmas 9, 19 and 21 all this can be done in $O(k)$ time. Therefore we have the following lemma.

Lemma 22 *The total time required to update our data structure because of a* DeleteEdge *operation is* $O(k)$.

the clusters containing x and y after the splitting has been performed. In order to perform the actual insertion of edge (x,y), we distinguish two cases depending on whether (x,y) is an edge internal to a cluster or an inter-cluster edge.

If $\mathcal{C}_x = \mathcal{C}_y = \mathcal{C}$, then (x,y) is an internal edge. As a result, the insertion of (x,y) changes the interior of $\mathcal{C}$, that is the full and the tree representation $\mathcal{F}(\mathcal{C})$ and $\mathcal{T}(\mathcal{C})$ of $\mathcal{C}$. The change in $\mathcal{T}(\mathcal{C})$ may induce changes in the balanced search tree $E(\mathcal{C}',\mathcal{C})$, for each cluster $\mathcal{C}'$ neighbor of $\mathcal{C}$. But this implies that only the data structures pertinent to $\mathcal{C}$ need to be updated. By Lemma 9, all these data structures can be recomputed from scratch in $O(k)$ time.

If $\mathcal{C}_x \neq \mathcal{C}_y$, then we have to insert an inter-cluster edge (x,y). The updates needed are in the full representations of clusters $\mathcal{C}_x$ and $\mathcal{C}_y$ and in the two balanced search trees $E(\mathcal{C}_x,C_y)$ and $E(\mathcal{C}_y,\mathcal{C}_x)$. As said before, this can be done in $O(k)$ time by simply reinitializing the data structures pertinent to clusters $\mathcal{C}_x$ and $\mathcal{C}_y$. Furthermore, we may have to update the edges between $\mathcal{C}_x$ and $\mathcal{C}_y$ in $\widetilde{G}$, but this can be easily done within the $O(k)$ time bound.

In summary, inserting an edge (x,y) causes at most two clusters to be split and a constant number of clusters to be updated. By Lemma 19, each cluster can be split in $O(k)$ time, and by Lemma 9 each cluster and its pertinent data structures can be recomputed from scratch in $O(k)$ each time an update is needed. This leads to the following lemma.

Lemma 20 *The total time required to update our data structure because of an* InsertEdge *operation is* $O(k)$.

The deletion of an edge (x,y) can be performed as follows. Again we denote by $\mathcal{C}_x$ and $\mathcal{C}_y$ the clusters containing respectively x and y. When the edge (x,y) is being deleted, we may have to apply the reverse of the transformation used to keep the vertex degrees no greater than three. This can be easily taken care of, since it deletes only a constant number of vertices and edges and it involves only a constant number of clusters. If because of this either $\mathcal{C}_x$ or $\mathcal{C}_y$ has now size less than k, then it is merged with an adjacent cluster. If this adjacent cluster has now size greater than $3k-2$, it must be split into two clusters. Since the cluster resulting from this merging has size at most $(k-1)+(3k-2) < 4k$, this splitting can be carried out in $O(k)$ time because of Lemma 19. Notice that there is no further propagation of merging and splitting.

Merging two clusters involves updates similar to splitting a cluster and can be carried out in $O(k)$ time as the following lemma shows.

Lemma 21 *Let $\mathcal{C}'$ be a cluster whose size is less than k and $\mathcal{C}''$ be a cluster adjacent to $\mathcal{C}'$. Then $\mathcal{C}'$ and $\mathcal{C}''$ can be merged into a new cluster $\mathcal{C}$ in $O(k)$ time.*

Proof: The updates needed are in the data structures pertinent to the newly created cluster $\mathcal{C}$, in all the balanced search trees $E(\mathcal{C},\mathcal{C}_i)$ for any cluster $\mathcal{C}_i$ adjacent to $\mathcal{C}$, and in the super-graph $\widetilde{G}$.

Then $\mathcal{G}_{x,y}(G)$ has $O(k+\frac{n}{k})$ vertices and edges. Therefore, each *Same2EdgeBlock* operation can now be performed in $O(k+\frac{n}{k}+\frac{n}{k}\log k) = O(k+\frac{n}{k}\log k)$ time. □

We now show how to support *InsertEdge*(x,y). As usual we denote by $\mathcal{C}_x$ and $\mathcal{C}_y$ the clusters containing x and y respectively. When the new edge (x,y) is inserted, the degrees of x and y in the original graph increase by one. If either the degree of x or the degree of y becomes four, then the transformation given at the beginning of the section has to be applied. If either degree becomes larger than four then the transformation has been already applied but now must be updated. In both cases, this introduces at most a constant number of extra vertices and edges and can be easily handled. However, because of the insertion of extra vertices to keep a degree no greater than three, it may happen that the size of either $\mathcal{C}_x$ or $\mathcal{C}_y$ has now become greater than $3k-2$. When the size of a cluster $\mathcal{C}$ becomes greater than $3k-2$, we split $\mathcal{C}$ into two clusters $\mathcal{C}'$ and $\mathcal{C}''$.

Lemma 19 *A cluster $\mathcal{C}$ whose size is greater than $3k-2$ but is still smaller than $4k$ can be split into two clusters C' and C'', such that they are connected and of size at least k and at most $3k-2$. The total time required to update all the data structures because of this splitting is $O(k)$.*

Proof: Since $\mathcal{C}$ has size $O(k)$, two clusters $\mathcal{C}'$ and $\mathcal{C}''$ that satisfy the cluster requirements, i.e., they are connected and of size between k and $3k-2$, can be found in $O(k)$ time (see for instance [18]). We now analyze the time needed to update all the data structures because of this splitting. The changes involve the data structures pertinent to the newly created clusters $\mathcal{C}'$ and $\mathcal{C}''$, all the balanced search trees $E(\mathcal{C}',\mathcal{C}_i)$ and $E(\mathcal{C}'',\mathcal{C}_j)$ (for any clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ adjacent respectively to $\mathcal{C}'$ and $\mathcal{C}''$), and the super-graph $\widetilde{G}$.

The full and tree representations of the new clusters $\mathcal{C}'$ and $\mathcal{C}''$ as well as the balanced search trees pertinent to $\mathcal{C}'$ and $\mathcal{C}''$ (i.e., $E(\mathcal{C}_i,\mathcal{C}')$ and $E(\mathcal{C}_j,\mathcal{C}'')$, for any clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ adjacent to $\mathcal{C}'$ and $\mathcal{C}''$ respectively) can be recomputed from scratch in $O(k)$ as explained in Lemma 9. The other update needed is to compute the balanced search trees $E(\mathcal{C}',\mathcal{C}_i)$ and $E(\mathcal{C}'',\mathcal{C}_j)$ for any clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ that are adjacent to $\mathcal{C}'$ and $\mathcal{C}''$. We do this by visiting the old balanced search trees $E(\mathcal{C},\mathcal{C}_h)$ according to increasing preorder numbers. Each visited item is inserted either in $E(\mathcal{C}',\mathcal{C}_h)$ or in $E(\mathcal{C}'',\mathcal{C}_h)$, depending on whether the corresponding edge is incident now to $\mathcal{C}'$ or $\mathcal{C}''$. Because of the preorder numbering, items are always inserted at the end of $E(\mathcal{C}',\mathcal{C}_h)$ and $E(\mathcal{C}'',\mathcal{C}_h)$. As a result, these two balanced search trees can be computed in a total of $O(|E(\mathcal{C},\mathcal{C}_h)|)$ time. The total time for computing all these balanced search trees is therefore still $O(k)$. Because of the splitting, also the super-graph $\widetilde{G}$ must be updated. But this can be easily performed within the $O(k)$ time bound. Consequently, all the updates needed can be carried out in a total of $O(k)$ time. □

When the edge (x,y) has to be inserted, we apply possibly the transformation to keep vertices of degree no more than three, and split either cluster $\mathcal{C}_x$ or cluster $\mathcal{C}_y$ if its size exceeds $3k-2$. As explained above, this can be done in $O(k)$. Let now $\mathcal{C}_x$ and $\mathcal{C}_y$ denote

correct because of Lemma 8. Otherwise, we have still to check whether there is a bridge separating x and y in G that is internal to some cluster $\mathcal{C}$. By Lemma 8, we can restrict our attention to clusters that are articulation points separating x and y in $\mathcal{G}_{x,y}(G)$. We proceed as follows.

Let $\mathcal{S}$ be the set of clusters such that (i) they are articulation points in $\mathcal{G}_{x,y}(G)$, and (ii) they separate x and y. Notice that in the worst case, $\mathcal{S}$ contains as many as $O(\frac{m}{k})$ clusters. By Lemma 10, all the clusters in $\mathcal{S}$ as well as their $\mathcal{X}_{\mathcal{C}}^{i}$, $\mathcal{Y}_{\mathcal{C}}^{j}$ and $\mathcal{Z}_{\mathcal{C}}^{h,t}$ neighbors can be found in $O(\frac{m^2}{k^2})$ time. We run the algorithm of Lemma 17 on all the clusters in $\mathcal{S}$. If either $\mathcal{S} = \emptyset$ or there is no bridge in each cluster in $\mathcal{S}$ separating x and y, we return *true*. Otherwise, there is at least one bridge internal to a cluster and we return *false*.

Lemma 18 *A* Same2EdgeBlock(x, y) *correctly returns* true *if the two vertices x and y are in the same 2-edge-connected component of G, and returns* false *otherwise. The total time required is $O(k + \frac{m^2}{k^2})$. When the graph is planar, the total time reduces to $O(k + \frac{n}{k}\log k)$.*

Proof: If there is a bridge separating x and y in $\mathcal{G}_{x,y}(G)$, then *Same2EdgeBlock*(x, y) returns *false*. This is correct because of Lemma 8 and requires $O(k + \frac{m^2}{k^2})$ time because of Lemma 7. Otherwise we compute $\mathcal{S}$ in $O(\frac{m^2}{k^2})$ time as described in Lemma 10 and run the algorithm given in Lemma 17 on all the clusters in $\mathcal{S}$. This gives a correct answer because of Lemmas 8 and 17. The total time required to test whether there is a bridge internal to a cluster in $\mathcal{S}$ is

$$\sum_{\mathcal{C}\in\mathcal{S}} d(\mathcal{C}) \log k$$

by Lemma 17. To bound this time, we notice that by Lemma 8 all the clusters in $\mathcal{S}$ are articulation points in $\mathcal{G}_{x,y}(G)$. Therefore each cluster $\mathcal{C}'$ can be in each of the sets $\{\mathcal{X}_{\mathcal{C}}^{i}\}_{1\leq i\leq p(\mathcal{C})}$, $\{\mathcal{Y}_{\mathcal{C}}^{j}\}_{1\leq j\leq q(\mathcal{C})}$, and $\{\mathcal{Z}_{\mathcal{C}}^{h,t}\}_{3\leq h\leq s(\mathcal{C}),1\leq t\leq r_h(\mathcal{C})}$ at most once for all possible choices of the articulation point $\mathcal{C}$ in $\mathcal{S}$ (that is, during the whole execution of the *Same2EdgeBlock*(x, y) operation). As a result, we have that

$$\sum_{\mathcal{C}\in\mathcal{S}} d(\mathcal{C}) = O\left(\frac{m}{k}\right).$$

Hence, the total time spent in checking whether there is a bridge inside the articulation points of $\mathcal{G}_{x,y}(G)$ is at most $O(\frac{m}{k}\log k)$. This gives a total time of $O(k + \frac{m^2}{k^2} + \frac{m}{k}\log k) = O(k + \frac{m^2}{k^2})$ for each *Same2EdgeBlock* operation.

If the underlying graph G is planar and insertions of new edges leave it planar, we have that $m = O(n)$. Therefore the topological partition contains $O(\frac{n}{k})$ clusters. Since G is planar and contraction preserves planarity, the graph obtained by contracting each cluster of G into one vertex is itself planar. As a result, denoting by $d(\mathcal{C})$ the external degree of cluster $\mathcal{C}$, we have

$$\sum_{\mathcal{C}} d(\mathcal{C}) = O\left(\frac{n}{k}\right).$$

and y in G.

This shows how to deal with case (3) of Lemma 13. Case (4) is analogous. Case (2) can be dealt with in a very similar fashion. The only difference is that now while searching for an edge satisfying condition (ii), we have to consider separately the two paths π_ρ between ρ and the root and π_β between β and the root. By applying the above algorithm at most twice, each time on a different path, we are able to check whether there is an edge in $\pi_{\rho,\beta}$ satisfying both conditions (i) and (ii) of Lemma 11.

Lemma 17 *Given a cluster $\mathcal{C}$ that is an articulation point separating x and y in $\mathcal{G}_{x,y}(G)$, we can find whether $\mathcal{C}$ contains a bridge separating x and y in G in $O(d(\mathcal{C})\log k)$ time, where $d(\mathcal{C})$ is the external degree of cluster $\mathcal{C}$.*

Proof: We analyze the time complexity of the above algorithm as follows. Condition (i) of Lemma 11 can be checked in $O((p(\mathcal{C})+q(\mathcal{C}))\log k)$ by Lemma 15.

We now bound the time required to check condition (ii). As explained before, the preprocessing required to find and sort the vertices $\gamma'_1, \gamma'_2, \ldots, \gamma'_r$ is $O(d(C)\log k)$. The time involved in examining vertex γ'_i for some i, $1 \le i \le r$, is the time spent in the range queries in the balanced search trees, which is $O(r_{i'}(\mathcal{C})\log k)$ by Lemma 14. Consequently, the total time required to examine the $r \le s(C)$ vertices γ'_i throughout the algorithm is

$$O(\sum_{i=1}^{s(\mathcal{C})} r_i(\mathcal{C})\log k) = O(d(\mathcal{C})\log k).$$

We now have to bound the time required to check whether there is an edge satisfying both conditions (i) and (ii). This involves performing range queries to test whether there is any either black or red vertex below the vertex γ' defined above. By Lemma 14, this can be implemented in $O((p(C)+q(C))\log k) = O(d(\mathcal{C})\log k)$ time.

Since case (2) is the most expensive part of Lemma 13, and it requires applying at most twice the above algorithm, we can check whether there is a bridge internal to a cluster $\mathcal{C}$ in a total of $O(d(\mathcal{C})\log k)$ time. □

3.4 Implementing the Operations

We perform a *Same2EdgeBlock*(x,y) operation as follows. We first compute $\mathcal{G}_{x,y}(G)$, the super-graph induced by vertices x and y. By Lemma 7, it can be computed in $O(k)$ time from $\widetilde{G}$. We then find the biconnected components of $\mathcal{G}_{x,y}(G)$ by using the linear-time algorithm of Tarjan [50]. Because of Lemma 7 this requires $O(k + \frac{m^2}{k^2})$ time in the worst case. If we find a bridge separating x and y in $\mathcal{G}_{x,y}(G)$, then we stop and declare x and y not to be in the same 2-edge-connected component of G by returning *false*. This is

(I1) Before examining vertex γ'_i, $1 \leq i \leq r$, $CANDIDATE = \gamma'_j$, $0 \leq j \leq i-1$, where either γ'_j is the root of $\mathcal{T}(\mathcal{C})$ or γ'_j is the deepest vertex such that $W'_1, W'_2, \ldots, W'_{i-1}$ are all compatible with e'_j (the edge from γ'_j to its parent).

At the beginning, $i = 1$, $CANDIDATE = \gamma'_0 = \rho$ and invariant (I1) trivially holds. We describe inductively how to examine vertex γ'_i, $1 \leq i \leq r$. Let $CANDIDATE = \gamma'_j$, $0 \leq j \leq i-1$, while examining vertex γ'_i. If $\gamma'_i = \gamma'_j$, then we set i to $i+1$. Since W'_i is compatible with $e'_i = e'_j$, invariant (I1) still holds. Otherwise, we check whether all the balanced search trees $E(\mathcal{Z}_C^{i',t}, \mathcal{C})$, $1 \leq t \leq r_{i'}(\mathcal{C})$, have vertices in the subtree rooted at γ'_j. By Lemma 14, this can be accomplished in a total of $O(r_{i'}(\mathcal{C}) \log k)$ time by means of $r_{i'}(\mathcal{C})$ range queries. If $E(\mathcal{Z}_C^{i',t}, \mathcal{C})$, $1 \leq t \leq r_{i'}(\mathcal{C})$, have no vertices in the subtree rooted at γ'_j, then the edge e'_j from γ'_j to its parent in $\mathcal{T}(\mathcal{C})$ is compatible with W'_i. Therefore γ'_j is still the deepest vertex such that $W'_1, W'_2, \ldots, W'_i$ are all compatible with e'_j. Again we set i to $i+1$ and invariant (I1) still holds. Otherwise, there is at least one vertex in the balanced search trees $E(\mathcal{Z}_C^{i',t}, \mathcal{C})$, $1 \leq t \leq r_{i'}(\mathcal{C})$, that is in the subtree rooted at γ_j. This implies that all the edges $e'_j, e'_{j+1}, \ldots, e'_{i-1}$ (corresponding to γ'_j and all the vertices examined after γ'_j) are not compatible with W'_i. However, either γ'_i is the root or edge e'_i is now compatible with $W'_1, W'_2, \ldots, W'_i$. Consequently, we set $CANDIDATE = \gamma'_i$ and $i = i+1$.

This shows that invariant (I1) is maintained after each vertex γ'_i is examined. Let $CANDIDATE = \gamma'$ after all the vertices have been examined. Let e' be the edge from γ' to its parent. Because of invariant (I1), e' is the deepest edge in $\{e'_0, e'_1, \ldots, e'_r\}$ that satisfies condition (ii) of Lemma 11. Notice that if γ' is the root of $\mathcal{T}(\mathcal{C})$, then e' is not defined; but in this case by Lemma 16 no edge of $\mathcal{T}(\mathcal{C})$ satisfies condition (ii). If γ' is not the root, we check whether there is an edge that satisfies both conditions (i) and (ii) as follows.

If γ' is an ancestor of β, then e' is not in $\pi_{\rho,\beta}$ and therefore cannot satisfy condition (i). This implies that also the edges in $\{e'_0, e'_1, \ldots, e'_r\}$ above e' cannot be in $\pi_{\rho,\beta}$ and consequently do not satisfy condition (i). Since by invariant (I1) e' is the deepest edge in $\{e'_0, e'_1, \ldots, e'_r\}$ satisfying condition (ii), the edges in $\{e'_0, e'_1, \ldots, e'_r\}$ below e' do not satisfy condition (ii). Therefore no edge in $\{e'_0, e'_1, \ldots, e'_r\}$ satisfies both conditions (i) and (ii). By Lemma 16, no edge in $\mathcal{T}(\mathcal{C})$ satisfy both conditions (i) and (ii), and by Lemma 11 there is no bridge in $\mathcal{C}$ separating x and y in G.

Otherwise, γ' is a proper descendant of β. By invariant (I1) e' is the deepest edge in $\{e'_0, e'_1, \ldots, e'_r\}$ satisfying condition (ii). To check whether e' satisfies also condition (i), we have to check that there are no black vertices in the subtree rooted at γ'. By Lemma 14, this can be accomplished in $O(q(\mathcal{C}) \log k)$ time. If there are no black vertices below γ', then e' satisfies both conditions (i) and (ii) and by Lemma 11 the corresponding edge of $\mathcal{C}$ is a bridge separating x and y in G. Otherwise, there is at least one black vertex in the subtree rooted at γ' and therefore e' cannot satisfy condition (i). Consequently, edges in $\{e'_0, e'_1, \ldots, e'_r\}$ above e' do not satisfy condition (i) either. By Lemma 16, no edge in $\mathcal{T}(\mathcal{C})$ satisfy both conditions (i) and (ii), and by Lemma 11 there is no bridge in $\mathcal{C}$ separating x

it γ_h. By Lemma 12, this can be done in $O(r_h(\mathcal{C}))$ time. Thus, in a total of

$$O(\sum_{h=3}^{s(\mathcal{C})} r_h(\mathcal{C})) = O(d(\mathcal{C}))$$

time, we can compute all the vertices $\gamma_3, \gamma_4, \ldots, \gamma_{s(\mathcal{C})}$. If some γ_h, $3 \leq h \leq s(\mathcal{C})$, is not in the path between ρ and the root of $\mathcal{T}(\mathcal{C})$, then the removal of any edge in $\pi_{\rho,\beta}$ leaves all the vertices in $E(\mathcal{Z}_{\mathcal{C}}^{h,t},\mathcal{C})$, $3 \leq t \leq r_h(\mathcal{C})$, on the same side. Therefore the corresponding component W_h is compatible with any edge in the path $\pi_{\rho,\beta}$. We examine all the γ_h vertices and discard the vertices not in the path between ρ and the root. Since we can test in $O(1)$ time whether a vertex γ_h is in the path between ρ and the root by simply checking whether $\gamma_h = \mathrm{lca}(\rho, \gamma_h)$, this step can be accomplished in a total of $O(s(\mathcal{C})) = O(d(\mathcal{C}))$ time. At the end of this step, we are left with $r \leq s(\mathcal{C})$ vertices $\gamma_{h_1}, \gamma_{h_2}, \ldots, \gamma_{h_r}$ in the path between ρ and the root of $\mathcal{T}(\mathcal{C})$, corresponding to the connected components for which we have still to check condition (ii) of Lemma 11. We sort the vertices γ_{h_i}, $1 \leq i \leq r$, in $O(r \log r)$ time according to their preorder numbering in $\mathcal{T}(\mathcal{C})$. Notice that $r \leq s(\mathcal{C}) \leq d(\mathcal{C})$. Furthermore, r is at most the total number of edges incident to cluster $\mathcal{C}$: $r \leq k$. Hence, $r \leq \min\{d(\mathcal{C}), k\}$ and $O(r \log r) = O(d(\mathcal{C}) \log k)$. Let $\gamma'_1, \gamma'_2, \ldots, \gamma'_r$ be the resulting sorted list, with $preorder(\gamma'_i) \geq preorder(\gamma'_{i+1})$. Let $\gamma'_0 = \rho$. For $0 \leq i \leq r$ let e'_i be the edge of $\mathcal{T}(\mathcal{C})$ from γ'_i to its parent, if γ'_i is not the root, and let it be undefined otherwise. Let W'_i be the connected component corresponding to γ'_i, and let $\mathcal{Z}_{\mathcal{C}}^{i',t}$, $1 \leq t \leq r_{i'}(\mathcal{C})$, be the clusters in W'_i adjacent to $\mathcal{C}$. The following lemma states that in our search for edges satisfying condition (ii) of Lemma 11, without any loss of generality we can restrict our attention to the edges $e'_1, e'_2, \ldots, e'_r$.

Lemma 16 *Let β be a proper ancestor of ρ. There is an edge of $\mathcal{T}(\mathcal{C})$ satisfying both conditions (i) and (ii) of Lemma 11 if and only if at least one edge e'_i, $0 \leq i \leq r$, satisfies conditions (i) and (ii).*

Proof: Assume there is an edge of $\mathcal{T}(\mathcal{C})$, say $e = (a, b)$ with b being the parent of a, satisfying both conditions (i) and (ii) of Lemma 11. By Lemma 13, the two vertices a and b must be in the path $\pi_{\rho,\beta}$. Let $T_1^{(a,b)}$ be the subtree of $\mathcal{T}(\mathcal{C})$ rooted at a, and $T_2^{(a,b)} = \mathcal{T}(\mathcal{C}) - \{T_1^{(a,b)} \cup (a, b)\}$. Let $\gamma'_0, \gamma'_1, \ldots, \gamma'_r$ be the sorted list of vertices in the path between ρ and the root as defined above. Let $\gamma'_0, \gamma'_1, \ldots, \gamma'_\ell$, $0 \leq \ell \leq r$, be all such vertices in the path between ρ and a (ρ and a included). Since e satisfies condition (ii), then each W'_i, $1 \leq i \leq r$, is compatible with e. This means that all the edges between $W'_1, W'_2, \ldots, W'_\ell$ and $\mathcal{C}$ are incident to $T_1^{(a,b)}$, and that all the edges between $W'_{\ell+1}, W'_{\ell+2}, \ldots, W'_r$ and $\mathcal{C}$ are incident to $T_2^{(a,b)}$. By definition of the γ'_i's, the edges between $W'_1, W'_2, \ldots, W'_\ell$ and $\mathcal{C}$ are all incident to vertices of $\mathcal{T}(\mathcal{C})$ below γ'_ℓ. Therefore each W'_i, $1 \leq i \leq r$, is compatible with e'_ℓ too. □

We examine the vertices γ'_i, $1 \leq i \leq r$, according to decreasing preorder numbering, i.e., starting with γ'_1. We maintain a vertex called *CANDIDATE* satisfying the following invariant:

$E(\mathcal{C}',\mathcal{C})$ is in the subtree of $\mathcal{T}(\mathcal{C})$ rooted at v if and only if the preorder number of u satisfies $preorder(v) \leq preorder(u) \leq preorder(v) + size(v) - 1$. Consequently, we have to check whether the balanced search tree $E(\mathcal{C}',\mathcal{C})$ contains at least one item in the range $[preorder(v), preorder(v) + size(v) - 1]$. This range query can be accomplished by finding the smallest item of the balanced search tree $E(\mathcal{C}',\mathcal{C})$ greater than or equal to $preorder(v)$, which can be done in $O(\log |E(\mathcal{C}',\mathcal{C})|) = O(\log k)$ time. Then $E(\mathcal{C}',\mathcal{C})$ contains at least one item in the range $[preorder(v), preorder(v) + size(v) - 1]$ if and only if the item found is less than or equal to $preorder(v) + size(v) - 1$. □

Lemma 15 *Given a tree representation $\mathcal{T}(\mathcal{C})$ of a cluster $\mathcal{C}$ that is an articulation point separating x and y in $\mathcal{G}_{x,y}(G)$, we can find whether red and black vertices can be separated in $\mathcal{T}(\mathcal{C})$ in $O((p(\mathcal{C}) + q(\mathcal{C})) \log k)$ time.*

Proof: The three vertices ρ, β and v can be computed in $O(p(\mathcal{C}) + q(\mathcal{C}))$ time by Corollary 1. By Lemma 13, we have to check only four possible conditions. Conditions (1) and (2) can be checked in constant time and require no further work.

We show how to perform efficiently the test implied in condition (3) for each $E(\mathcal{Y}_{\mathcal{C}}^{j},\mathcal{C})$, $1 \leq j \leq q(\mathcal{C})$. Notice that in order to check whether red and black vertices can be separated in $\mathcal{T}(\mathcal{C})$, it is enough to check that there are no black vertices in the subtree of $\mathcal{T}(\mathcal{C})$ rooted at ρ. By Lemma 14, we can check in $O(\log k)$ time whether there are edges between $\mathcal{Y}_{\mathcal{C}}^{j}$ and $\mathcal{C}$. Since we have to test this for all the possible clusters $\mathcal{Y}_{\mathcal{C}}^{j}$, $1 \leq j \leq q(\mathcal{C})$, condition (3) requires a total of $O(q(\mathcal{C}) \log k)$ time.

Condition (4) of Lemma 13 can be checked with the same algorithm, but this time using the clusters $\mathcal{X}_{\mathcal{C}}^{i}$ instead, $1 \leq i \leq p(C)$. This requires a total of $O(p(\mathcal{C}) \log k)$ time.

In summary, we can check whether there is an edge separating red and black vertices in $\mathcal{T}(\mathcal{C})$ in a total of $O((p(\mathcal{C}) + q(\mathcal{C})) \log k)$ time. □

By Lemma 15, we can check in $O((p(\mathcal{C}) + q(\mathcal{C})) \log k)$ time whether black and red vertices can be separated in cluster $\mathcal{C}$, i.e., whether there is an edge of $\mathcal{T}(\mathcal{C})$ that satisfies condition (i) of Lemma 11. If red and black vertices cannot be separated, then by Lemma 11 there cannot be any bridge inside cluster $\mathcal{C}$ separating vertices x and y in G.

Otherwise, if red and black vertices can be separated, we check condition (ii) as follows. Let ρ be the top-red vertex and let β be the top-black vertex in $\mathcal{T}(\mathcal{C})$. If red and black vertices can be separated, it means that we are either in case (2), (3) or (4) of Lemma 13. As shown in Lemma 13, $\pi_{\rho,\beta}$, the path between vertices ρ and β in $\mathcal{T}(\mathcal{C})$, contains all the possible edges that separates red and black vertices and for which condition (ii) of Lemma 11 has to be checked.

We first consider case (3) of Lemma 13. We consider each connected component W_h, $3 \leq h \leq s(\mathcal{C})$. We compute the lca of all the vertices in $E(\mathcal{Z}_{\mathcal{C}}^{h,t},\mathcal{C})$, $1 \leq t \leq r_h(\mathcal{C})$, and call

either two black vertices or two red vertices cannot separate black and red vertices. We now analyze the four different cases.

In case (1) $v = \rho = \beta$. By definition of lca we must have two (not necessarily distinct) red vertices r_1 and r_2 and two (not necessarily distinct) black vertices b_1 and b_2 such that the path π_R between r_1 and r_2 and the path π_B between b_1 and b_2 intersect in v. Notice however that if $b_1 = b_2$ [$r_1 = r_2$] then it must be $v = b_1 = b_2$ [$v = r_1 = r_2$]. Assume by contradiction that there is an edge e in $\mathcal{T}(\mathcal{C})$ whose removal separates red and black vertices. Clearly e cannot be in π_B, otherwise two black vertices b_1 and b_2 will be separated. For the same reason e cannot be in π_R. But this implies that after the removal of e, we have that a red vertex (r_1 or r_2) is still reachable from a black vertex (b_1 or b_2), a contradiction.

We now show that in cases (2), (3) and (4) only edges in $\pi_{\rho,\beta}$ can possibly separate red and black vertices. Indeed the removal of an edge not in $\pi_{\rho,\beta}$ leaves β and ρ in the same tree. The same argument given in case (1) shows that such an edge cannot separate red and black vertices.

Consider now case (2). All the red vertices are in the subtree rooted at ρ, and all the black vertices are in the subtree rooted at β. Since $v = \mathrm{lca}(\rho, \beta)$ is different from both ρ and β, then all the edges in $\pi_{\rho,\beta}$ (and as shown before only them) separate red and black vertices.

As for case (3), let $e = (a, b)$ be any edge in $\pi_{\rho,\beta}$, with b being the parent of a. If there is no black vertex in the subtree rooted at a, then the removal of e will separate red and black vertices. Assume now there is a black vertex, say b_1, in the subtree rooted at a. Since β is the lca of all the black vertices and it is a proper ancestor of a, there must be another black vertex $b_2 \neq b_1$ outside the subtree rooted at a. Therefore the removal of e will split $\mathcal{T}(\mathcal{C})$ into two pieces, each of which contains at least one black vertex. As a result, e cannot separate red and black vertices.

Case (4) is analogous to case (3). □

Conditions (1) and (2) of Lemma 13 are both easy to check once ρ, β and v are available; furthermore, they immediately give all edges separating red and black vertices. We still need to explain how to check conditions (3) and (4). Without loss of generality we restrict our attention to condition (3). We do it separately for each $\mathcal{Y}_{\mathcal{C}}^{j}$ cluster inducing black vertices in $\mathcal{T}(\mathcal{C})$. The idea underlying a fast algorithm to accomplish this task is to use a kind of binary search instead of checking all the possible black vertices.

Lemma 14 *Let $\mathcal{C}'$ be any cluster adjacent to $\mathcal{C}$, and let v be any vertex of $\mathcal{T}(\mathcal{C})$. Then we can check whether there are edges between $\mathcal{C}'$ and $\mathcal{C}$ incident below v in $O(\log k)$ time.*

Proof: By definition of $E(\mathcal{C}', \mathcal{C})$, there are edges between $\mathcal{C}'$ and $\mathcal{C}$ below v if and only if there are vertices of $E(\mathcal{C}', \mathcal{C})$ below v. Because of preorder numbering, a vertex u in

Proof: For $1 \leq i \leq \ell$ find λ_i, the lca of all the vertices in $E(\mathcal{A}_i,\mathcal{C})$. Since $E(\mathcal{A}_i,\mathcal{C})$ is sorted according to the preorder numbering of $\mathcal{T}(\mathcal{C})$, this can be done by first finding the leftmost and rightmost vertices u_i and v_i in $E(\mathcal{A}_i,\mathcal{C})$, and then by computing $\lambda_i = \mathrm{lca}(u_i, v_i)$. Then $\lambda(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_\ell)$ can be found as the lca of all the λ_i's by performing at most $O(\ell)$ lca queries. The total time required can be bounded as follows. Finding the leftmost and rightmost item in a balanced search tree $E(\mathcal{A}_i,\mathcal{C})$ requires $O(\log|E(\mathcal{A}_i,\mathcal{C})|)$ time. However, this can be performed in constant time if we maintain extra pointers from the root of each balanced search tree to the leftmost and rightmost items in the tree. The time required to perform an lca query in $\mathcal{T}(\mathcal{C})$ is $O(1)$. Since we have at most a total of $O(\ell)$ queries, the time spent in finding the leftmost and rightmost vertices in all the balanced search trees $E(\mathcal{A}_i,\mathcal{C})$, $1 \leq i \leq \ell$, and in performing all the lca queries will be $O(\ell)$. Consequently, the total time required is $O(\ell)$. □

Define the *top-red* vertex ρ as $\lambda(\mathcal{X}_\mathcal{C}^1, \mathcal{X}_\mathcal{C}^2, \ldots, \mathcal{X}_\mathcal{C}^{p(\mathcal{C})})$. Notice that ρ is the lca of all the red vertices in $\mathcal{T}(\mathcal{C})$. Similarly, define the *top-black* vertex β as $\lambda(\mathcal{Y}_\mathcal{C}^1, \mathcal{Y}_\mathcal{C}^2, \ldots, \mathcal{Y}_\mathcal{C}^{q(\mathcal{C})})$, that is as the lca of all the black vertices in $\mathcal{T}(\mathcal{C})$. The following corollary is a consequence of Lemma 12.

Corollary 1 *The top-red and the top-black vertices ρ and β in $\mathcal{T}(\mathcal{C})$ can be found in $O(p(\mathcal{C}) + q(\mathcal{C}))$ time.*

After locating the top-red and top-black vertices ρ and β, we compute in $O(1)$ time $v = \mathrm{lca}(\rho, \beta)$. We now have four different cases depending on the relative positions of the three vertices v, ρ and β in $\mathcal{T}(\mathcal{C})$.

Lemma 13 *Let ρ be the top-red vertex, β be the top-black vertex, and let $v = \mathrm{lca}(\rho, \beta)$. Let $\pi_{\rho,\beta}$ be the path in $\mathcal{T}(\mathcal{C})$ between ρ and β. Then*

(1) If $v = \rho = \beta$, no edge in $\mathcal{T}(\mathcal{C})$ can separate black and red vertices.

(2) If $v \neq \rho$ and $v \neq \beta$, then all the edges in $\pi_{\rho,\beta}$ (and only them) separate black and red vertices.

(3) If $v = \beta \neq \rho$, an edge e separates black and red vertices if and only if e is in $\pi_{\rho,\beta}$ and there are no black vertices below e.

(4) If $v = \rho \neq \beta$, an edge e separates black and red vertices if and only if e is in $\pi_{\rho,\beta}$ and there are no red vertices below e.

Proof: We exploit the following fact. The deletion of an edge splits the tree $\mathcal{T}(\mathcal{C})$ into two trees. Since there is at least one red vertex and one black vertex in $\mathcal{T}(\mathcal{C})$, an edge separates black and red vertices if and only if its removal leaves all the black vertices in one tree and all the red vertices in the other tree. As a result, an edge that separates

Figure 3.5: An edge satisfying Lemma 11

the clusters $\mathcal{Y}_{\mathcal{C}}^{j}$, $1 \leq j \leq q(\mathcal{C})$, as *black clusters*. The other clusters $\mathcal{Z}_{\mathcal{C}}^{h,t}$, $3 \leq h \leq s(\mathcal{C})$, $1 \leq t \leq r_h(\mathcal{C})$, are referred to as *W-clusters*. Notice that since $p(\mathcal{C}) \geq 1$ and $q(\mathcal{C}) \geq 1$, we have at least one black vertex and one red vertex in $\mathcal{T}(\mathcal{C})$.

We need the following new terminology. Given the tree representation $\mathcal{T}(\mathcal{C})$ of cluster $\mathcal{C}$, and an edge e in $\mathcal{T}(\mathcal{C})$, let T_1^e and T_2^e be the two trees obtained by removing e from $\mathcal{T}(\mathcal{C})$. We recall that $W_1, W_2, \ldots, W_{s(\mathcal{C})}$ are the connected components left in the super-graph $\widetilde{G}$ after the removal of cluster $\mathcal{C}$. We say that a connected component W_h, $1 \leq h \leq s(\mathcal{C})$, is *compatible with* edge e if all the edges between clusters $\mathcal{Z}_{\mathcal{C}}^{h,t}$, $1 \leq t \leq r_h(C)$, are incident to either T_1^e or T_2^e, but not to both.

Lemma 11 *Let $\mathcal{C}$ be a cluster that is an articulation point separating x and y in $\mathcal{G}_{x,y}(G)$. There is a bridge inside cluster $\mathcal{C}$ that separates vertices x and y in G if and only if there is an edge e in $\mathcal{T}(\mathcal{C})$ that satisfies the following two conditions:*

(i) the removal of e separates red and black vertices; and

(ii) each connected component W_h, $3 \leq h \leq s(\mathcal{C})$, is compatible with e.

Proof: By definition of the clusters $\mathcal{X}_{\mathcal{C}}^{i}$ and $\mathcal{Y}_{\mathcal{C}}^{j}$, every path between x and y in G must have the following structure. It starts with a path outside $\mathcal{C}$ ending at a vertex of $\mathcal{C}$ that corresponds to a red vertex in $\mathcal{T}(\mathcal{C})$. It ends with a path outside $\mathcal{C}$ starting at a vertex of $\mathcal{C}$ that corresponds to a black vertex in $\mathcal{T}(\mathcal{C})$.

Thus, there is an edge e in $\mathcal{T}(\mathcal{C})$ for which conditions (i) and (ii) hold if and only if the removal of the edge corresponding to e in G separates x and y. Condition (i) holds if and only if any path in $\mathcal{C}$ between a vertex corresponding to a red vertex and a vertex corresponding to a black vertex contains e. Condition (ii) holds if and only if there is no detour avoiding e through component W_h, $3 \leq h \leq s(\mathcal{C})$. □

Figure 3.5 shows an example of an edge that satisfies both conditions of Lemma 11.

We remark that, in order to check conditions (i) and (ii) of Lemma 11, we cannot even afford to enumerate all the red and black vertices explicitly. Indeed this can take as much as $\Omega(k)$ time for each cluster $\mathcal{C}$ we need to consider. Since we may have to perform this task in $\Omega(\frac{m}{k})$ different clusters, these times would sum up to a total of $\Omega(m)$. We check the two conditions of Lemma 11 independently, and then intersect the solutions of these two computations.

We first describe how to check efficiently condition (i). Before, we need few technical lemmas. Let $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_\ell$, $\ell > 0$, be any ℓ clusters adjacent to cluster $\mathcal{C}$. Then define by $\lambda(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_\ell)$ the deepest vertex in $\mathcal{T}(\mathcal{C})$ such that all the edges between $\mathcal{A}_i$, $1 \leq i \leq \ell$, and $\mathcal{C}$ are incident below $\lambda(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_\ell)$.

Lemma 12 *The vertex $\lambda(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_\ell)$ can be found in $O(\ell)$ time.*

removing $\mathcal{C}$ from $\widetilde{G}$ breaks $\widetilde{G}$ into, say, $s(\mathcal{C}) \geq 2$ connected components $W_1, W_2, \ldots, W_{s(\mathcal{C})}$. Without loss of generality, let W_1 be the connected component containing $\mathcal{C}_x$, and let W_2 be the connected component containing $\mathcal{C}_y$. Denote by $\mathcal{Z}_{\mathcal{C}}^{h,1}, \mathcal{Z}_{\mathcal{C}}^{h,2}, \ldots, \mathcal{Z}_{\mathcal{C}}^{h,r_h(\mathcal{C})}$ the clusters previously adjacent to $\mathcal{C}$ and now in connected component W_h, $1 \leq h \leq s(\mathcal{C})$. Let $\mathcal{X}_{\mathcal{C}}^i = \mathcal{Z}_{\mathcal{C}}^{1,i}$, $i = 1,2,\ldots r_1(\mathcal{C}) \equiv p(\mathcal{C})$ (i.e., $\mathcal{X}_{\mathcal{C}}^1, \mathcal{X}_{\mathcal{C}}^2, \ldots, \mathcal{X}_{\mathcal{C}}^{p(\mathcal{C})}$ are the clusters previously adjacent to $\mathcal{C}$ and now in the same connected component as $\mathcal{C}_x$). Similarly, let $\mathcal{Y}_{\mathcal{C}}^j = \mathcal{Z}_{\mathcal{C}}^{2,j}$, $j = 1,2,\ldots r_2(\mathcal{C}) \equiv q(\mathcal{C})$ (i.e., $\mathcal{Y}_{\mathcal{C}}^1, \mathcal{Y}_{\mathcal{C}}^2, \ldots, \mathcal{Y}_{\mathcal{C}}^{q(\mathcal{C})}$ are the clusters previously adjacent to $\mathcal{C}$ and now in the same connected component as $\mathcal{C}_y$). Notice that $p(\mathcal{C}) \geq 1$, $q(\mathcal{C}) \geq 1$, $r_h(\mathcal{C}) \geq 1$, $3 \leq h \leq s(\mathcal{C})$, and $p(\mathcal{C}) + q(\mathcal{C}) + \sum_{h=3}^{s(\mathcal{C})} r_h(\mathcal{C}) = d(\mathcal{C})$, where $d(\mathcal{C})$ is the number of clusters adjacent to $\mathcal{C}$, also called the *external degree* of cluster $\mathcal{C}$.

Lemma 10 *Given $\widetilde{G}$, all the clusters $\mathcal{C}$ that are articulation points separating x and y in $\mathcal{G}_{x,y}(G)$, as well as their neighboring clusters $\mathcal{X}_{\mathcal{C}}^i$, $1 \leq i \leq p(\mathcal{C})$, $\mathcal{Y}_{\mathcal{C}}^j$, $1 \leq j \leq q(\mathcal{C})$, and $\mathcal{Z}_{\mathcal{C}}^{h,t}$, $3 \leq h \leq s(\mathcal{C})$, $1 \leq t \leq r_h(\mathcal{C})$, can be found in a total of $O(\frac{m^2}{k^2})$ time.*

Proof: Compute the biconnected components of $\widetilde{G}$ using the algorithm of Tarjan [50], and subsequently construct the block tree of $\widetilde{G}$ (as defined for instance in [23]). We recall here that a block tree of a graph is composed of square nodes (corresponding to vertices in the graph) and round nodes (corresponding to biconnected components in the graph): whenever a vertex belongs to a given biconnected component, there is a tree edge between the corresponding square and round nodes in the block tree. As a consequence, each path in a block tree alternates between square and round nodes, and the block tree has size bounded by the total number of square nodes, i.e., by the number of vertices in the original graph. The construction of the block tree is immediate once the biconnected components have been found. Since $\widetilde{G}$ has $O(\frac{m}{k})$ vertices and $O(\frac{m^2}{k^2})$ edges, its biconnected components and its block tree can be computed in $O(\frac{m^2}{k^2})$ time. Furthermore, the block tree has size $O(\frac{m}{k})$. Due to the property of block trees, all the clusters $\mathcal{C}$ that are articulation points separating x and y in $\mathcal{G}_{x,y}(G)$ can be found as the square nodes in the path between $\mathcal{C}_x$ and $\mathcal{C}_y$ in the block tree of $\widetilde{G}$. Furthermore, each cluster $\mathcal{C}'$ can be adjacent to at most two clusters that are articulation points separating x and y in $\mathcal{G}_{x,y}(G)$. Consequently, each cluster $\mathcal{C}'$ can be in each of the sets $\{\mathcal{X}_{\mathcal{C}}^i\}_{1 \leq i \leq p(\mathcal{C})}$, $\{\mathcal{Y}_{\mathcal{C}}^j\}_{1 \leq j \leq q(\mathcal{C})}$, and $\{\mathcal{Z}_{\mathcal{C}}^{h,t}\}_{3 \leq h \leq s(\mathcal{C}), 1 \leq t \leq r_h(\mathcal{C})}$ at most once for all possible choices of the articulation point $\mathcal{C}$ in $\widetilde{G}$. As a result, all the neighbors $\mathcal{X}_{\mathcal{C}}^i$, $\mathcal{Y}_{\mathcal{C}}^j$, and $\mathcal{Z}_{\mathcal{C}}^{h,t}$ of all these clusters $\mathcal{C}$ can be found in a total of $O(\frac{m^2}{k^2})$ time by visiting both $\widetilde{G}$ and its block tree. □

We now show how to check whether there is a bridge internal to a cluster $\mathcal{C}$ that is an articulation point in $\mathcal{G}_{x,y}(G)$. Assume that a vertex v in $\mathcal{T}(\mathcal{C})$, the tree representation of $\mathcal{C}$, is colored *red* if there is an edge in G between a vertex in cluster $\mathcal{X}_{\mathcal{C}}^i$, $1 \leq i \leq p(\mathcal{C})$, and v. Similarly, assume that a vertex v in $\mathcal{T}(\mathcal{C})$ is colored *black* if there is an edge in G between a vertex in cluster $\mathcal{Y}_{\mathcal{C}}^j$, $1 \leq j \leq q(\mathcal{C})$, and v. The red vertices are all the vertices in $E(\mathcal{X}_{\mathcal{C}}^i, \mathcal{C})$, $1 \leq i \leq p(\mathcal{C})$, and the black vertices are all the vertices in $E(\mathcal{Y}_{\mathcal{C}}^j, \mathcal{C})$, $1 \leq j \leq q(\mathcal{C})$. Similarly, we denote the clusters $\mathcal{X}_{\mathcal{C}}^i$, $1 \leq i \leq p(\mathcal{C})$, as *red clusters*, and

$\mathcal{T}(\mathcal{C})$; and

(iii) for each non-empty edge set $E_{i,j}$ between two clusters $\mathcal{C}_i$ and $\mathcal{C}_j$, we maintain the two balanced search trees $E(\mathcal{C}_i,\mathcal{C}_j)$ and $E(\mathcal{C}_j,\mathcal{C}_i)$.

$\mathcal{F}(\mathcal{C})$, $\mathcal{T}(\mathcal{C})$, and all the balanced search trees $E(\mathcal{C}',\mathcal{C})$ for clusters $\mathcal{C}'$ adjacent to $\mathcal{C}$ are referred to as the *data structures pertinent to cluster* $\mathcal{C}$. The following lemma bounds the total time required to initialize all the data structures pertinent to a cluster as well as their space usage.

Lemma 9 *All the data structures pertinent to a cluster* $\mathcal{C}$ *can be initialized in a total of* $O(k)$ *time and require* $O(k)$ *space.*

Proof: The full representation $\mathcal{F}(\mathcal{C})$ of a cluster contains at most $O(k)$ edges and vertices and therefore can be computed in $O(k)$ time. On the other hand, the tree representation $\mathcal{T}(\mathcal{C})$ can be obtained by computing the 2-edge-connected components of the subgraph $\mathcal{G}(\mathcal{C})$ induced by cluster $\mathcal{C}$. Since $\mathcal{G}(\mathcal{C})$ has at most $O(k)$ vertices and edges this can be done in $O(k)$ time [50]. Furthermore, the preprocessing involved in computing $size(v)$ and $preorder(v)$ for each vertex v, as well as in organizing $\mathcal{T}(\mathcal{C})$ as a fast lca data structure requires $O(|\mathcal{T}(\mathcal{C})|) = O(k)$ time [25, 42].

We compute all the balanced search trees $E(\mathcal{C}',\mathcal{C})$, for each cluster $\mathcal{C}'$ neighbor of $\mathcal{C}$, at the same time. We traverse $\mathcal{T}(\mathcal{C})$ in preorder and we build the balanced search trees by inserting an item in $E(\mathcal{C}',\mathcal{C})$ each time we find an edge between the vertex v of $\mathcal{T}(\mathcal{C})$ we are currently visiting and cluster $\mathcal{C}'$. Since $\mathcal{T}(\mathcal{C})$ is visited in a preorder fashion, items are always inserted at the end of a balanced search tree. Therefore each balanced search tree can be built in linear time. The total time involved in building all these balanced search trees is

$$O(\sum_{\mathcal{C}'} |E(\mathcal{C}',\mathcal{C})|) = O(k),$$

since there can be at most a total of $O(k)$ edges of G incident to cluster $\mathcal{C}$.

Consequently, all the data structures pertinent to a cluster $\mathcal{C}$ can be initialized in a total of $O(k)$ time. Obviously, they require $O(k)$ space. □

3.3 Testing for Bridges Inside Clusters

The data structures given above can be used to check efficiently whether there is a bridge inside a cluster $\mathcal{C}$ that is an articulation point separating x and y in $\mathcal{G}_{x,y}(G)$.

Consider the super-graph $\widetilde{G}$. Since $\widetilde{G}$ can be obtained from $\mathcal{G}_{x,y}(G)$ by shrinking the two clusters $\mathcal{C}_x$ and $\mathcal{C}_y$ into one super-vertex and by reducing to two the number of multiple edges, $\mathcal{C}$ must be an articulation point separating $\mathcal{C}_x$ and $\mathcal{C}_y$ in $\widetilde{G}$. Thus,

By Lemma 8, a type (1) bridge can be detected by simply computing the 2-edge-connected components of $\mathcal{G}_{x,y}(G)$. This can be performed by using the off-line linear-time algorithm of Tarjan [50] on $\mathcal{G}_{x,y}(G)$, which requires $O(k + \frac{m^2}{k^2})$ time because of Lemma 7. The crucial task is therefore how to check efficiently whether there is a bridge separating x and y within a cluster $\mathcal{C}$. We will show how to perform this task in the next subsection. First, we need to introduce some new terminology and data structures.

For each cluster $\mathcal{C}$, we maintain its *tree representation* defined as follows. We recall that we denote by $\mathcal{G}(\mathcal{C})$ the subgraph of G induced by cluster $\mathcal{C}$ (i.e., the graph having only vertices and edges internal to $\mathcal{C}$). Compute the 2-edge-connected components of $\mathcal{G}(\mathcal{C})$. They form a tree $\mathcal{T}(\mathcal{C})$, whose edges are the bridges in $\mathcal{G}(\mathcal{C})$. $\mathcal{T}(\mathcal{C})$ is called the *tree representation* of cluster $\mathcal{C}$. As a consequence of any cluster having at most $O(k)$ edges and vertices, the tree representation of a cluster has size at most $O(k)$ and can be computed in $O(k)$ time.

For each cluster $\mathcal{C}$, we root arbitrarily at any vertex the tree representation $\mathcal{T}(\mathcal{C})$, and maintain it as a data structure that supports fast *lowest common ancestor* (in short *lca*) queries [25, 42]. We assume that in a rooted tree edges are directed from the children to the parent. We recall here that given a rooted tree T and two vertices u and v, the lowest common ancestor of u and v, denoted by $\mathrm{lca}(u, v)$, is the deepest vertex in T that is ancestor of both u and v. The data structures proposed in [25, 42] answer each lca query in $O(1)$ time after a linear-time preprocessing on the tree. Therefore in our case, we need an extra $O(k)$ time to organize $\mathcal{T}(\mathcal{C})$ as a fast lca data structure; but then we can answer any lca query in $\mathcal{T}(\mathcal{C})$ in constant time. Finally, for each vertex v of $\mathcal{T}(\mathcal{C})$, we compute $size(v)$ as the size of the subtree rooted at v, and $preorder(v)$ as the preorder number of v. For all vertices v in $\mathcal{T}(\mathcal{C})$, $size(v)$ and $preorder(v)$ can be computed in a total of $O(k)$ time during a preorder traversal of $\mathcal{T}(\mathcal{C})$.

Furthermore, for any two clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ let $E_{i,j} = \{e_1, e_2, \ldots, e_h\}$ be the set of edges of the original graph G between $\mathcal{C}_i$ and $\mathcal{C}_j$. We represent edges in $E_{i,j}$ using two data structures: one for $\mathcal{C}_i$, named $E(\mathcal{C}_i,\mathcal{C}_j)$, and the other for $\mathcal{C}_j$, named $E(\mathcal{C}_j,\mathcal{C}_i)$. $E(\mathcal{C}_i,\mathcal{C}_j)$ stores information about the endpoints of edges in $E_{i,j}$ falling into cluster $\mathcal{C}_j$. $E(\mathcal{C}_j,\mathcal{C}_i)$ is similar. Specifically, $E(\mathcal{C}_i,\mathcal{C}_j)$ contains all the vertices u in $\mathcal{T}(\mathcal{C}_j)$ such that there is an edge in $E_{i,j}$ incident to u. We recall that vertices of $\mathcal{T}(\mathcal{C}_j)$ represent 2-edge-connected components of $\mathcal{G}(\mathcal{C}_j)$, the subgraph of G induced by cluster $\mathcal{C}_j$. Furthermore we maintain $E(\mathcal{C}_i,\mathcal{C}_j)$ as an ordered set according to the preorder numbering of $\mathcal{T}(\mathcal{C}_j)$. To access and update $E(\mathcal{C}_i,\mathcal{C}_j)$ and $E(\mathcal{C}_j,\mathcal{C}_i)$ efficiently, we maintain them as balanced search trees [2]; the key of vertex v is $preorder(v)$.

In summary, throughout our sequence of *Same2EdgeBlock*, *InsertEdge* and *DeleteEdge* operations we maintain the topological partition into clusters together with the following data structures:

(i) $\widetilde{G}$, the super-graph of G;

(ii) for each cluster $\mathcal{C}$ we maintain its full representation $\mathcal{F}(\mathcal{C})$ and its tree representation

Figure 3.4: A graph and the super-graph induced by two vertices

is, replace vertex $\mathcal{C}_x$ and the edges incident to it in $\widetilde{G}$ with $\mathcal{G}(\mathcal{C}_x)$ (the subgraph induced by cluster $\mathcal{C}_x$) and the edges of G incident to cluster $\mathcal{C}_x$. Do exactly the same for cluster $\mathcal{C}_y$. Denote the obtained graph $\mathcal{G}_{x,y}(G)$ and call it *super-graph of G induced by vertices x and y*. Figure 3.4 shows a graph $G = (V, E)$ and the super-graph induced by two vertices.

We now characterize some properties of $\mathcal{G}_{x,y}(G)$.

Lemma 7 *For any two vertices x and y, $\mathcal{G}_{x,y}(G)$ has at most $O(k + \frac{m}{k})$ vertices and $O(k + \frac{m^2}{k^2})$ edges and can be computed in a total of $O(k)$ time from the super-graph $\widetilde{G}$.*

Proof: Let $\widetilde{G}$ be the super-graph of G. Since there are at most $O(\frac{m}{k})$ clusters in the topological partition of the original graph G, the number of vertices in $\widetilde{G}$ is $O(\frac{m}{k})$. Furthermore, there can be at most two multiple edges between any two vertices of $\widetilde{G}$; consequently, the number of edges in $\widetilde{G}$ is $O(\frac{m^2}{k^2})$. $\mathcal{G}_{x,y}(G)$ is obtained from $\widetilde{G}$ by replacing at most two of its vertices with the full representation of their corresponding clusters. Since the full representation of a cluster has at most $3k - 2$ vertices each of degree at most three, this introduces at most $O(k)$ new vertices and edges. As a result, $\mathcal{G}_{x,y}(G)$ has at most $O(k + \frac{m}{k})$ vertices and $O(k + \frac{m^2}{k^2})$ edges. Furthermore, once the super-graph $\widetilde{G}$ is given, $\mathcal{G}_{x,y}(G)$ can be computed in $O(k)$ time. □

Let x and y be two vertices that are not in the same 2-edge-connected component of G. Then there is a bridge separating x and y in G. Any such bridge e can be of two different types: type (1) if e is either internal to $\mathcal{C}_x$ and $\mathcal{C}_y$ or is an edge between two different clusters; and type (2) if e is internal to a cluster $\mathcal{C}$, $\mathcal{C} \neq \mathcal{C}_x$, $\mathcal{C} \neq \mathcal{C}_y$.

Lemma 8 *Let x and y be any two vertices of G. Then the following is true:*

(i) e is a type (1) bridge in G if and only if the corresponding edge is a bridge in $\mathcal{G}_{x,y}(G)$.

(ii) If e is a type (2) bridge separating x and y in G, and e is internal to a cluster $\mathcal{C}$ ($\mathcal{C} \neq \mathcal{C}_x$, $\mathcal{C} \neq \mathcal{C}_y$), then $\mathcal{C}$ is an articulation point in $\mathcal{G}_{x,y}(G)$.

Proof: To prove (i), we observe that an edge e in $\mathcal{G}_{x,y}(G)$ corresponds either to an internal edge of $\mathcal{C}_x$ or $\mathcal{C}_y$ or to an edge connecting two different clusters. It is easy to see that there is a cycle in $\mathcal{G}_{x,y}(G)$ containing edge e if and only if there is a cycle in G containing the corresponding edge.

We now turn to (ii). If e is a bridge separating x and y in G, then its removal disconnects x and y in G. Since $\mathcal{G}_{x,y}(G)$ can be obtained from G by contracting each cluster except $\mathcal{C}_x$ and $\mathcal{C}_y$ into a single vertex and by reducing to two the number of multiple edges, then the removal of $\mathcal{C}$ must disconnect x and y in $\mathcal{G}_{x,y}(G)$. Therefore $\mathcal{C}$ must be an articulation point in $\mathcal{G}_{x,y}(G)$. □

Figure 3.3: The full representation of a cluster

Figure 3.2: A graph and its supergraph

This transformation preserves 2-edge-connected components, as the following lemma shows.

Lemma 6 *Let be (x,y) any edge of G and let (x_i,y_j) the corresponding edge in G'. Then (x,y) is a bridge in G if and only if (x_i,y_j) is a bridge in G'.*

Proof: Immediate from the fact that by construction (x,y) lies in a (not necessarily simple) cycle in G if and only if (x_i,y_j) lies in a (not necessarily simple) cycle in G'. □

As a consequence of Lemma 6 and the fact that new edges of G' cannot be bridges, we restrict our attention to graphs with $O(m)$ edges and vertices and whose vertices have degree not exceeding three. We obtain efficient algorithms for our problem by partitioning a graph G into clusters and by maintaining a succinct internal representation of each cluster. We do this as follows. We consider any spanning tree T of G and generate a topological partition of T of order k as defined in [18], with k being a positive integer to be chosen later. We recall that a topological partition of T of order k consists of $O(\frac{m}{k})$ vertex clusters of T such that each cluster contains a connected subgraph with between k and $3k-2$ vertices. As shown in [18], this can be accomplished in $O(m)$ time. Throughout the sequence of operations, we maintain for each vertex v of G information about the cluster of the topological partition containing v, in such a way that we can find in constant time the cluster containing a given vertex. This information can be easily initialized and updated during any sequence of our operations, and we will not mention this any further. Furthermore, we maintain the graph $\widetilde{G}$ obtained from G by contracting each cluster $\mathcal{C}$ into a single vertex (also denoted by $\mathcal{C}$). If there are more than two edges between cluster $\mathcal{C}_i$ and $\mathcal{C}_j$ in G, then we represent only two edges between $\mathcal{C}_i$ and $\mathcal{C}_j$ in $\widetilde{G}$. We refer to $\widetilde{G}$ as the *super-graph of* G. Figure 3.2 shows a graph G together with its super-graph $\widetilde{G}$.

Furthermore for each cluster $\mathcal{C}$ in the topological partition of T, we maintain a *full representation* defined as follows. An edge of G with both endpoints in a cluster $\mathcal{C}$ is said to be *internal* to $\mathcal{C}$, while an edge of G with only one endpoint in $\mathcal{C}$ is said to be *incident* to $\mathcal{C}$. Consider the subgraph induced by cluster $\mathcal{C}$ (i.e., the graph having only vertices and edges internal to $\mathcal{C}$). Call this subgraph $\mathcal{G}(\mathcal{C})$. The *full representation* of cluster $\mathcal{C}$ is a graph $\mathcal{F}(\mathcal{C})$, which consists of $\mathcal{G}(\mathcal{C})$ plus the clusters adjacent to $\mathcal{C}$ (shrunk into vertices) and the edges incident to $\mathcal{C}$. In other words, the vertex set of $\mathcal{F}(\mathcal{C})$ consists of the vertices internal to $\mathcal{C}$ and the clusters adjacent to $\mathcal{C}$. The edge set of $\mathcal{F}(\mathcal{C})$ is composed of all the edges of G internal and incident to $\mathcal{C}$. Since $\mathcal{C}$ contains at most $O(k)$ vertices, each of which has degree no greater than three, then $\mathcal{F}(\mathcal{C})$ has at most $O(k)$ vertices and edges and can be computed in $O(k)$ time. Figure 3.3 shows the full representation of a cluster.

Given the super-graph $\widetilde{G}$ and the full representation of each cluster in the topological partition, we use these data structures to test whether two vertices x and y are in the same 2-edge-connected component, as follows. Denote by $\mathcal{C}_x$ and $\mathcal{C}_y$ the clusters containing respectively vertices x and y. Notice that $\mathcal{C}_x$ and $\mathcal{C}_y$ may not be necessarily distinct. Replace in the super-graph $\widetilde{G}$ the vertices $\mathcal{C}_x$ and $\mathcal{C}_y$ with their full representation. That

initially given a non-null graph $G_0 = (V_0, E_0)$ and we allow $O(|V_0| + |E_0|)$ preprocessing time. In addition, our data structures can support insertions of new vertices and deletions of disconnected vertices.

We remark that the fully dynamic maintenance of the connected components of a graph differs substantially from the fully dynamic maintenance of the 2-edge-connected components. Indeed, in the first problem a single edge insertion can merge at most two components into one. On the other hand, for the second problem a single edge insertion can produce a cycle in the graph that might combine as many as $\Theta(n)$ 2-edge-connected components into one. As a result, by simply alternating the insertion and deletion of the same edge, we can generate a sequence of operations that at each step changes the number of components by $\Theta(n)$. This also explains why we do not maintain 2-edge-connected components' names: indeed in such a scenario a *DeleteEdge* operation would have to specify as many as $\Theta(n)$ different new names.

Our techniques combine a variety of graph properties, data structures and new algorithmic tools. Following the ideas of Frederickson [18], we partition graphs into vertex clusters. However, this by itself is not enough to achieve efficient algorithms for our problems. We find a way to maintain a succinct encoding of each cluster which satisfies nice properties with respect to 2-edge-connected components in the graph. This enables us to answer queries about edge connectivity properties without having to look at the entire graph each time.

The remainder of this chapter consists of five sections. In Section 2 we describe our data structure for connected graphs. Section 3 shows how to detect some particular kind of bridges in our graph. Section 4 deals with the implementation of our operations, while Section 5 analyzes their time complexity. Section 6 extends the same time and space bounds to unconnected graphs.

3.2 The Data Structure for Connected Graphs

For sake of simplicity, we first describe algorithms dealing with connected graphs, and we then show how to extend the algorithms to unconnected graphs. We will describe a data structure operating on graphs with bounded degree. It is well known (see for example references [18] and [23, page 132]) that each graph $G = (V, E)$ can be transformed into a graph $G' = (V', E')$, whose vertices have degree no greater than three and G' preserves the bridges of G. The transformation is as follows. For each vertex u of G of degree $d \geq 4$, where $v_0, v_1, \ldots, v_{d-1}$ are the vertices adjacent to u, replace u with new vertices $u_0, u_1, \ldots, u_{d-1}$ and replace edge (u, v_i) with edge (u_i, v_i), $0 \leq i \leq d-1$. Furthermore, add new edges $(u_i, u_{(i+1) \bmod d})$, $0 \leq i \leq d-1$. Notice that each edge (x, y) in G corresponds to a unique edge (x_i, y_j) in G' (but the converse is not always true since G' may contain the new edges of the form $(u_i, u_{(i+1) \bmod d})$, $0 \leq i \leq d-1$). It is easy to show that given a connected graph G with m edges and n vertices, G' contains at most $O(m)$ edges and vertices and can be generated in $O(m)$ time.

Figure 3.1: A graph and its biconnected and bridge-connected components

blocks) of a graph G are the components of G obtained after the removal of all the bridges. Two edges e' and e'' are in the same bridge-connected component if and only if either (i) $e' = e''$ or (ii) there is a (not necessarily simple) cycle (i.e., a cycle with no repetition of edges but with possible repetition of vertices) of G containing both e' and e''. Figure 3.1 shows a graph together with its biconnected and bridge-connected components.

The problem of computing the bridge-connected components of undirected graphs arise naturally in many applications and have been extensively studied. Tarjan [50] gives optimal linear-time sequential algorithms. Awerbuch and Shiloach [6] and Tarjan and Vishkin [55] give logarithmic-time parallel algorithms. Recently, Westbrook and Tarjan [57] 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 the inverse Ackermann's function. This bound is optimal for pointer algorithms [38, 51] as well as in the cell-probe model of computation [59].

In this chapter we study the problem of maintaining the 2-edge-connected components of a graph under an intermixed sequence of the following operations starting from the null graph (i.e., a graph with n vertices and no edges).

Same2EdgeBlock(u,v): Return *true* if vertices u and v are in the same 2-edge-connected component. Return *false* otherwise.

InsertEdge(x,y): Insert a new edge between the two vertices x and y.

DeleteEdge(x,y): Delete the edge between the two vertices x and y.

Motivations for studying this problem arise from the design and management of reliable networks.

As mentioned in [57], in such a fully dynamic setting there has been no known solution better than a repeated application of an off-line algorithm, which yields a single operation worst-case time complexity of $O(m)$, where m is the current number of edges in the graph. Reif [40] introduced the notion of *complete dynamic problems*, as a class of problems with the property that if one problem can be solved in $o(m)$ time per operation, then all the problems in the class can be solved in $o(m)$ time per operation. Reif mentioned as examples of complete dynamic problems the acceptance of a linear-time Turing machine, path systems, the Boolean circuit evaluation problem, unit resolution and depth first search numbering of a graph. These problems seem to have no better solution in response to dynamic changes than simply running a linear-time off-line algorithm. Westbrook and Tarjan [57] ask if there are sublinear-time algorithms for the fully dynamic maintenance of 2-edge-connected components or whether it is possible to show that this is a complete dynamic problem. We answer this question by showing that this problem can be solved in $O(m^{2/3})$ time per operation. When the graph is planar, the operations can be supported more efficiently in $O(\sqrt{n \log\log n}\,)$ time each. We obtain the same bounds even if we are

Chapter 3

Fully Dynamic Algorithms for 2-Edge Connectivity

In this chapter we study the problem of maintaining the 2-edge-connected components of a graph undergoing repeated dynamic modifications, such as edge insertions and edge deletions. We show how to test at any time whether two vertices belong to the same 2-edge-connected component, and how to insert and delete an edge in $O(m^{2/3})$ time in the worst case, where m is the current number of edges in the graph. This answers a question posed by Westbrook and Tarjan [57]. For planar graphs, we present algorithms that support all these operations in $O(\sqrt{n \log\log n}\,)$ worst-case time each, where n is the total number of vertices in the graph.

3.1 Introduction

An undirected graph $G = (V, E)$ is said to be *biconnected* if there is no vertex whose removal disconnects G: such a vertex is called an *articulation point*. It is well known [2] that a graph G is biconnected if and only if there are at least two vertex disjoint paths from each vertex to every other vertex. Given a graph $G = (V, E)$, let $E_1, E_2, \ldots, E_h$ be the partition of E into equivalence classes such that two edges e' and e'' are in the same equivalence class if and only if either (i) $e' = e''$ or (ii) there is a simple cycle (i.e., a cycle with no repetition of edges and vertices) of G containing both e' and e''. For $1 \leq i \leq h$ let V_i be the set of vertices that are endpoints of the edges in E_i. The subgraphs $G_i = (V_i, E_i)$ are called the *biconnected components* (or *blocks*) of G. An articulation point appears in more than one biconnected component. The set of edges of every graph can be partitioned into biconnected components in a unique way and every edge belongs exactly to one biconnected component. An edge contained in no cycle is in a biconnected component by itself and is referred to as a *bridge*. A graph with no bridges is called *2-edge-connected* (or *bridge-connected*). If the graph has bridges, then the removal of a bridge disconnects the graph. The *2-edge-connected components* (or *bridge-connected components* or *bridge-*

a maximal path from x to y after the insertion of edge (i,j), then the following equalities hold.

$$\delta(x,x) = 0, \forall x,y \in V; \tag{2.15}$$

$$\delta(x,y) = \max_{u \in V}\{\delta(x,u) + \delta(u,y)\}, \forall x,y \in V; \tag{2.16}$$

$$\delta_{new}(x,y) = \max\{\delta(x,y), \delta(x,i) + 1 + \delta(j,y)\}, \forall x,y \in V; \tag{2.17}$$

It is important to underline that equality (2.16) does not hold for cyclic graphs. Therefore results analogous to the ones shown in the previous sections are extendable only to directed acyclic graphs. In particular, Theorem 2 can be extended as follows.

Theorem 4 *Given a directed acyclic graph G, provided that new edge insertions do not introduce any cycles, there exists a data structure which supports each* length *operation in $O(1)$ time and each* maxpath *operation in $O(k)$ time, where k is the length of the traced (maximal) path. The total time required to maintain the data structure during the insertion of at most $O(n^2)$ edges is $O(n^3 \log n)$, where n is the number of vertices in the graph. The space complexity of the data structure is $O(n^2)$.*

We conclude this section by mentioning that also Theorem 3 can be extended to longest simple paths on directed acyclic graphs.

now a packet of operations λ_j, $2 \leq j \leq \frac{n}{3}$, in terms of the two operations μ_j and ν_j as follows:

$$\lambda_j = \mu_j \underbrace{\nu_j \nu_j \ldots \nu_j}_{C-1}.$$

Perform the sequence of operations $\Lambda = \lambda_2, \lambda_3, \ldots, \lambda_{\frac{n}{3}}$. Notice that the total number of operations in Λ is $O(Cn)$. Each packet λ_j forces $(n/3+1)(2n/3-j+1)C$ changes in the matrix D. Hence the total number of updates required in D is

$$\sum_{j=2}^{n/3} \left(\frac{n}{3}+1\right)\left(\frac{2}{3}n-j+1\right)C,$$

which is $\Omega(Cn^3)$.

We notice that the worst-case per operation time complexity of our data structure remains $O(n^2)$. As the above example shows, this is also the total number of updates that may be required in the matrix D during either the insertion of a new edge or an edge-cost decrease. For the special case of planar graphs, we achieve a $\Theta(Cn^3)$ bound for our problem.

2.6 On-Line Maintenance of Longest Paths in Directed Acyclic Graphs

A modification of our algorithms is able to perform *maxpath* operations (i.e., queries about longest simple paths) on directed acyclic graphs where the insertions of new edges do not introduce cycles. This is not a significant restriction, since the longest path problem is known to be NP-complete for arbitrary graphs [22], while a polynomial time algorithm exists for directed acyclic graphs [36].

We now sketch how to handle longest paths in directed acyclic graphs with unit edge costs. Following the ideas outlined in the previous sections, we maintain the descendants and ancestors of each vertex as trees of forward and backward *maximal* length paths. The operation *length* can be performed as before, while *maxpath* can be performed as the *minpath* operation given above. In addition, simultaneous queries about minimal and maximal length paths can be supported by maintaining both the trees of minimal and maximal length, thus only at the price of doubling the space required per vertex.

In case of maximal length simple path in directed acyclic graphs, equalities analogous to (2.1), (2.2) and (2.3) hold. In fact denoting by $\delta(x,y)$ the length of a maximal path from x to y ($\delta(x,y) = -\infty$ if there is no path from x to y), and by $\delta_{new}(x,y)$ the length of

during any sequence $\mathcal{S}$ of operations of the extended repertoire is bounded by the total number of benign pairs plus the total number of malignant pairs during the edge insertions and edge-cost decreases of $\mathcal{S}$.

We start by bounding the total time spent because of benign pairs. We observe that for each pair $\langle x, y\rangle$ that is benign during an update operation σ, the entry (x, y) of the distance matrix D is decreased at least by 1 during σ. Since for each edge (i, j) we can have one edge insertion and at most C edge-cost decreases, there can be at most $O(Cn)$ updates of each entry (x, y) of the matrix D during any sequence of operations. As a result, the total time spent because of benign pairs is $O(Cn^3)$. In order to bound the total time spent because of malignant pairs, we use the following credit policy:

When performing an operation σ that either inserts or decreases the cost of an edge (i, j), for each benign pair $\langle x, y\rangle$ during σ give $4\left\lceil \frac{Cn}{d(x,y)} \right\rceil$ credits both to x and y, where $d(x, y) \geq 1$ is the minimum distance from x to y after operation σ.

Using the same ideas in the proof of Theorem 2, the total number of credits given during any sequence $\mathcal{S}$ of operations is

$$O\left(\sum_{x\in V}\sum_{y\in V}\sum_{d=1}^{Cn}\left\lceil\frac{Cn}{d}\right\rceil\right) \leq O(Cn^3\log(nC)).$$

We have to show now that these credits are enough to pay for the work done for malignant pairs. Let us consider any update operation σ which either inserts an edge (i, j) or decreases the cost of an edge (i, j), and let $\langle x, y\rangle$ be any malignant pair during σ. By applying the same argument as in the proof of Theorem 2, it can be shown that at least $2n$ credits are given to both x and y during operation σ. These credits are sufficient to pay for the work of all malignant pairs during σ of the form $\langle x, *\rangle$ and $\langle *, y\rangle$. This gives the theorem. □

The bound given in Theorem 3 is only a factor of $\log(nC)$ away from the best possible bound for the problem of maintaining on-line a solution to the All Pairs Shortest-Paths Problem with integer edge costs in $[1 \ldots C]$. In fact we show now that there exist sequences of *add* and *decrease* operations that force as many as $\Omega(Cn^3)$ changes in the distance matrix D. Consider once again the graph $G = (V, E)$ given in Figure 2.3, with vertex set

$$V = \{s_1, s_2, \ldots, s_{\frac{n}{3}}, x_1, x_2, \ldots, x_{\frac{n}{3}}, t_1, t_2, \ldots, t_{\frac{n}{3}}\},$$

where n is a multiple of 3, and edges (s_i, x_1), $i = 1, \ldots, \frac{n}{3}$, (x_i, x_{i+1}), $i = 2, \ldots, \frac{n}{3} - 1$, and $(x_{\frac{n}{3}}, t_i)$, $i = 1, 2, \ldots, \frac{n}{3}$, each of which has cost C. For $2 \leq j \leq \frac{n}{3}$ denote by μ_j operation *add*(x_1, x_j, C). Similarly, for $2 \leq j \leq \frac{n}{3}$ denote by ν_j operation *decrease*$(x_1, x_j, 1)$. Define

by the data structure is $O(n^2)$. In this case too, there are sequences of edge insertions that force $\Omega(Cn^3)$ changes in the distance matrix D. Once again, our algorithm is only a logarithmic factor away from the best possible bound.

Denoting by $w(i,j)$ the cost of edge (i,j), we extend the repertoire of allowable operations as follows. Operations *length* and *minpath* are unchanged. Operation *add*(i,j,w) inserts the edge (i,j) with integer cost w, $1 \leq w \leq C$. We consider also a *decrease*(i,j,Δ) operation that decreases the cost $w(i,j)$ of edge (i,j) by the positive integer Δ $(0 < \Delta < w(i,j))$. Notice that *add*(i,j,w) is a special case of a *decrease* operation; indeed, *add*(i,j,w) can be thought of as decreasing the cost of edge (i,j) from $+\infty$ to w. As a result, if we take care of edge costs this extended set of operations can be supported by means of the same data structures and procedures presented in section 3. The only difference is that now trees of forward and backward shortest paths are maintained throughout any sequence of operations of the extended repertoire. Obviously we must modify the lines 4 and 9 of the procedure UpdateForward by substituting $w(i,j)$ in place of 1, where $w(i,j)$ is the new cost of edge (i,j) after either the insertion of (i,j) or a *decrease* operation on (i,j). In order to prove the correctness we observe that equalities (2.1) and (2.2) still hold and that equality (2.3) must be replaced by the following, which takes care of the new cost $w(i,j)$:

$$d_{new}(x,y) = \min\{d_{old}(x,y), d_{old}(x,i) + w(i,j) + d_{old}(j,y)\}, \qquad \forall x,y \in V. \quad (2.14)$$

The correctness is now an extension of Theorem 1. The total time required to maintain the data structure during any sequence $\mathcal{S}$ of (at most $O(Cn^2)$) edge insertions and edge-cost decreases is $O(Cn^3 \log(nC))$, as the following theorem shows.

Theorem 3 *There exists a data structure which supports each* length *operation in* $O(1)$ *time and each* minpath *operation in* $O(k)$ *time, where* k *is the length of the traced (minimal) path. The total time required to maintain the data structure during the insertion of at most* $O(n^2)$ *edges and during at most* $O(Cn^2)$ decrease *operations is* $O(Cn^3 \log(nC))$, *where* n *is the number of vertices in the graph. The space complexity of the data structure is* $O(n^2)$.

Proof: We proceed along the lines of the proof of Theorem 2. We notice that during any update operation σ (either an edge insertion or an edge-cost decrease) it is still possible to define benign and malignant pairs. There is only a change in terminology, due to the fact that now the cost of the same edge (i,j) can be decreased many times by different *decrease* operations. Therefore, while for unit edge costs only one operation in a given sequence could deal with a given edge (i,j), now there can be many operations in a given sequence dealing with edge (i,j) (i.e., either inserting or decreasing the cost of edge (i,j)). As a consequence, we now define benign and malignant pairs *during the update operation* σ. In a similar vein, it is possible to extend Lemma 4 and Lemma 5 to the new setting of operations. Because of Lemma 4, the total time required to maintain the data structure

Figure 2.3: The graph used to prove the lower bound

$$\leq \quad O(n^3 \log n).$$

□

The bound given in Theorem 2 is only a factor of $\log n$ away from the best possible bound for the problem of maintaining on-line a solution to the All Pairs Minimal Length Paths Problem. In fact we show now that there exist sequences of *add* operations that force as many as $\Omega(n^3)$ changes in the distance matrix D. Consider the graph $G = (V, E)$ given in Figure 2.3, with vertex set

$$V = \{s_1, s_2, \ldots, s_{\frac{n}{3}}, x_1, x_2, \ldots, x_{\frac{n}{3}}, t_1, t_2, \ldots, t_{\frac{n}{3}}\},$$

where n is a multiple of 3, and edges (s_i, x_1), $i = 1, 2, \ldots, \frac{n}{3}$, (x_i, x_{i+1}), $i = 2, \ldots, \frac{n}{3} - 1$, and $(x_{\frac{n}{3}}, t_i)$, $i = 1, 2, \ldots, \frac{n}{3}$. If now the $(n/3 - 1)$ edges

$$(x_1, x_i), \quad i = 2, \ldots, \frac{n}{3},$$

are inserted into G by increasing i, the changes in the distance matrix D can be computed as follows. The insertion of edge (x_1, x_i), $2 \leq i \leq \frac{n}{3}$, creates new shorter paths from vertex s_j, $1 \leq j \leq \frac{n}{3}$, and from vertex x_1 to vertices x_h, $i \leq h \leq \frac{n}{3}$, and t_k, $1 \leq k \leq \frac{n}{3}$. This gives a total of $(n/3 + 1)(2n/3 - i + 1)$ changes required in the distance matrix D. After inserting all the edges, the total number of updates required in D is therefore

$$\sum_{i=2}^{n/3} \left(\frac{n}{3} + 1\right) \left(\frac{2}{3}n - i + 1\right),$$

which is $\Omega(n^3)$.

Notice that the graph shown in Figure 2.3 is planar, and remains planar after the edge insertions. This implies that there can be as many as $\Omega(n^3)$ updates required in the matrix D also for planar graphs. Since a planar graph can have at most $O(n)$ edges, and the worst-case per operation time complexity of our data structure is $O(n^2)$, we achieve a $\Theta(n^3)$ total time bound for planar graphs.

2.5 Integer Edge Costs

The data structure presented in section 3 can be adapted to deal with integer edge costs in the range $[1 \ldots C]$. The total time required to maintain on-line a solution to the All Pairs Shortest-Paths Problem during (at most $O(n^2)$) insertions of edges and (at most $O(Cn^2)$) edge-cost decreases is $O(Cn^3 \log(nC))$ in the worst-case. The space required

the lengths of π_x and π_y. Notice that the case $d_x = d_y = 0$ cannot happen, since this would imply $\langle x, y\rangle = \langle i, j\rangle$ and therefore $\langle x, y\rangle \notin M_{i,j}$. As a result, $d_x + d_y > 0$. Because of Lemma 5, for any vertex $u \neq x$ in π_x and $v \neq y$ in π_y, the pairs $\langle u, y\rangle$ and $\langle x, v\rangle$ are benign for (i, j). The total number of credits given to vertex x because of vertices in π_y is

$$4 \sum_{h=d_x+1}^{d_x+d_y} \left\lceil \frac{n}{h} \right\rceil \geq 4n \log_e \left(\frac{d_x + d_y + 1}{d_x + 1} \right).$$

Similarly, the total number of credits given to vertex y because of vertices in π_x is

$$4 \sum_{h=d_y+1}^{d_x+d_y} \left\lceil \frac{n}{h} \right\rceil \geq 4n \log_e \left(\frac{d_x + d_y + 1}{d_y + 1} \right).$$

As a consequence, the number of credits given to both x and y during the insertion of edge (i, j) is bounded below by

$$4n \log_e \frac{(d_x + d_y + 1)^2}{(d_x + 1)(d_y + 1)}.$$

Since for d_x and d_y positive integers such that $d_x + d_y > 0$ we have that

$$\frac{(d_x + d_y + 1)^2}{(d_x + 1)(d_y + 1)} \geq 2$$

then the credits given to x and y are bounded below by $4n \log_e 2 > 2n$. The credits given to both x and y can henceforth pay for the work due to all the malignant pairs of the form $\langle x, *\rangle$ and $\langle *, y\rangle$ which are at most $2n$. Since this argument can be repeated for any malignant pair, it proves that $\Phi_{i,j} \geq |M_{i,j}|$. Namely, all the credits given during operation $add(i, j)$ are able to pay for the overall work due to malignant pairs during the same operation.

To conclude our proof, we notice that because of equations (2.11), (2.12) and (2.13), the total time spent while updating the data structure during any sequence $\mathcal{S}$ of edge insertions is

$$\begin{aligned} O\left(\sum_{(i,j)\in\mathcal{S}} (|B_{i,j}| + |M_{i,j}|) \right) &= O\left(\sum_{(i,j)\in\mathcal{S}} |B_{i,j}| \right) + O\left(\sum_{(i,j)\in\mathcal{S}} |M_{i,j}| \right) \\ &\leq O(n^3) + O\left(\sum_{(i,j)\in\mathcal{S}} \Phi_{i,j} \right) \end{aligned}$$

Proof: The correctness of *length* and *add* operations derives from Theorem 1, while the data structure correctly supports *minpath* operations because of Theorem 1 and Lemma 1. The space required is $O(n^2)$ since three $n \times n$ matrices and $2n$ trees, each of size at most n, are used. The time required by procedure *length*(x, y) is $O(1)$ since it accesses entry (x, y) of the distance matrix D. As previously noticed, procedure *minpath*(x, y) takes $O(k)$ to return a minimal path from x to y (if one exists), where k is the length of the achieved minimal path.

We have now to compute the total time required to maintain the data structure during any sequence $\mathcal{S}$ of edge insertions. Because of Lemma 4, this time is bounded by the total number of benign pairs plus the total number of malignant pairs during the insertions of edges:

$$O\left(\sum_{(i,j)\in\mathcal{S}} (|B_{i,j}| + |M_{i,j}|)\right). \tag{2.11}$$

During the insertion of edge (i, j), for each pair $\langle x, y\rangle$ in $B_{i,j}$ the entry (x, y) of the distance matrix D is either changed from $+\infty$ to a positive integer value $\ell \leq n - 1$ or decreased at least by 1. Since each entry of D can hold only the values $1, 2, \ldots n - 1, +\infty$ and there are at most $O(n^2)$ different entries, the total time spent because of benign pairs is

$$\sum_{(i,j)\in\mathcal{S}} |B_{i,j}| \leq O(n^3). \tag{2.12}$$

In order to bound the total time spent because of malignant pairs, we use an amortization argument based upon the credit technique by Tarjan [53]. Our credit policy is the following:

When inserting edge (i, j), for each benign pair $\langle x, y\rangle$ in $B_{i,j}$ give $4\left\lceil \frac{n}{d(x,y)} \right\rceil$ credits both to x and y, where $d(x, y) \geq 1$ is the minimum distance from x to y after the insertion of (i, j).

Denote by $\Phi_{i,j}$ the number of credits given during the insertion of edge (i, j). Since we give credits to a pair of vertices if and only if it is a benign pair, their distance strictly decreases. Therefore, the total number of credits given during any sequence $\mathcal{S}$ of edge insertions is

$$\sum_{(i,j)\in\mathcal{S}} \Phi_{i,j} \leq O\left(\sum_{x\in V}\sum_{y\in V}\sum_{d=1}^{n-1} \left\lceil \frac{n}{d} \right\rceil\right) \leq O(n^3 \log n). \tag{2.13}$$

To complete our proof, we have to show that the credits given are enough to pay for the work done because of malignant pairs. We will show something stronger, namely that all the credits $\Phi_{i,j}$ given to benign pairs while inserting edge (i, j) are enough to pay for the work done because of malignant pairs during the insertion of the same edge (i, j).

Consider any malignant pair $\langle x, y\rangle$ in $M_{i,j}$. Denote by π_x the path in $ANC(i)$ from x to i and by π_y the path in $DESC(j)$ from j to y. Let $d_x \geq 0$ and $d_y \geq 0$ be, respectively,

Figure 2.2: On the proof of Lemma 5

Lemma 5 *For any pair $\langle x, y\rangle$ in $M_{i,j}$, denote by π_x the path from x to i in* ANC(i) *and by π_y the path from j to y in* DESC(j). *For any vertex $u \neq x$ in π_x and for any vertex $v \neq y$ in π_y then the pairs $\langle u, y\rangle$ and $\langle x, v\rangle$ are benign for (i, j).*

Proof: We proceed by contradiction. Consider first a malignant pair $\langle x, y\rangle$ and assume that there exists a vertex $u \neq x$ in π_x such that also the pair $\langle u, y\rangle$ is malignant (see Figure 2.2). This means that vertices in the subtree rooted at y in *DESC*(j) are not included in the pruned tree N given to the ancestors of u on line 11 of procedure *UpdateForward*. Since $T(x) \subseteq N$ because of line 11 in procedure *UpdateForward* then $y \notin T(x)$, which is clearly a contradiction. To prove the second part of the lemma, consider a malignant pair $\langle x, y\rangle$ and assume that there exists a vertex $v \neq y$ in π_y such that also the pair $\langle x, v\rangle$ is malignant. This means that the vertices in the subtree rooted at v in *DESC*(j) are not visited while updating *DESC*(x), because of line 4 in procedure *UpdateForward*. Therefore $y \notin T(x)$, again a contradiction. □

The following theorem proves the correctness and analyzes the complexity of our approach.

Theorem 2 *There exists a data structure that correctly supports each* length *operation in $O(1)$ time and each* minpath *operation in $O(k)$ time, where k is the length of the traced (minimal) path. The total time required to maintain the data structure during the insertion of at most $O(n^2)$ edges is $O(n^3 \log n)$, where n is the number of vertices in the graph. The space complexity of the data structure is $O(n^2)$.*

$$\begin{aligned} d(x,y) &\leq d(x,u)+d(u,v)+d(v,y) \\ &\leq d(x,u)+d(u,i)+1+d(j,v)+d(v,y). \end{aligned}$$

Because of the induction step $d(j,v)+d(v,y) = D[j,v]+D[v,y]$ and by Lemma 2 (remember that y is contained in the subtree of $DESC(j)$ rooted at v), we have that $D[j,v]+D[v,y] = D[j,y]$. Applying again the induction step to the last equality, we derive $d(j,v)+d(v,y) = d(j,y)$. Similarly, we obtain $d(x,u)+d(u,i) = d(x,i)$. Therefore, $d(x,y) \leq d(x,i)+1+d(j,y)$. Applying equality (2.3), we get $d_{new}(x,y) = \min\{d(x,y), d(x,i)+1+d(j,y)\} = d(x,y)$. Thus, $D_{new}[x,y] > d_{new}(x,y) = d(x,y) = D[x,y]$. This is clearly a contradiction since lines 4 and 9 of procedure *UpdateForward* enforce that entries in the matrix D can only be decreased.

Since it cannot be either $D_{new}[x,y] < d_{new}(x,y)$ or $D_{new}[x,y] > d_{new}(x,y)$ for any pair of vertices, equality (2.10) must hold. □

Before analyzing the time and space complexity of the data structure, we need some new terminology. During the insertion of an edge (i,j) in the graph G, for any x ancestor of i let us denote by $T(x)$ the set of vertices in $DESC(j)$ visited by *UpdateForward* while updating the tree $DESC(x)$. Note that as a special case, $T(x)$ might be empty. We say that a pair of vertices $\langle x,y\rangle$ is *benign for edge* (i,j) if x is an ancestor of i, y is in $T(x)$, and before the insertion of (i,j) $D[x,y] > D[x,i]+1+D[j,y]$ (i.e., (i,j) introduces a new shorter path from x to y). Furthermore, we say that a pair of vertices $\langle x,y\rangle$ is *malignant for edge* (i,j) if x is an ancestor of i, y is in $T(x)$, and before the insertion of (i,j) $D[x,y] \leq D[x,i]+1+D[j,y]$ (i.e., (i,j) introduces no new shorter path from x to y). Denote by $B_{i,j}$ the set of pairs which are benign for (i,j) and by $M_{i,j}$ the set of pairs which are malignant for (i,j). Since edge (i,j) can be inserted at most once in the graph, we have by definition that $\langle i,j\rangle \notin M_{i,j}$, and $\langle i,j\rangle \in B_{i,j}$.

Lemma 4 *The total time required by procedure* add *to insert an edge* (i,j) *is* $O(|B_{i,j}|+|M_{i,j}|)$.

Proof: It is easy to verify that the time required to insert edge (i,j) is proportional to the total numbers of vertices inserted in the queue Q during subsequent calls of procedure *UpdateForward*. Since at the end of the algorithm Q is empty, the number of vertices put in Q is equal to the number of vertices removed from Q. But when a vertex y is removed from Q during a call to *UpdateForward*(x,i,j,T) then either $D[x,i]+1+D[j,y] < D[x,y]$, which implies that $\langle x,y\rangle$ is benign for (i,j), or $D[x,i]+1+D[j,y] \geq D[x,y]$, which implies that $\langle x,y\rangle$ is malignant for (i,j). Consequently, the total time required to insert edge (i,j) is $O(|B_{i,j}|+|M_{i,j}|)$. □

We now characterize a property of benign and malignant pairs.

Combining (2.7) and (2.8) gives inequality (2.5). □

The correctness of our approach hinges on the following theorem.

Theorem 1 *After each operation, for the data structure the following equality holds*

$$D[x,y] = d(x,y) \qquad \forall x,y \in V. \tag{2.9}$$

That is, given any two vertices x and y, the entry (x,y) of the matrix D correctly gives the length of the minimal path from x to y.

Proof: We proceed by induction on the number of operations performed. Since *minpath* and *length* operations do not modify the data structure, we consider only *add* operations.

The basis of the induction is true since at the beginning the graph contains no edges, $D[x,y] = d(x,y) = +\infty$ for any two vertices x and y ($x \neq y$), and $D[x,x] = d(x,x) = 0$ for each vertex x.

Suppose equality (2.9) holds before the insertion of an edge (i,j). Let us denote by $D_{new}[x,y]$ and by $d_{new}(x,y)$ respectively the values of $D[x,y]$ and $d(x,y)$ after inserting the edge (i,j). We want to prove that given $D[x,y] = d(x,y)$

$$D_{new}[x,y] = d_{new}(x,y) \qquad \forall x,y \in V. \tag{2.10}$$

The case $D_{new}[x,y] < d_{new}(x,y)$ cannot happen since by lines 4 and 9 in procedure *UpdateForward* and by the induction step :

$$\begin{aligned} D_{new}[x,y] &\geq \min\{D[x,y], D[x,i]+1+D[j,y]\} \\ &= \min\{d(x,y), d(x,i)+1+d(j,y)\} \\ &= d_{new}(x,y). \end{aligned}$$

Suppose now there exists a pair of vertices $x,y \in V$ such that $D_{new}[x,y] > d_{new}(x,y)$. This implies that the distance from x to y has decreased after the insertion of the edge (i,j) but *UpdateForward* did not update the entry $D[x,y]$. Namely, $D_{old}[x,i]+1+D_{old}[j,y] < D_{old}[x,y]$, $d_{new}(x,y) = D_{old}[x,i]+1+D_{old}[j,y]$, and $D_{new}[x,y] = D_{old}[x,y]$. This can only happen if and only if x is an ancestor of i, y is a descendant of j and y was not inserted in the queue Q during the execution of the procedure *UpdateForward*$(x,i,j,DESC(j))$. Indeed, if y was inserted in the queue Q, then line 9 of procedure *UpdateForward* would have updated correctly $D[x,y]$. As a consequence, there must be two vertices u and v such that x is contained in the subtree of $ANC(i)$ rooted at u and y is contained in the subtree of $DESC(j)$ rooted at v, and for which procedure *UpdateForward*$(u,i,j,DESC(j))$ pruned at v the tree considered because the condition on line 4 was not satisfied. Hence, $D[u,v] \leq D[u,i]+1+D[j,v]$ or, what is the same for the induction step, $d(u,v) \leq d(u,i)+1+d(j,v)$. Using this inequality yields

considered for possible insertion in $DESC(x)$ because of line 4). As a consequence of the second induction step $D[x,v] = D[x,i] + 1 + D[j,v]$ is equal to the depth of v in the tree $DESC(x)$. Since $D[x,y]$ will be set to $D[x,i] + 1 + D[j,y]$ (line 9) and $D[j,y] = D[j,v] + 1$ because of the first induction step on tree $DESC(j)$, then $D[x,y] = D[x,i] + 1 + D[j,v] + 1$ will be equal to the depth of y in the tree $DESC(x)$. Since for vertices y not in $DESC(x)$ after the update the entry $D[x,y]$ is not changed, this concludes the two induction steps and gives the lemma. □

Lemma 1 shows that if the distance matrix D is correctly updated, then procedure *minpath* correctly returns a path with minimal number of edges. We recall that we denote by $d(x,y)$ the length of a minimal path between vertices x and y in the graph.

Lemma 2 *Assume that $D[a,b] = d(a,b)$ for any pair of vertices a and b in the graph. Then for any pair of vertices j and v such that v is reachable from j, and for any vertex y contained in the subtree of* DESC(j) *rooted at v, after each operation the following equality holds.*

$$D[j,y] = D[j,v] + D[v,y]. \tag{2.4}$$

Proof: We proceed by contradiction. If $D[j,y] > D[j,v] + D[v,y]$, then since $d(a,b) = D[a,b]$ for any pair of vertices a and b, we have $d(j,y) > d(j,v) + d(v,y)$ which contradicts equality (2.2). If $D[j,y] < D[j,v] + D[v,y]$, then $d(v,y) = D[v,y] > D[j,y] - D[j,v]$. Since by Lemma 1 and by the hypothesis $D[a,b] = d(a,b)$ the path from v to y in $DESC(j)$ is a path in the graph of length $D[j,y] - D[j,v]$, this contradicts the fact that $d(v,y)$ is the length of a minimal path from v to y. Since it can be neither $D[j,y] > D[j,v] + D[v,y]$ nor $D[j,y] < D[j,v] + D[v,y]$, then $D[j,y] = D[j,v] + D[v,y]$. □

Lemma 3 *Assume that $D[a,b] = d(a,b)$ for any pair of vertices a and b in the graph before inserting an edge from i to j. Then given any pair of vertices u and v such that $d(u,v) \leq d(u,i) + 1 + d(j,v)$, for any vertex y contained in the subtree of* DESC(j) *rooted at v and for any vertex x contained in the subtree of* ANC(i) *rooted at u, the following two inequalities hold.*

$$d(u,y) \leq d(u,i) + 1 + d(j,y) \tag{2.5}$$

$$d(x,v) \leq d(x,i) + 1 + d(j,v) \tag{2.6}$$

Proof: We prove only inequality (2.5). The proof for inequality (2.6) is completely analogous and therefore it has been omitted. Consider any vertex y in the subtree of $DESC(j)$ rooted at v. Due to equality (2.2) and the hypothesis

$$d(u,y) \leq d(u,v) + d(v,y) \leq d(u,i) + 1 + d(j,v) + d(v,y). \tag{2.7}$$

Since y is in the subtree of $DESC(j)$ rooted at v, then by Lemma 2 $D[j,y] = D[j,v] + D[v,y]$. Since $D[a,b] = d(a,b)$ for any pair of vertices a and b in the graph, this is equivalent to saying

$$d(j,y) = d(j,v) + d(v,y). \tag{2.8}$$

The updates of the trees of minimal backward paths can be carried out in a completely analogous fashion. In particular, the procedure *UpdateBackward* that updates the trees $ANC(y)$ for each y descendant of vertex j, can be obtained by interchanging the roles of the trees $DESC$ and ANC and of the matrices $FORWARD$ and $BACKWARD$ and by neglecting line 9. Alternatively, the updates to the ancestor and descendant trees can be done at the same time in the while loop of procedure UpdateForward, since each descendant change corresponds to an ancestor change.

2.4 Correctness and Time Complexity

In this section, we analyze the time complexity of the data structure and prove its correctness. Before stating our results, we need few technical lemmas.

Lemma 1 *After each operation and for any pair of vertices x and y in the graph, $D[x,y]$ is equal to the depth of vertex y in the tree* DESC(x)*, if y is a descendant of x. Otherwise, y is not in* DESC(x) *and $D[x,y] = +\infty$.*

Proof: We proceed by induction on the number of operations performed. At the beginning, the lemma is true since $D[x,y] = +\infty$ for $x \neq y$, $D[x,x] = 0$ and $DESC(x) = \{x\}$, for any $x \in V$. Assume now that the lemma holds before the i-th operation. If the i-th operation is either a *length* or a *minpath* operation, then the lemma still holds afterwards because these two operations do not modify the data structure.

Assume that the i-th operation is $add(i,j)$. We first notice that the algorithm that updates the data structures after each edge insertion terminates. Indeed procedure *UpdateForward*(x,i,j,T) is called each time with a different vertex x in $ANC(i)$. Therefore there will be at most $|ANC(i)|$ recursive calls of this procedure and the algorithm terminates. Now we show that, after performing this operation, for any vertex x the x-th row of matrix D still satisfies the lemma. We have now two cases, depending on whether the x-th row was modified or not. It can be shown that the x-th row of matrix D is not modified if and only if either x is not an ancestor of i (and therefore no procedure *UpdateForward*(x,i,j,T) will be executed) or (i,j) does not create a new shorter path between x and j (and therefore the execution of procedure *UpdateForward*(x,i,j,T) does nothing since $D[x,j] \leq D[x,i]+1$). In this case, the lemma will hold afterwards. Otherwise, if the x-th row of D has been modified, we proceed by using a second induction, this time on the number of elementary operations performed to $DESC(x)$ (i.e., the number of vertices inserted in $DESC(x)$). The basis of the second induction is satisfied since j is inserted into $DESC(x)$ on line 6 of procedure *UpdateForward* as a child of i. This implies that j is correctly inserted at depth $D[x,j] = D[x,i]+1$ in $DESC(x)$ because of the first induction step. Assume now that all the insertions in the tree $DESC(x)$ before inserting a new vertex $y \neq j$ in it satisfy the lemma. Because of line 7 of procedure UpdateForward, y is inserted in $DESC(x)$ as a child of vertex v, where v is the parent of y in the tree $DESC(j)$. Furthermore, v must have been inserted in $DESC(x)$ by procedure *UpdateForward* (otherwise vertex y will not be

occurrence of y in the tree is implicitly deleted. With the help of the *FORWARD* and *BACKWARD* matrices this task can be accomplished in constant time and therefore will not affect the asymptotic time spent during the update. At the end of this step, all the children of y in T are examined in a recursive fashion.

(ii) When a vertex y in T for which $D[x,i] + 1 + D[j,y] \geq D[x,y]$ is reached, then no shorter path from x to y was introduced by the insertion of the edge (i,j). In this case no update is performed in $DESC(x)$ and the tree T is pruned by cutting the subtree rooted at y. In fact, for each vertex y' in such a subtree the new arc (i,j) does not decrease the distance from x to y'.

After $DESC(x)$ has been updated, the pruned copy of T is passed to the children of x in $ANC(i)$. The update of the data structure is performed by procedure *UpdateForward*(x,i,j,T), where (i,j) is the inserted arc, x is an ancestor of i, and T is a subtree of $DESC(j)$ containing the nodes possibly used to update $DESC(x)$. The first call to the procedure is *UpdateForward*$(i,i,j,DESC(j))$. The remaining updates are performed by recursive calls to *UpdateForward*. Notice that instead of pruning the tree T, the procedure copies the useful part of it in a local variable N.

```
        procedure UpdateForward (vertices : x,i,j ; tree : T) ;
        vertices : y,w; queue : Q; tree : N;
        begin
1.        Q := {j}; N := ∅;
2.        while Q ≠ ∅ do begin
3.          remove an item y from the queue Q;
4.          if D[x,i] + 1 + D[j,y] < D[x,y] then begin
5.                if y = j then begin
6.                      insert j in DESC(x) as a child of i; N := {j};
                        update FORWARD[x,j] accordingly ;
                    end
                  else begin
7.                      insert y in DESC(x) with the same parent it has in DESC(j);
8.                      insert y in N with the same parent it has in DESC(j);
                        update FORWARD[x,y] accordingly ;
                    end
9.                D[x,y] := D[x,i] + 1 + D[j,y];
10.               for each w child of y in T
                    do insert w into the queue Q;
              end;
            end;
11.         if N ≠ ∅ then for each edge (x,u) in ANC(i)
              do UpdateForward(u,i,j,N) ;
        end;
```

We now show how to update the distance matrix D, the two matrices $FORWARD$ and $BACKWARD$, and the $2n$ trees $DESC(v)$ and $ANC(v)$, after the insertion of a new edge (i,j). Denote by $d(x,y)$ the length of a minimal path from vertex x to vertex y in the graph G (hereafter referred to also as minimal length function). The following properties hold:

$$d(x,x) = 0, \forall x \in V. \tag{2.1}$$

$$d(x,y) = \min_{u \in V}\{d(x,u) + d(u,y)\}, \forall x,y \in V. \tag{2.2}$$

Furthermore, if $d_{new}(x,y)$ denotes the minimal length function after the insertion of an edge (i,j) and $d_{old}(x,y)$ denotes the minimal length function before the insertion of (i,j), then the following property must be true:

$$d_{new}(x,y) = \min\{d_{old}(x,y), d_{old}(x,i) + 1 + d_{old}(j,y)\}, \forall x,y \in V. \tag{2.3}$$

Equation (2.3) states that the insertion of an edge (i,j) can introduce new shorter paths only from ancestors x of vertex i (for which $d(x,i) < +\infty$) to descendants y of vertex j (for which $d(j,y) < +\infty$). In this case, the trees of minimal forward paths might have to be updated for vertices in $ANC(i)$ and the trees of minimal backward paths might have to be updated for vertices in $DESC(j)$. We now describe how to update trees of minimal forward paths because of the insertion of an edge (i,j). A similar argument can be applied to the trees of minimal backward paths. In the update of the trees of minimal forward paths, a crucial role is played by the two trees $ANC(i)$ and $DESC(j)$: in order to guarantee a correct update of the forward trees, it is sufficient that for each vertex x in $ANC(i)$ the tree $DESC(x)$ is updated using $DESC(j)$.

However, to improve the time required for updating the data structure we need a more sophisticated approach. Our algorithm can be described as follows. We start from vertex i and update both the i-th row of the matrix D and the tree $DESC(i)$ by visiting the tree $DESC(j)$. The details of this update will be given later on. At the end of the update of $DESC(i)$, a tree T contained in $DESC(j)$ is defined which includes only vertices v for which the distance from i to v has decreased after the insertion of edge (i,j). This tree T is given to the children of i in $ANC(i)$ which will perform the updates of their forward trees of minimal paths in a recursive fashion. In other words, each vertex x in $ANC(i)$ starting from vertex i receives a tree T that is a (possibly pruned) copy of $DESC(j)$. After receiving T, vertex x updates $DESC(x)$ using T. At the same time, x performs a further pruning of T and passes it to its children in $ANC(i)$.

The update of $DESC(x)$ for each vertex x in $ANC(i)$ is carried out as follows. Vertex x visits the tree T it receives, starting from the root of T. The subsequent update of $DESC(x)$ and the pruning of T are subject to the following two rules.

(i) When a vertex y in T for which $D[x,i] + 1 + D[j,y] < D[x,y]$ is reached, then a shorter path from x to y has been found. In this case, vertex y is inserted into $DESC(x)$ either as a child of i if $y = j$ or as a child of the same parent it has in $DESC(j)$ otherwise. By inserting a vertex y into a tree, we mean that any previous

rooted at v and $ANC(v)$ as a tree of backward minimal paths rooted at v. Moreover, we use two $n \times n$ matrices of pointers, named $FORWARD$ and $BACKWARD$ defined as follows. $FORWARD[u,v]$ contains a pointer to vertex v in the tree of minimal forward paths $DESC(u)$ if vertex v is a descendant of vertex u. Otherwise, $FORWARD[u,v]$ contains a special *null* value. Similarly, $BACKWARD[u,v]$ contains a pointer to vertex v in the tree of minimal backward paths $ANC(u)$ if vertex v is an ancestor of vertex u. Otherwise, $BACKWARD[u,v]$ contains a special *null* value. All of these data structures can be initialized in $O(n^2)$ time.

```
procedure initialize;
vertices : x, y;
begin
   for each x ∈ V do begin
      for each y ∈ V do begin
         FORWARD[x, y] := null;
         BACKWARD[x, y] := null;
         D[x, y] := +∞ ;
      end ;
      DESC(x) := {x} ;
      ANC(x) := {x} ;
      D[x, x] := 0 ;
      let FORWARD[x, x] point to the root of DESC(x) ;
      let BACKWARD[x, x] point to the root of ANC(x) ;
   end ;
end ;
```

A $length(x,y)$ operation can be performed in $O(1)$ time by simply returning $D[x,y]$.

```
function length (vertices : x, y) : integer;
begin
   return(D[x, y])
end;
```

A minimal path from u to v can be returned by first examining the entry $FORWARD[u,v]$. If it is *null*, then there is no path from u to v. Otherwise $FORWARD[u,v]$ allows us to locate v in $DESC(u)$, the tree of forward minimal paths rooted at u. If for each vertex in such trees reverse pointers to the parent are maintained, a bottom-up traversal from v to the root u in $DESC(u)$ takes at most $O(k)$ time to return a minimal length path from u to v in G, where $k \leq n$ is the length of the achieved path. A path $e_1, e_2, \ldots, e_k$ is returned as the sorted list of vertices $t(e_1), t(e_2), \ldots, t(e_k)$; if there is no path from x to y then the empty list is returned.

Figure 2.1: Trees of forward and backward paths of minimal length

Given a digraph $G = (V, E)$ and a vertex $x \in V$, a *tree of forward paths rooted at* x [30] is a tree $T_f(x) = (X, S)$ rooted at x such that

(i) X is the set of descendants of x in G; and

(ii) $S \subseteq E$.

Similarly, a *tree of backward paths* rooted at x is a tree $T_b(x) = (X, S)$ rooted at x such that

(i) X is the set of ancestors of x in G; and

(ii) $S \subseteq E^R$, where $E^R = \{(y, x) \mid (x, y) \in E\}$ is called the *set of reversed edges.*

A tree of forward (backward) paths $T = (X, S)$ rooted at x is said to be of *minimal length* (or equivalently it is said to be a *tree of forward (backward) minimal paths*) if for each vertex $v \in X$, the depth of v in T coincides with the length of the minimal path from x to v. In a similar vein, given a dag G, a tree of forward (backward) paths $T = (X, S)$ rooted at x is said to be of *maximal length* (or equivalently it is said to be a *tree of forward (backward) maximal paths*) if for each vertex $v \in X$, the depth of v in T coincides with the length of the maximal path from x to v. Figure 2.1 shows a digraph G with the trees of forward and backward paths of minimal length rooted at one of its vertices.

The above terminology can be extended to a graph G with edge costs. Denote by $w(i, j)$ the cost of edge (i, j). The cost of a path p can be defined as $w(p) = \sum_{(i,j)\in p} w(i, j)$. A path p from x to y is said to be the *shortest path* from x to y if there is no path from x to y whose cost is less than $w(p)$. A *tree of forward (backward) shortest paths* T is a tree of forward (backward) paths such that each path in T is a shortest path in G.

2.3 The Data Structure

We now present a data structure that maintains on-line a solution to the All Pairs Minimal Length Paths Problem. Namely, we assume that all the edges have unit costs. For sake of brevity, we denote by $add(i, j)$ the operation $add(i, j, 1)$. Our data structure supports each *length* operation in $O(1)$ time, and each *minpath* operation in $O(k)$ time, where k is the length of the minimal length path traced. The overall time required to maintain the data structure during a sequence of (at most $O(n^2)$) *add* operations is $O(n^3 \log n)$. The space complexity is $O(n^2)$.

In addition to maintaining and updating the $n \times n$ distance matrix D, we extend some ideas introduced in [30] by associating with each vertex $v \in V$ the two sets of its descendants and ancestors, called *DESC(v)* and *ANC(v)*. In order to retrieve information about minimal length paths, $DESC(v)$ is organized as a tree of forward minimal paths

As an implication of our techniques, we show also how to achieve the same bounds while maintaining longest paths in directed acyclic graphs, when *add* operations do not introduce cycles. This is not a restriction, since the longest path problem in cyclic graphs is known to be NP-complete [22].

The remainder of the chapter consists of five sections. In section 2 we give preliminary definitions. A data structure for maintaining incrementally shortest paths with unit edge costs is introduced in section 3, and its time complexity is analyzed in section 4. Section 5 discusses the case of integer edge costs, while section 6 is devoted to longest paths in directed acyclic graphs.

2.2 Preliminaries

We assume that the reader is familiar with the standard graph theoretical terminology for computer science as contained for example in [2, 52]. In particular, a *directed graph* $G = (V, E)$ (also referred to as a *digraph*) consists of a finite set $V = \{1, 2, \ldots, n\}$ of *vertices* and a finite set E of (directed) edges such that each edge e has a *head* $h(e) \in V$ and a *tail* $t(e) \in V$. We consider the edge $e = (t(e), h(e))$ as directed from $t(e)$ to $h(e)$ and we say that e leaves $t(e)$ and enters $h(e)$. A (directed) *path* $p = e_1, e_2, \ldots e_k$ is a sequence of edges such that $h(e_i) = t(e_{i+1})$ for $1 \leq i \leq k-1$. The path p is from vertex $t(p) = t(e_1)$ to vertex $h(p) = h(e_k)$ and contains the edges $e_1, e_2, \ldots, e_k$ and the vertices $t(e_1), t(e_2), \ldots t(e_k), h(e_k)$. The path is *simple* if all its vertices are distinct. The *length* of a path is the number of edges it contains. A path p from x to y is said to be of *minimal length* (or in short *minimal*) if there is no other path from x to y that contains a smaller number of edges. A (directed) *cycle* is a nonempty (directed) path from a vertex to itself. A vertex v is said to be *reachable* from a vertex u if there is a path p from u to v. If v is reachable from u, then u is said to be an *ancestor* of v and v a *descendant* of u. If $u \neq v$, u is a *proper ancestor* of v and v is a *proper descendant* of u. If there is an edge from u to v, then u and v are said to be *adjacent*. If $G = (V, E)$ is a digraph, the digraph which has the same vertex set as G and has an edge from u to v if and only if v is reachable from u in G, is called the *transitive closure* of G. A digraph with no cycles is called a *directed acyclic graph* (in short *dag*). A *rooted tree* T is a dag satisfying the following three properties.

(i) There is only one vertex, referred to as the *root*, with no edges entering it.

(ii) Every vertex but the root has exactly one entering edge.

(iii) There is a (unique) path from the root to each vertex.

A *leaf* of the tree is a vertex with no outgoing edges. If there is an edge (u, v) in T, u is said to be a *parent* of v and v a child of u. The *depth* of a vertex v in T is the length of the path from the root to v. Given a digraph $G = (V, E)$, a *spanning tree* is a rooted tree $T = (V, S)$ such that $S \subseteq E$.

decrease(x, y, Δ) : decrease by $\Delta > 0$ the cost of edge (x, y);

length(x, y) : return the length of the shortest path from x to y, if one exists; return $+\infty$ otherwise;

minpath(x, y) : return the shortest path from x to y, if one exists.

The problem must be solved in an on-line fashion, i.e., each operation must be performed before the next one is known. Another constraint of the problem is that we would like to answer *length* queries in $O(1)$ time and perform *minpath* operations in time linear in the length of the traced path. We assume that *add* operations insert edges that are not already in the graph, since repeated insertions of the same edge can be easily taken care of.

This problem has been previously studied [16, 41, 46, 60] but the known solutions seem far from optimal. Even and Gazit [16] showed how to maintain a solution to the All Pairs Shortest-Paths during edge insertions, edge deletions and edge-cost updates. Their data structure requires $O(n^2)$ time to be updated after either an edge insertion or an edge-cost decrease and $O(mn + n^2 \log n)$ time after either an edge deletion or an edge-cost increase, where m is the current number of edges in the graph. Notice that the time required by an edge deletion or an edge-cost increase is the same as the time required to recompute all pairs shortest paths from scratch [19]. Rohnert [41] and recently Yellin [60] gave similar bounds. We remark that the bounds achieved by all these data structures cannot be amortized against sequences of operations; in particular, the total time they require to support a sequence of σ both edge insertions and edge-cost decreases is $O(\sigma n^2)$.

In this chapter, we consider the problem of maintaining on-line a solution to the All Pairs Shortest-Path Problem on a directed graph under edge insertions and edge-cost decreases, with integer edge costs in a fixed range $[1 \ldots C]$. The maximum number of edge updates for this problem is $O(Cn^2)$, where n is the number of vertices in the graph, since each edge can be inserted at most once and its cost can be decreased at most C times. As a consequence, the algorithms in [16, 41, 60] solve this problem in a total of $O(Cn^4)$ worst-case time. In this chapter, we present algorithms and data structures to solve this problem in a total of $O(Cn^3 \log(nC))$ worst-case time and $O(n^2)$ space. Since we show that there are sequences of edge insertions and edge-cost decreases that force as many as $\Omega(Cn^3)$ changes in the distance matrix D, our algorithm is only a factor of $\log(nC)$ away from the best possible bound. Furthermore, our algorithm is simple and therefore likely to be used in practice.

For the case of unit edge costs, previous algorithms [16, 41, 60] require $O(n^4)$ worst-case time over a sequence of at most $O(n^2)$ edge insertions. Using our algorithm, the overall time required to solve this problem becomes $O(n^3 \log n)$ in the worst case. Since there are sequences of edge insertions that require $\Omega(n^3)$ changes in the distance matrix D, our algorithm is again only a logarithmic factor away from the best possible bound. In the remainder of this chapter we refer to the All Pairs Shortest-Paths Problem with unit edge costs as the All Pairs Minimal Length Paths Problem.

Chapter 2

Dynamic Shortest-Path Algorithms

In this chapter we study the problem of maintaining on-line a solution to the All Pairs Shortest-Paths Problem in a directed graph $G = (V, E)$ where edges may be dynamically inserted or have their cost decreased. For the case of integer edge costs in a given range $[1 \ldots C]$, we introduce a new data structure which is able to answer queries concerning the length of the shortest path between any two vertices in constant time, and to trace out the shortest path between any two vertices in time linear in the number of edges reported. The total time required to maintain the data structure under a sequence of at most $O(n^2)$ edge insertions and at most $O(Cn^2)$ edge-cost decreases is $O(Cn^3 \log(nC))$ in the worst case, where n is the total number of vertices in G. For the case of unit edge costs, the total time required to maintain the data structure under a sequence of at most $O(n^2)$ insertions of edges becomes $O(n^3 \log n)$ in the worst case. The same bounds can be achieved for the problem of maintaining on-line longest paths in directed acyclic graphs. All our algorithms improve previously known algorithms and are only a logarithmic factor away from the best possible bounds.

2.1 Introduction

We define the On-Line All Pairs Shortest-Paths Problem as follows. We wish to maintain, during both edge insertions and edge-cost decreases, a distance matrix D such that entry $D[i, j]$ contains the length of the shortest path from i to j, for any pair of vertices (i, j). We assume to start from an empty graph, i.e., a graph with n vertices and no edges. Another way to look at this problem is to maintain an underlying directed graph under an arbitrary sequence of operations of the following four kinds:

$add(x, y, w)$: insert an edge of cost w from vertex x to vertex y;

Sometimes it is useful to say that one function has strictly smaller asymptotic growth than another. The "little oh" notation is defined as follows:

$$g(n) \in o(f(n)) \text{ if and only if } \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0.$$

That is, $o(f(n))$ is the set of functions with order strictly smaller than $f(n)$.

We consider the analysis of algorithms from three different points of view. The first is the *worst case* of the algorithm, that is we consider those input instances of that size on which the algorithm takes the most time. The second is the *average case* of the algorithm, that is we average the execution times on instances of a certain size. The last is the *amortized case* [53], which means averaging the running time per operation of an algorithm over a worst-case sequence of operations.

1.3 Models of Computation

We consider two different models of computation for analyzing our algorithms. The first is the *pointer machine* [43, 51], and the second is the *random access machine*. They can be defined as follows.

The storage of a pointer machine consists of an unbounded collection of records connected by pointers. Each record can contain an arbitrary amount of additional information, but no arithmetic is allowed to compute the address of a record.

Another model of computation considered in the literature is the *random access machine*, whose memory consists of an unbounded sequence of registers, each of which is capable of holding an arbitrary integer. The main difference with pointer machines is that in random access machines the use of address arithmetic techniques is permitted and there is a fixed instruction set. In order to make the achieved lower and upper bounds meaningful, usually it is assumed that the size of a register is bounded by $O(\log n)$ bits. A formal definition of random access machines can be found in [2, pages 5–14]. Another model of computation, known as the *cell-probe model of computation*, was introduced by Andrew Yao [59]. In this model, the cost of computation is measured by the total number of memory accesses to a random access memory with $\log n$ bits cell size. All other computations are considered to be free.

operation is supported in $O(n^{2/3})$ time in the worst case, where n is the total number of vertices in the graph. We use again a partition of the graph into vertex clusters, but this time we combine it with a different representation of the interior of each cluster.

In Chapter 5 we present efficient algorithms for the partially dynamic maintenance of the 3-edge-connected of a graph during edge and vertex insertions. Our algorithms support each operation in $O(\alpha(q,n))$ time, where α is the inverse Ackermann's function, q is the total number of operations, and n is the final number of vertices. We exploit graph properties to represent 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 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.

Finally, Chapter 6 lists some open problems and concluding remarks.

Chapter 2 is joint work with Giorgio Ausiello, Alberto Marchetti-Spaccamela, and Umberto Nanni. Chapters 3, 4, and 5 are joint work with Zvi Galil.

1.2 Methods of Analysis

Although the problems we take into account have practical applications, we consider the analysis of the algorithms presented in this paper from the perspective of theoretical computer science. We concentrate on the ways the resource usage (computing time, memory space) grows with increasing problem size. Therefore we compare the merit of different algorithms based on formulae for their resource usage, rather than on experimental information. Further, we ignore for the most part the constant factors in these formulae, giving them only up to orders of magnitude. That is, we concentrate on asymptotic growth rates of these formulae rather than in their exact mathematical expressions.

This is described by the following terminology. We use the notation $O(f(n))$ ("big Oh" notation) to describe the set of functions that grow asymptotically *no faster* than $f(n)$. The notation $\Omega(f(n))$ describes the set of functions that grow asymptotically *at least* as fast as $f(n)$. Functions that grow asymptotically *at the same rate* as $f(n)$ are expressed by $\Theta(f(n))$. Formally the definitions for O, Ω, and Θ are:

$g(n) \in O(f(n))$ if there exists constants $n_0 \geq 0$ and $c > 0$ such that $g(n) \leq cf(n)$, for all $n \geq n_0$

$g(n) \in \Omega(f(n)$ if there exists constants $n_0 \geq 0$ and $c > 0$ such that $g(n) \geq cf(n)$, for all $n \geq n_0$

$g(n) \in \Theta(f(n)$ if there exists constants $n_0 \geq 0$ and $c > 0$ such that $g(n) = cf(n)$, for all $n \geq n_0$

dynamic maintenance of a minimum spanning tree requires $O(\log m)$ [1] time per insertion [18], where m is the current number of edges in the graph, by using the dynamic trees of Sleator and Tarjan [44, 45]. However, in both cases no better bound than $O(\sqrt{m})$ is known for the corresponding fully dynamic problems [18]. Moreover, despite intensive research on dynamic graphs, there are very few graph-theoretic problems for which a fully dynamic non-trivial algorithm is known. These are mostly problems on very restricted classes of graphs such as trees [44, 45], series parallel graphs [13, 32], st-planar graphs and spherical st-planar graphs with fixed embedding [47, 48, 49], and planar graphs with fixed embedding [15]. The only fully dynamic technique known to date for general graphs is the algorithm by Fredrickson for maintaining minimum spanning trees and connected components of undirected graphs [18]. Although considerable effort has been spent on many other fundamental graph problems, a fully dynamic solution of these problems has remained elusive.

1.1 Overview of the Dissertation

In this dissertation we consider fully and partially dynamic algorithms and data structures for graph problems, such as edge connectivity, vertex connectivity and shortest paths. The solutions of all these dynamic graph problems share many common features and combine a variety of graph properties, novel algorithmic techniques and new data structures. Motivations for studying these problems arise in different areas, including the design and management of reliable communication networks, and network analysis in electrical engineering. They also appear as subproblems in many other graph problems.

Chapter 2 presents efficient solutions to the dynamic shortest-path problem on a directed graph with integer edge costs in $[1 \ldots C]$. The total time required to perform a sequence of at most $O(Cn^2)$ edge insertions and edge-cost decreases is $O(Cn^3 \log(nC))$, where n is the total number of vertices in the graph. This is obtained by maintaining the trees of shortest paths rooted at each vertex of the graph, and by developing sophisticated algorithms for updating such trees efficiently. Our time bound is only a logarithmic factor away from the best possible bound.

In Chapter 3 we describe efficient algorithms for maintaining the 2-edge-connected components of an undirected graph under insertions and deletions of edges. We show how to support each operation in $O(m^{2/3})$ for general graphs, and in $O(\sqrt{n \log\log n})$ for planar graphs, where m and n are respectively the current numbers of edges and vertices in the graph. Our algorithms for this problem partition the graph into vertex clusters, and find a way to maintain a succinct encoding of each cluster which satisfies nice properties with respect to 2-edge-connected components in the graph. This enables one to answer queries about edge connectivity properties without having to look at the entire graph each time.

In Chapter 4 we introduce a new data structure that is able to maintain the biconnected components of a planar graph under insertions and deletions of edges. Each

[1] All the logarithms are assumed to be to the base 2 unless explicitly specified otherwise.

Chapter 1

Introduction

In the last decade there has been a growing interest in dynamic problems on graphs. 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 often 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 (either 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 often more difficult to design than their static counterparts.

In the realm of graph problems, several dynamic data structures have been proposed to support insertions and deletions of edges and/or vertices in a graph, in addition to certain types of queries. In particular, much attention has been devoted to (among others) the dynamic maintenance of connected components [17, 18, 24, 40, 44, 56, 57], 2- and 3-connectivity [8, 56, 57], transitive closure [5, 29, 30, 31, 39, 48, 49, 60], planarity [7, 8, 47], shortest paths [13, 16, 41, 46, 60], and minimum spanning trees [12, 15, 18, 46]. Dynamic algorithms for graph problems are both of theoretical and of practical interest in several application areas, including communication networks, computer aided design [3], distributed computation, database systems [1, 14, 33, 58], logic programming [4, 9], incremental data flow analysis [10, 11], high level languages for incremental computations [61], incremental generation of parsers [28] and programming environments [3, 26, 35].

Fully dynamic graph problems are usually much more difficult to solve than the corresponding partially dynamic problems. For example, the partially dynamic maintenance of the connected components of an undirected graph can be done in $O(\alpha(q,n))$ amortized time per insertion using set-union data structures [54], where α is a very slowly growing function (a functional inverse of Ackermann's function), q is the total number of operations, and n is the total number of vertices in the graph. Similarly, the partially

Acknowledgments

I am deeply indebted to my advisor, Zvi Galil, with whom I did most of the work in this dissertation. Being one of his students has been a wonderful human and professional experience: I could not have wished myself a better advisor.

Many other people have helped me in many different ways during my years at Columbia. Talking to Alberto Apostolico has given me new perspectives on several problems. Giorgio Ausiello introduced me to dynamic graph problems: he has been much like a long-distance advisor for me. I am indebted to Dany Breslauer for his generosity, and for his insightful comments on some of my work. I am grateful to Raffaele Giancarlo for many technical as well as non-technical discussions. It has been a rare pleasure working with Alberto Marchetti-Spaccamela and Umberto Nanni. My interests on graph algorithms and data structures have been greatly enhanced by contacts with Kurt Mehlhorn. The time spent working with Moti Yung in front of a board has been memorable. I benefitted greatly from having him (officially or unofficially) in my office during my stay at Columbia.

I would like also to express my gratitude to the other members of my thesis defense committee for their suggestions and useful comments: Dan Bienstock, Terry Boult, Jonathan Gross and Neil Sarnak. I have been very fortunate to have Neil as a technical liaison during my IBM fellowship years: I have benefitted a lot from his insight, enthusiasm, and encouragement. I wish to thank also Thanasis Tsantilas for serving on my thesis proposal committee, and Lane Hemachandra and Henryk Wozniakowski for serving on my area qualifying committee.

I am grateful to many other friends for what they taught me. Bill Schilit taught me all the software I could possible learn and American life issues in exchange of just few Italian pasta recipes. Matt Franklin taught me some good jokes in exchange of bad ones. Kunsoo Park taught me that a silent person is often a great person. Jong Gyun Lim taught me few Korean words at no exchange. Bülent Yener taught me that it is possible to survive my terrible puns for almost four years (but not more). At various stages of this research, I have had profitable interactions with Claudio Arbib, Marek Chrobak, Fabrizio d'Amore, David Eppstein, Joan Feigenbaum, Paolo Franciosa, Giorgio Gambosi, Stuart Haber, Torben Hagerup, Shay Kutten, Stefan Näher, Alessandro Panconesi, Baruch Schieber, Maurizio Talamo, Jeff Westbrook, Dan Yellin, and XiangDong Yu. A number of friends, such as Enrico Cadorin, Oona Gargan, Giuseppe Scelsi, and Alison Ray, have made my life in New York City memorable. On the other hand, Rosemary Addarich, Germaine L'Eveque, Kerny McLaughlin, and Renate Valencia have been making my life at Columbia much easier with their assistance.

My family had a crucial influence in making me who I am today. My parents, my sister, her husband, and my little nephew Giorgio have shown exceptional support and care for me during my years away from Italy. All their love helped me through the difficult times and made this dissertation much easier to write.

This research has been supported in part by an IBM Graduate Fellowship, and by NSF Grants CCR-8814977 and CCR-9014605.

Ai miei genitori
To my parents

List of Figures

2.1 9

2.2 17

2.3 21

3.1 28

3.2 31

3.3 32

3.4 34

3.5 39

4.1 55

4.2 58

4.3 60

4.4 62

4.5 63

5.1 79

5.2 83

5.3 84

5.4 86

5.5 91

4 Maintaining Biconnected Components of Dynamic Planar Graphs 53

4.1 Introduction 53

4.2 A Data Structure for Connected Planar Graphs 56

4.3 Implementing the Operations 67

4.4 Dealing with Unconnected Planar Graphs 69

4.5 Concluding Remarks 70

5 Partially Dynamic Algorithms for 3-Edge Connectivity 71

5.1 Introduction 71

5.2 Some Properties of k-Edge Connectivity 73

5.3 A Linear-time Algorithm for 3-Edge Connectivity 74

5.4 Maintaining 3-Edge-Connected Components of 2-Edge-Connected Graphs . 77

5.5 Maintaining 3-Edge-Connected Components of Connected Graphs 89

6 Conclusions and Further Research 94

Contents

1 Introduction **1**

1.1 Overview of the Dissertation 2

1.2 Methods of Analysis 3

1.3 Models of Computation 4

2 Dynamic Shortest-Path Algorithms **5**

2.1 Introduction 5

2.2 Preliminaries 7

2.3 The Data Structure 8

2.4 Correctness and Time Complexity 13

2.5 Integer Edge Costs 20

2.6 On-Line Maintenance of Longest Paths in Directed Acyclic Graphs 24

3 Fully Dynamic Algorithms for 2-Edge Connectivity **26**

3.1 Introduction 26

3.2 The Data Structure for Connected Graphs 29

3.3 Testing for Bridges Inside Clusters 36

3.4 Implementing the Operations 45

3.5 Time Complexity 50

3.6 Dealing with Unconnected Graphs 52