Figure 5

Figure 4

Figure 3

Figure 2

Figure 1

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

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

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

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

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

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

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

[55] M. Yannakakis, "Graph theoretic methods in database theory", *Proc. ACM Conf. on Principles of Database Systems*, 1990, 230-242.

[56] A. C. Yao, "Should tables be sorted?", *J. Assoc. Comput. Mach.* 31 (1984), 245–281.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[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", *Proc. First Annual ACM-SIAM Symp. on Discrete Algorithms*, 1990, 1–11.

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

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

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

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

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

[21] Z. Galil, and G. F. Italiano, "Maintaining the 3-edge-connected components of a graph on-line", manuscript, 1990.

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

[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 (1984), 338–355.

[26] R. Hoover, "Incremental Graph Evaluation", Ph. D. Thesis, Technical Report 87-836, Department of Computer Science, Cornell University, 1987.

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

[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 (1983), 95–97.

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

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

References

[1] R. Agrawal, A. Borgida, and H. V. Jagadish, "Efficient management of transitive relationships in large data and knowledge bases", *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", *Proc. 1st Annual ACM-SIAM Symp. on Discrete Algorithms*, 1990, 32–42.

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

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

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

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

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

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

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

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

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

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

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

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 6.1 *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 $O(\sqrt{n \log\log n})$.*

7 Concluding Remarks

In this paper we have described efficient algorithms to maintain 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.

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.

We can achieve the same $O(m^{2/3})$ bound per operation also for the fully dynamic maintenance of the 3-edge-connected components of a graph [22].

The problem of maintaining the biconnected components of a graph during edge insertions and deletions deserves further study. We have been able to prove that if the graph is planar, this problem can be solved in $O(n^{2/3})$ worst-case time per operation [20], and we are currently exploring the case of general graphs.

Acknowledgments

We are grateful to Greg Frederickson for his valuable comments. We would like to thank also Dany Breslauer, Kurt Mehlhorn, Neil Sarnak and Moti Yung for useful discussions.

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 5.1 and 5.2 can be made worst-case as the following theorem shows.

Theorem 5.3 *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

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

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. □

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

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 4.1. 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 2.4 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 5.2 *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 3.8). 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 $\mathcal{T}(\mathcal{C}_j)$, 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 [38], which implies that the time required to initialize the data structures pertinent to a cluster (as analyzed in Lemma 2.4) 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 5.1) 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

$E(\mathcal{C},\mathcal{C}_i)$ 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 4.2 and 4.4, 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 2.4. 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 4.2. 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 4.2 and 4.4, 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 2.4, 4.2 and 4.4 all this can be done in $O(k)$ time. Therefore we have the following lemma.

Lemma 4.5 *The total time required to update our data structure because of a* DeleteEdge *operation is* $O(k)$.

5 Time Complexity

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

Theorem 5.1 *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 4.1, 4.3 and 4.5, 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

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 2.4, 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 4.2, each cluster can be split in $O(k)$ time, and by Lemma 2.4 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 4.3 *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 4.2. 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 4.4 *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 $\widetilde{G}$.

All the data structures pertinent to the new cluster $\mathcal{C}$ can be recomputed from scratch in $O(k)$ by Lemma 2.4. 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

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

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

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 4.2 *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 2.4. 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 the clusters containing

Since case (2) is the most expensive part of Lemma 3.4, 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. □

4 Implementing the Operations

We perform a *Same2EdgeBlock*(x, y) operation as follows. We first compute $\mathcal{G}_{x,y}(G)$, the supergraph induced by vertices x and y. By Lemma 2.2, 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 [49]. Because of Lemma 2.2 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 correct because of Lemma 2.3. 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 2.3, 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 3.1, all the clusters in $\mathcal{S}$ as well as their $\mathcal{X}^i_{\mathcal{C}}$, $\mathcal{Y}^j_{\mathcal{C}}$ and $\mathcal{Z}^{h,t}_{\mathcal{C}}$ neighbors can be found in $O(\frac{m^2}{k^2})$ time. We run the algorithm of Lemma 3.8 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 4.1 *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 2.3 and requires $O(k + \frac{m^2}{k^2})$ time because of Lemma 2.2. Otherwise we compute $\mathcal{S}$ in $O(\frac{m^2}{k^2})$ time as described in Lemma 3.1 and run the algorithm given in Lemma 3.8 on all the clusters in $\mathcal{S}$. This gives a correct answer because of Lemmas 2.3 and 3.8. 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 3.8. To bound this time, we notice that by Lemma 2.3 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 $\{\mathcal{X}^i_{\mathcal{C}}\}_{1 \le i \le p(\mathcal{C})}$, $\{\mathcal{Y}^j_{\mathcal{C}}\}_{1 \le j \le q(\mathcal{C})}$, and $\{\mathcal{Z}^{h,t}_{\mathcal{C}}\}_{3 \le h \le s(\mathcal{C}), 1 \le t \le r_h(\mathcal{C})}$ 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.

Notice that if γ' is the root of $\mathcal{T}(\mathcal{C})$, then e' is not defined; but in this case by Lemma 3.7 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 3.7, no edge in $\mathcal{T}(\mathcal{C})$ satisfy both conditions (i) and (ii), and by Lemma 3.2 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 3.5, 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 3.2 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 3.7, no edge in $\mathcal{T}(\mathcal{C})$ satisfy both conditions (i) and (ii), and by Lemma 3.2 there is no bridge in $\mathcal{C}$ separating x and y in G.

This shows how to deal with case (3) of Lemma 3.4. 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 3.2.

Lemma 3.8 *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 3.2 can be checked in $O((p(\mathcal{C}) + q(\mathcal{C})) \log k)$ by Lemma 3.6.

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, $1 \leq i \leq r$, is the time spent in the range queries in the balanced search trees, which is $O(r_{i'}(\mathcal{C}) \log k)$ by Lemma 3.5. Consequently, the total time required to examine the $r \leq s(C)$ 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 3.5, this can be implemented in $O((p(C) + q(C)) \log k) = O(d(\mathcal{C}) \log k)$ time.

$r \leq k$. 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 3.2, without any loss of generality we can restrict our attention to the edges $e'_1, e'_2, \ldots, e'_r$.

Lemma 3.7 *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 3.2 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 3.2. By Lemma 3.4, 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:

(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 3.5, 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 3.2.

Lemma 3.6 *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 3.1. By Lemma 3.4, 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 3.5, 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 3.4 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 3.6, 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 3.2. If red and black vertices cannot be separated, then by Lemma 3.2 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 3.4. As shown in Lemma 3.4, $\pi_{\rho,\beta}$, 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 3.2 has to be checked.

We first consider case (3) of Lemma 3.4. 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 it γ_h. By Lemma 3.3, 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 3.2. 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 $\mathcal{C}$:

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 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 3.4 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 3.5 *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 $E(\mathcal{C}',\mathcal{C})$ is in the subtree of $\mathcal{T}(\mathcal{C})$ rooted at v if and only if the preorder number of u satisfies $\mathit{preorder}(v) \leq \mathit{preorder}(u) \leq \mathit{preorder}(v) + \mathit{size}(v) - 1$. Consequently, we have to check whether the balanced search tree $E(\mathcal{C}',\mathcal{C})$ contains at least one item in the range $[\mathit{preorder}(v), \mathit{preorder}(v) + \mathit{size}(v) - 1]$. 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 $\mathit{preorder}(v)$, 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 $[\mathit{preorder}(v), \mathit{preorder}(v) + \mathit{size}(v) - 1]$ if and only if the item found is less than or equal to $\mathit{preorder}(v) + \mathit{size}(v) - 1$. □

We remark that, in order to check conditions (i) and (ii) of Lemma 3.2, 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 3.2 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 3.3 *The vertex $\lambda(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_\ell)$ can be found in $O(\ell)$ time.*

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 3.3.

Corollary 3.1 *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 3.4 *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.

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 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 3.2 *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 5 shows an example of an edge that satisfies both conditions of Lemma 3.2.

[Figure 5]

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 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, 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 3.1 *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 [49], 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,

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 $\mathcal{T}(\mathcal{C})$; 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 2.4 *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 [49]. 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].

[Figure 4]

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

Lemma 2.2 *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 2.3 *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 $\mathcal{G}_{x,y}(G)$. □

By Lemma 2.3, 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 [49] on $\mathcal{G}_{x,y}(G)$, which requires $O(k+\frac{m^2}{k^2})$ time because of Lemma 2.2. 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

Lemma 2.1 *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 2.1 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 2 shows a graph G together with its super-graph $\widetilde{G}$.

[Figure 2]

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 shows the full representation of a cluster.

[Figure 3]

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 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 4 shows a graph $G=(V,E)$ and the super-graph induced by two vertices.

response to dynamic changes than simply running a linear-time off-line algorithm. Westbrook and Tarjan [54] 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 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 paper consists of six 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. Finally, in Section 7 we list some concluding remarks.

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.

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

closure [6, 29, 30, 31, 36, 47, 48, 57], planarity [13, 14, 46], shortest paths [5, 12, 16, 37, 41, 45, 57], and minimum spanning trees [11, 15, 18, 45]. 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, 33, 55], logic programming [4, 8], incremental data flow analysis [9, 10], high level languages for incremental computations [58], 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 [51], 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 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 [43, 44]. 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 [43, 44], series parallel graphs [12, 32], st-planar graphs and spherical st-planar graphs with fixed embedding [46, 47, 48], 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.

In this paper 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 [54], 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

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

1 Introduction

An undirected graph $G = (V, E)$ is said to be *biconnected* if there are at least two vertex disjoint paths from each vertex to every other vertex. It is well known [2] that a graph G is not biconnected if and only if there is a vertex whose removal disconnects G: such a vertex is called an *articulation point*. 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-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 1 shows a graph together with its biconnected and bridge-connected components.

[Figure 1]

The problem of computing the bridge-connected components of undirected graphs arise naturally in many applications and have been extensively studied. Tarjan [49] gives optimal linear-time sequential algorithms. Awerbuch and Shiloach [7] and Tarjan and Vishkin [52] give logarithmic-time parallel algorithms. Recently, Westbrook and Tarjan [54] 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 [39, 50] as well as in the cell-probe model of computation [56].

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, 43, 53, 54], 2- and 3-connectivity [14, 53, 54], transitive

Fully Dynamic Algorithms for 2-Edge Connectivity*

Zvi Galil
Columbia University, and
Tel-Aviv University

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

Abstract

We study the problem of maintaining the 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 [54]. 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.

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

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

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