Data Structures for Maintaining Biconnectivity and 2-edge Connectivity — A Bibliographic Survey

Manor Mendel

September 1994

1 Introduction

For an undirected graph $G = (V, E)$ the following terms are defined:

Biconnected component (also called *2-vertex connected component*) is a subset of V such that each two vertices in the biconnected component are connected and would remain connected even if one vertex is removed. Biconnected components are abbreviated *blocks*. A vertex that appears in more than one block is called articulation point. A removal of an articulation point would disconnect the blocks.

Bridge-connected component (also called *2-edge connected component*) is s subset of V such that each two vertices in the bridge-connected component are connected and would remain connected even if one edge is removed. Bridge-connected components are abbreviated *bridge-blocks*. Bridge-blocks partition the vertices of G into equivalence classes. An edge that connects vertices from different bridge-blocks is called a *bridge*. A removal of bridge would disconnect the bridge-blocks.

The problem of statically finding blocks and bridge-blocks is well understood. Hopcroft and Tarjan [9] and Tarjan [18] show sequential algorithms that run in time $O(|V| + |E|)$. In this article we review articles that attempt to solve the following problem: "Given a dynamic changing undirected graph and two vertices in the graph, are the two vertices biconnected? 2-edge connected?".

We distinct between two types of dynamic graphs:

Partially-dynamic that allows only insertions of vertices and edges.

Fully-dynamic that allows insertions and deletions of vertices and edges.

In general, it seems that biconnectivity is harder than 2-edge connectivity since bridge-blocks are equivalence classes of vertices, while blocks are not. Galil and Italiano [8] support this opinion by presenting a constant time per operation reduction from k-edge connectivity to k-vertex connectivity. Fig. 1 illustrates a graph transformation that reduce 2-edge connectivity to biconnectivity (but do not preserve planeness). No reverse reduction is known.

Figure 1: An example of transformation for vertex v.

2 Partially-Dynamic Graphs

Partially-dynamic data structures support operations of biconnectivity/2-edge connectivity queries, edges insertions and vertices insertions, intermixed.

Tamassia [16, 17] shows a data structure for biconnectivity queries about plane embedded graph that achieves $O(\log n)$ amortized time per operation.

Westbrook and Tarjan [19] presents algorithms that support biconnectivity or 2-edge connectivity queries in $O(\alpha(m,n))$ amortized time per operation, where n is the number of vertices, m is the number of operations and α is a functional inverse of Ackerman's function. We only sketch their construction and only for 2-edge connectivity queries.

First, we describe a simple data structure called *Bridge-Block Forest* (BBF) that takes $O(\log n)$ time per operation. Given an undirected graph $G=(V,E)$, BBF(G) is a forest of rooted trees that contain two types of nodes: *square nodes* which represent the vertices of G; and *rounded nodes* which represent the bridge-blocks. The square node are the leaves of the BBF and their parents represent their bridge-blocks. Two rounded nodes are connected if and only if their respective bridge-blocks are connected (via bridge). See Fig. 2 for illustration of a BBF.

The following operation are allowed on the BBF:

evert(r)**.** Rerooting the tree containing the rounded node r such that r is its new root.

condense(r,s)**.** Condensing the simple path between the rounded nodes r and s (till their Lowest Common Ancestor) such that in the new tree, all the

Figure 2: (a) A graph example. (b) It's BBF

rounded nodes in the path, except the LCA, disappear and their square sons becomes the sons of the LCA.

link (r, s)**.** link r as direct son of s, where r and s are the roots of their respective trees.

The operations on G are translated to operations on BBF as follows:

insert vertex (v)**.** Make a tree composed of a square node v and a rounded father.

find block (v)**.** Return the label of the parent of v.

insert edge(u, v)**.** If u and v are in different connected components C_u and C_v, let r =*parent*(u), s =*parent*(v). If $|C_u| \leq |C_v|$ execute *evert*(r) and *link*(r, s); otherwise execute *evert*(s) and *link*(s, r). If u and v are in the same connected component, execute *condense path*(r, s).

Testing whether two vertices are in the same connected components can be done using auxiliary union/find data structure.

The evert operation takes $\min\{|C_u|, |C_v|\}$ steps and there are no more than n such steps. Therefore the total number of steps of insert/link, $T(n)$, is bounded by

$$\begin{aligned} T(1) &= 0 \\ T(n) &\leq \max_{1 \leq k \leq n/2} \{T(k) + T(n-k) + ck\} \end{aligned}$$

which has the solution $T(n) = O(n \log n)$.

The total number of *condense* steps is at most $O(n)$ since there are at most n rounded nodes, and at each step at least one of them is deleted. Query takes

$O(1)$ steps. Therefore the total number of steps is $O(n \log n + m)$. To implement condensing efficiently we use the union/find data structure, which means that the *parent* operation for square nodes takes $O(\alpha(m,n))$ amortized time. The total time is $O(n \log n + m\alpha(m,n)) = O(n \log n + m)$.

To reach the desired performance Westbrook and Tarjan replaced the BBF with a more sophisticated data structure called *link/condense trees*. *Link/condense trees* is derived from the *dynamic trees* data structure of Sleator and Tarjan [14, 15]. *Dynamic trees* is an application of *splay trees* that enable link and evert in $O(\log n)$ time. *Link/condense trees* adds the condensing operation, also in $O(\log n)$ time.

Using *link/condense trees*, a tree T of size n can be everted in $O(\log n)$ (instead of $\Theta(n)$) and a link of T under other tree T' takes also $O(\log n)$. In this case the recursive upper bound of $T(n)$, the total number of steps to implement insert/link becomes

$$T(n) \leq \max_{1 \leq k \leq n/2} \{T(k) + T(n-k) + c \log k\}$$

which yields $T(n) = O(n)$. Thus the total running time of the algorithm is $O(m\alpha(m,n))$.

La Poutré et al. [11] designed a different data structure for maintaining bridge-blocks and 3-edge connected components that runs in total time of $O(m\alpha(m,n))$. Their approach can be extended to maintain blocks.

Westbrook and Tarjan [19] also showed a lower bound in the cell probe model, of $\Omega(\alpha(m,n))$ time per operation. The lower bound is proved using a reduction from the problem of maintaining connected components in a graph (see section 3.2), which in turn can simulate union/find in constant time per operation.

Summarizing the results given above, maintaining bi/2-edge connectivity in partially-dynamic graphs takes $\Theta(\alpha(m,n))$ amortized time per operation.

3 Fully-Dynamic Graphs

Fully-dynamic graphs support update operations intermixed with bi/2-edge connectivity queries. Update operation is either insert or delete of edge or vertex.

As opposed to the partially-dynamic case, the problem is not yet "closed", meaning that the lower bounds do not match the upper bounds. Yet, for planar graphs the upper bounds are considerably better. By the term "planar graph" we mean that the graph remains planar *without* changing the embedding (in the plane) in the process of inserting new edges to the graph. The results are given in terms of time per operation, where n is the number of vertices and m is the number of edges.

The first to arise the problem of bi/2-edge connectivity in fully dynamic graphs were Westbrook and Tarjan [19].

3.1 Results for 2-edge connectivity

Galil and Italiano [6] were the first to present non-trivial results for 2-edge connectivity. Their algorithm achieves $O(m^{2/3})$ time per operation. For planar graphs, they devised an algorithm that works in $O(\sqrt{n}\log\log n)$ time per operation.

Frederickson [2] developed a data structure for maintaining a *Minimum Spanning Tree* (MST) dynamically. The idea in his data structure is to partition the vertices set (of size m) to $\Theta(m/z)$ clusters, each of size $\Theta(z)$. Clusters are connected parts of the MST. The clusters themselves form a graph that can be further partitioned. Applying this composition hierarchically, we have a tree, called *topological tree of* G, that each level is a decomposition of the level above it. Using this technique, Frederickson maintains a MST of general graph with $O(\sqrt{m})$ time per update and $O(\log n)$ time per query. The variant for planar graphs achieves $O(\log^2 n)$ time per query.

Frederickson [3, 4], using the technique for handling MST dynamically, showed an algorithm that achieve, for general graphs, $O(\sqrt{m})$ time per update and $O(\log n)$ time per query. For this, he developed the *ambivalent data structure. ambivalent data structure* maintains, for each tree edge e, some "candidate edges" for covering e (e is covered by f means that e is on the tree path between the end vertices of f, which makes e non-bridge edge) and testing whether e is covered, is actually done only on a relevant 2-edge connectivity query. With this technique, recomputing the "candidate edges" can be done on the clusters level, those saving time in the process of updating, by working on less detailed graph. The algorithm's variant for planar graphs takes $O(\log^3 n)$ time per update.

Eppstein et al. [1] developed a general technique called *sparsification.* Sparsification transforms algorithms for sparse graphs to general graphs. It do so by splitting a given graph to $\lceil m/n \rceil$ subgraphs, n edges in each. The relevant information for each subgraphs is summarized in a sparse subgraph (of size $O(n)$) which is called *sparse certificate.* Certificates are merged in pairs producing larger certificates, which are made sparser by applying the certificate reduction. The result is a balanced binary tree in which each node is represented by a sparse certificate. Each update involves $\log(m/n)$ nodes, with $O(n)$ edges each, instead one graph with m edges.

They applied sparsification "on top" Frederickson's algorithm, taking as certificates $T_1 \cup T_2$, where T_1 is a spanning forest of the given graph, G, and T_2 is a spanning forest of $G \setminus E(T_1)$. Thus, their algorithm achieves $O(\sqrt{n}\log(m/n))$ time per update of general graphs.

Hershberger et al. [10] improved Frederickson's algorithm for planar graphs down to $O(\log^2 n)$ time per update.

3.2 Results for biconnectivity

Galil and Italiano [7] showed an algorithm for planar graphs that takes $O(n^{2/3})$ per operation.

Rauch [12] presented an algorithm that achieves, for general graphs, $O(m^{2/3})$ time per update and constant time per biconnectivity query. She used Frederickson's partition of graphs [2] to devise algorithm that works in two-level approach: high-level graph (graph of the clusters) and internal and local graphs. Each update in the graph causes to: (a) A complete rebuild of constant number of internal graphs ($O(z)$ time). (b) Recomputing (in case of deletion of one of the spanning tree's edges) the high-level spanning tree ($O(m/z)^2$ time). Choosing $z = \lfloor m^{2/3} \rfloor$ results in $O(m^{2/3})$ amortized time per update. The algorithm's variant for planar graphs takes $O(\sqrt{n \log n})$ per update and $O(\log^2 n)$ time per query.

Eppstein et al. [1] used their sparsification technique on the static algorithm to achieve, for general graphs, $O(n \log(m/n))$ time per update and $O(1)$ time per query. As certificate they used $T_1 \cup T_2$, where T_1 is a breadth-first forest of the given graph, G, and T_2 is a breadth first forest of $G \setminus E(T_1)$.

Rauch [13] improved her results from [12] and achieved $O(\sqrt{m} \log n)$ amortized time per update and $O(1)$ time per query. It is done by using *ambivalent data structure* [3, 4] and sparsification to enable recomputing the high-level spanning tree in just $O((z + m/z) \log n)$ amortized time, which yields the desired update time. The algorithm's variant for planar graph works in $O(\log^2 n)$ time per operation.

Rauch [13] also showed a lower bound for k-edge and k-vertex connectivity testing: In fully-dynamic (planar) graphs the amortized time per operation has a lower bound of $\Omega(\log n \,/\, \log \log n)$ in the cell probe model.

Her technique for bi/2-edge connectivity can be described as two-steps reduction. First, we use a reduction to the connectivity problem from the following *Partial Prefix Sum* (PPS) problem : Given an array $A[1], \ldots, A[n]$ of bits, execute Add(l) and Sum(l) operations, where Add(l) increase the value of $A[l]$ by one modulo 2 and Sum(l) returns $S_l := (\sum_{1 \le i \le l} A[i]) \pmod 2$. PPS is known [5] to have a lower bound of $\Omega(\log n \,/\, (\log \log n + \log b))$ amortized time per operation in the cell probe model with wordsize b. The reduction is as follows: Given an instance of the PPS. Assume $S_0 := 1$. We construct a graph consisting of $n+1$ ascending nodes $v_0, \ldots, v_n$. v_i is called *odd* if $S_i = 1$ and *even* otherwise. The even and odd vertices are connected in two separate ascending chains. So v_i and v_0 are connected iff $S_i = 1$, therefore the Sum operation corresponds to a connectivity query. Add(i) corresponds to series of two delete operations and two insert operations of edges, that "swap" the chains's tails from v_i and upward. We use Van Emde-Boas priority queue to find the nearest vertices to v_i from the other chain, in $O(\log \log n)$ time per operation.

The second reduction is a constant time reduction from the connectivity problem to bi/2-edge connectivity problem. It is done by duplicating each vertex v to two connected vertices v' and v'', such that v adjacent to u in the original graph, corresponds to v' adjacent to u' and v'' adjacent to u''. Note that the two-step reduction produces a planar embedded graph.

Summarizing the results, we have

Upper and lower bounds for fully-dynamic graphs

The problem	Upper bound			Lower Bound
	Graph type	Operation update	query	
2-edge connectivity	general	$O(\sqrt{n}\log(m/n))$	$O(\log n)$	$\Omega\left(\frac{\log n}{\log\log n}\right)$
	planar	$O(\log^2 n)$	$O(\log n)$	
2-vertex connectivity	general	$O(\sqrt{m}\log n)$	$O(1)$	
	planar	$O(\log^2 n)$	$O(\log^2 n)$	

References

[1] D. Eppstein, Z. Galil, G.F. Italiano and A. Nissenzweig. *Sparsification — A Technique for Speeding up Dynamic Graph Algorithms.* FOCS 33:60–69, 1992.

[2] G.N. Frederickson. *Data Structure for Online Updating of Minimum Spanning Trees, with applications.* SIAM J. comput. 14:781–798, 1985.

[3] G.N. Frederickson. *Ambivalent Data Structure for Dynamic 2-edge Connectivity and k-Smallest Spanning Trees.* FOCS 32:632–641, 1991.

[4] G.N. Frederickson. *Ambivalent Data Structure for Dynamic 2-edge Connectivity and k-Smallest Spanning Trees.* Technical Report CSD-TR-91-048, Purdue University, 1991.

[5] M.L. Fredman and M.E. Saks. *The Cell Probe Complexity of Dynamic Data Structures.* STOC 21:345–354, 1989.

[6] Z. Galil and G.F Italiano. *Fully Dynamic Algorithms for 2-edge Connectivity Problem.* STOC 23:317–327, 1991.

[7] Z. Galil and G.F Italiano. *Maintaining Biconnected Components of Dynamic Planar Graphs.* ICALP, LNCS, S–V, 18:331–350, 1991.

[8] Z. Galil and G.F Italiano. *Reducing edge connectivity to vertex connectivity* SIGACT News 22(1):57-61, 1991.

[9] J. Hopcroft and R.E. Tarjan. *Algorithm 449: Efficient Algorithms for Graph Manipulation.* Comm. ACM 16:372-378, 1973.

[10] J. Hershberger, M. Rauch and S. Suri. *Fully Dynamic 2-edge Connectivity in Planar Graphs.* SWAT 3:235–244, S–V LNCS 621, 1992.

[11] J.A. La Poutré, J. van Leeuwen and M. H. Overmars. *Maintenance of 2- and 3-Connected Components of Graphs.* Technical report RUU-CS-90-26, Utrecht University, 1990.

[12] M. Rauch. *Fully Dynamic Biconnectivity in Graphs.* FOCS 33:50–59, 1992.

[13] M. Rauch. *Improved Data Structure for Fully Dynamic Biconnectivity.* STOC 26:686–695, 1994.

[14] D.D. Sleator and R.E. Tarjan. *A Data Structure for Dynamic Trees.* J. Comput. System Sci., 26:362–391, 1983.

[15] D.D. Sleator and R.E. Tarjan. *Self-adjusting Binary Search Trees.* J. ACM, 32:652–686, 1985.

[16] R. Tamassia. *A Dynamic Data Structure for Planar graph embedding.* ICALP 1988.

[17] R. Tamassia. *Dynamic Data Structure for two-dimensional searching.* Ph.D. thesis, University of Illinois, 1988. Technical report ACT–100.

[18] R.E. Tarjan. *Depth First Search and Linear Graph Algorithms.* SIAM J. Comput. 1:146–160, 1972.

[19] J. Westbrook and R.E. Tarjan. *Maintaining Bridge-connected and Biconnected components online.* Algorithmica 7:433–464, 1992.