Global Edge Connectivity in Dynamic Graphs

Kuat Yessenov

May 25, 2007

Abstract

We survey results related to the data structures for dynamic graphs which can maintain global edge connectivity. We explain the major techniques used in these data structures and explore in detail the algorithm by Thorup for polylogarithmic edge-connectivity.

1 Introduction

In this paper, we study data structures that maintain the *edge connectivity* λ of a graph, i.e. the minimal number of edges whose removal disconnects the graph. We use $G = (V, E)$, $n = \#V$, $m = \#E$ to denote a dynamic graph that supports edge deletions and insertions. A *min-cut* of a graph is a partition of the vertices into two sets with the minimal number of edges connecting them. The number of edges in any min-cut is the edge connectivity λ. The edge connectivity is the global measure of sensitivity of the graph to edge removals. This draws the distinction between edge connectivity data structures and $O(1)$-edge connectivity queries for pairs of vertices. These data structures find their application in the design of computer networks resilient to multiple link failures.

A $\sqrt{2}$-approximate data structure by Thorup and Karger [5] answers the edge connectivity query in polylogarithmic amortized time. Another data structure by Thorup [4] can give the exact value of λ at the cost of $\mathrm{O}(\sqrt{n})$ worst case time per graph update for polylogarithmic λ. In fact, no sublinear time algorithm is known for edge connectivity higher than 3, and the data structure by Thorup matches this bound. Both of these solutions share a number of effective techniques which we explain in this paper.

We introduce greedy tree packings in section 2. Section 3 explains the minimal spanning forests data structures used for tree packings. Sections 4 and 5 explain the data structure by Thorup [4] in detail. We conclude with the technique by Karger [5] to extend the polylogarithmic edge connectivity data structures to arbitrary large values in section 6.

2 Tree Packings

An important combinatorial object that is essential to global edge connectivity data structures is a *tree packing*. A tree packing of a graph G is a set of spanning trees of G with possible multiple occurrences. Given a tree packing $\mathcal{T}$, each edge e is then loaded with the number of trees containing it. We define a *relative load* $\ell^{\mathcal{T}}(e)$ to be the proportion of the load of an edge to the size of the tree packing:

$$\ell^{\mathcal{T}}(e) = \frac{\#\{T \mid e \in T, T \in \mathcal{T}\}}{\#\mathcal{T}}$$

The value of a tree packing is defined as follows:

$$\text{value}(\mathcal{T}) = \frac{1}{\max_{e \in E(G)} \ell^{\mathcal{T}}(e)}$$

Throughout the paper, we use τ to denote the maximal value of a tree packing.

There is a surprising connection between the values of tree packings and the edge connectivity of a graph. To illustrate this connection we need a concept of a *cut value*. Given a partition $\mathcal{P}$ of the vertices $V(G)$, the cut value of $\mathcal{P}$ is:

$$\text{cut-value}(\mathcal{P}) = \frac{\#\{e \in E(G) \mid e \text{ crosses partitions in } \mathcal{P}\}}{\#\mathcal{P} - 1}$$

Notice that the cut value of a min-cut is λ as it partitions the vertices of a graph into two sets.

Theorem 1 (Nash-Williams)**.** *The minimum cut value of a partition equals τ, the maximal value of a tree packing.*

Corollary 2 (Karger [5])**.** $\lambda/2 < \tau \leq \lambda$

Proof. The right hand inequality follows from our observation that a min-cut defines a partition with cut value λ. To show the left hand side we choose a partition $\mathcal{P}$ with cut value τ. Then by definition:

$$\tau > \frac{\#\{e \in E(G) \mid e \text{ crosses partitions in } \mathcal{P}\}}{\#\mathcal{P}}$$

For each set in the partition $\mathcal{P}$, the number of graph edges connecting the vertices in the set to the rest of the graph is at least λ. If we sum these numbers over all sets, we get $2\#\{e \in E(G) \mid e \text{ crosses partitions in } \mathcal{P}\} \geq \lambda\#\mathcal{P}$, since each edge crossing partitions is counted twice, from which inequality follows directly. □

Now we turn to the algorithmic side of the construction and maintenance of a tree packing for our graph. A greedy approach appears to be sufficient to approximate the maximal value τ of a tree packing. We call a tree packing $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ *greedy*, if each subsequent tree T_i is the minimal spanning tree with respect to the loads $\ell_{i-1}(e)$ of previous trees where the load of edge e is:

$$\ell_i(e) = \#\{T_j \mid e \in T_j, j \leq i\}$$

Theorem 3 (Young)**.** *Any greedy packing $\mathcal{T}$ with at least $3\lambda \ln m/\varepsilon^2$ trees has value*

$$\textit{value}(T) \geq (1 - \varepsilon)\tau$$

Although the greedy tree packing approximates the minimal cut value of a graph up to a factor $\sqrt{2}$, it does not give us the exact value of the edge connectivity λ. As pointed out in [4], minimal spanning trees of a graph may cross a min-cut multiple times, so there is no direct correlation between tree packings and edge connectivity. However, a deeper study of the greedy technique led to discovery of the following purely combinatorial result:

Theorem 4 (Thorup [4])**.** *Any greedy packing $\mathcal{T}$ with $\omega(\lambda^7 \log^3 m)$ trees contains a tree crossing some min-cut exactly once.*

Both data structures in [5] and [4] maintain a greedy tree packing using a dynamic minimal spanning tree data structure for each tree. Each edge update can change the weights ℓ_i for $i < k$, and hence update the minimal spanning trees at each level i. The following lemma gives as estimate on the total number of changes in trees required per a graph update. Here we assume some assignment of priorities to the edges of the graph so that the minimal spanning tree is always unique for each i.

Lemma 5 (Karger [5])**.** *Assuming the graph remains connected, each insertion or deletion of an edge changes at most i edge loads ℓ_i for each i.*

Proof. Without loss of generality it suffices to consider the deletion of an edge e. We claim that for each i the loads of edges distinct from e do not decrease. Notice $\ell_i(e) \leq i$ by definition and the load of e drops to 0 after deletion. Since we assume that the graph remains connected, the total sum of loads of edges is invariant. Therefore, the operation can only affect at most i edge loads.

Let us denote $\ell_i^{new}(e)$ the edges loads after the deletion. We prove by induction on i that $\ell_i^{new}(e) \geq \ell_i(e)$. Notice that it suffices to consider the case of an edge e belonging to the old spanning tree T_i but not to the updated one, since otherwise the edge load $\ell_i(e)$ does not decrease by induction hypothesis. In this case, since the loads ℓ_{i-1} have not decreased, and e is not in the update minimal spanning tree T_i, we must have strict inequality $\ell_{i-1}^{new}(e) > \ell_{i-1}(e)$. Then $\ell_i^{new}(e) \geq \ell_{i-1}^{new}(e) \geq \ell_{i-1}(e) + 1 \geq \ell_i(e)$. □

It follows from the lemma that each update requires at most $1+2+\ldots+(k-1) \leq k^2$ load changes. Notice that we can limit our discussion to the case of dynamic graphs remaining connected. Both of our minimum spanning trees data structures in the next section, are suitable for finding spanning forests as well. The effect of insertion or deletion of any edge between two connected components is that the bridge edge is either added or deleted from all the spanning trees of the components. Since we in general maintain polylogarithmic number of them, this has the same cost as in usual edge update. Finally, the edge connectivity of a disconnected graph is always zero; so if we have a spanning forest consisting of multiple trees we report zero in our algorithms. Thus, our goal is to maintain a list of $k = O(\text{polylog}(n))$ dynamic forests of the greedy packing. Since the total number of load changes is at most k^2, we can make it fully dynamic by choosing an appropriate dynamic minimum spanning tree structure that then could be maintained independently over the trees in the packing.

3 Dynamic Minimum Spanning Forests

The paper by Thorup and Karger [5] describes a data structure that can approximate polylogarithmic edge connectivity within a multiplicative factor $\sqrt{2}$ and perform updates in polylogarithmic amortized time. This is achieved by using a minimum spanning forest structure by Holm et al described in [3] for each of the trees in the greedy packing of size k and reporting an approximation of τ obtained from the maximal relative load $\ell_k(e)$. The inequality in lemma 2 then confirms that τ is $\sqrt{2}$-approximation of λ. The minimum spanning forest in [3] performs each edge update in $\mathrm{O}(\log^4 n)$ time per update, and is partly based on the top trees which we present next. The key observation is that lemma 3 requires a polylogarithmic packing size k and we can set the approximation factor ε sufficiently small to be ignored by picking the closest integer.

Theorem 6 ([5]). *For graphs with edge connectivity at most λ_{max} we can maintain $(1-\varepsilon)$-approximate greedy tree packing of size $3\lambda_{max} \ln m/\varepsilon^2$ in $O(\lambda_{max}^2 polylog(n)/\varepsilon^4)$ amortized time per update. The algorithm announces if the edge connectivity exceeds λ_{max}.*

Proof. The updates to the minimal spanning trees of the graph can be amortized over load changes. Since a graph update requires at most k^2 load changes, and each MST update takes polylogarithmic amortized time, the result follows from lemma 3. Clearly, if the value of the packing exceeds λ_{max}, then the edge connectivity also exceeds λ_{max}. □

Another approach, which is undertaken in [4], attempts to answer queries for the exact value of λ. Instead of using MST data structure from [3], it uses another minimum spanning forest structure by Frederickson and Eppstein et al [2] that performs updates in $O(\sqrt{n})$ worst case time. However, we would like to use a combination of lemmas 3 and 4. Again we assume only queries for edge connectivity bounded by polylogarithmic λ_{max}. Thus, we obtain the following theorem:

Theorem 7 ([4]). *For graphs with edge connectivity at most λ_{max} we can maintain a greedy tree packing of polylogarithmic size for a fully-dynamic graph in $\tilde{O}(\sqrt{n})$ worst case time per edge update such that at least one tree in the packing crosses some min-cut exactly once. The algorithm announces if the edge connectivity exceeds λ_{max}.*

Since the contribution coming from applying same operations to each tree is only a polylogarithmic multiplicative factor, we can consider each tree independently. Therefore, the result follows the lower bounds in lemmas 3 and 4. In the next sections, we show the representation of trees in the packing so that we can query the size of the min-cut crossed exactly once by each tree. It is based on the top trees data structure, which we explain next.

4 Top Trees

We describe a dynamic tree data structure called top trees which was invented by Alstrup et al in 1997 [1]. The hallmark of this structure comparing to others such as ET-trees due to Henzinger and King which are simpler to implement, is that it supports path-oriented updates and queries and is well-suitable for divide-and-conquer algorithms. A variant of the top trees is used in minimal spanning forest algorithm by Holm et al [3]. The following version follows [4] and can be used after certain augmentation to maintain the min-cut. We define the top tree for a dynamic subtree of the graph G, and we call the edges of G that do not belong to it *non-tree edges*.

The key notions of the top trees are that of *boundary nodes* and *clusters*. We start from a tree T and at most two selected nodes of T which form the set ∂T of the *external boundary nodes*. For any connected subtree C of T, we call any node in C which either belongs to ∂T or is an endpoint of an edge that leaves C a *boundary point*. We denote the set of boundary points of C as $\partial_{(T,\partial T)}C$. Given the pair $(C, \partial_{(T,\partial T)}C)$, we notice that the boundary nodes of any subtree A of C are the same as the boundary nodes of A considered as a subtree of T. Therefore, we can simplify our notation of the boundary points of C to ∂C and eliminate the context of T. We call any subtree of T with at most two boundary points a cluster. Hence, T itself is a cluster. If two clusters share exactly one node, we refer to them as *neighbors*. For a cluster C with two boundary nodes, the tree path $\pi(C)$ connecting these nodes is called *a cluster path*.

Given the pair $(T, \partial T)$, the clusters are organized into a hierarchic structure of a binary tree with the root being the tree T itself as shown in Figure 1. Each node in the binary tree is a cluster

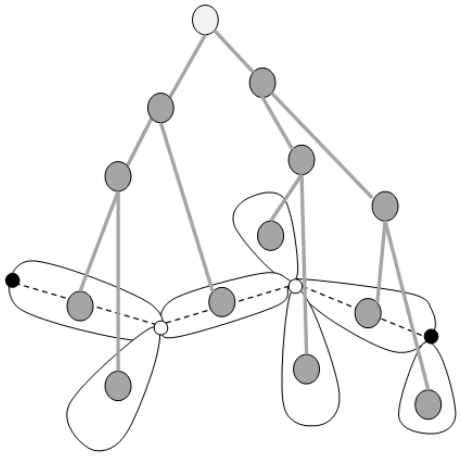

Figure 1: The binary tree structure of the top tree. The nodes marked with • are external boundaries.

of the form $(T, \partial T)$. For every internal node C with children A and B we require that clusters A and B are neighbors and C is the union of A and B as explained in Figure 2. Finally, we require that the number of vertices and incident non-tree edges of any leaf cluster to be at most $\sqrt{m}$. For the non-tree edges that have an end-point in a boundary node of a leaf cluster, we designate only one leaf cluster as the owner of the edge. This is an essential constraint for our variant of the top tree that differentiates it from the original top trees which had single edges at the leaves. Our variant is in a sense a prunned version of the cluster hierarchy which allows to assign additional information to the pairs of clusters as there only a few of them in this case.

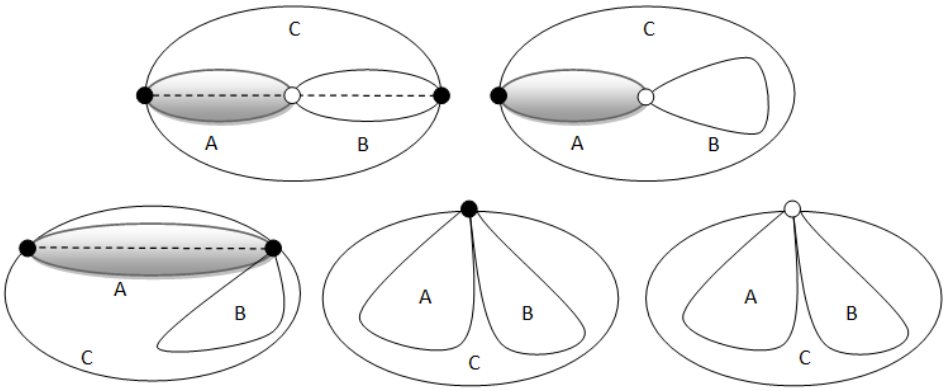

Figure 2: Various cases of arrangement of two neighbor clusters A and B and the parent node C in the top tree. The boundary nodes of the parent are marked with • while the rest of boundary nodes of children are marked with ∘. The cluster path of C is shown.

The interface to the top trees over underlying forest consists of the following *update operations*:

Link(v, w) adds the edge (v, w) to the dynamic forest assuming v and w are in different trees and sets the resulting boundary set $\partial T = \emptyset$;

Cut(v, w) removes the edge (v, w) and sets $\partial T = \emptyset$;

Expose(v, w) for v and w in the same tree, sets them to be the boundary nodes and returns the root cluster after the update; similar operation works for a single nodes or no nodes passed as arguments. One effect of this operation is that it modifies cluster paths.

The data structure translates each of above operations into a sequence of the *primitive operations*:

Create(L) where L is a cluster satisfying the last condition for leaves above creates a new top tree consisting of a single cluster L;

Destroy(L) eliminates the top tree consisting of a single cluster L;

Merge(A, B) where A and B are neighbors and the root clusters of two top trees, creates a new cluster $C = A \cup B$, makes it the common root of the single new top tree, and returns the new root cluster C;

Split(C) where C is the root cluster of a top tree, with two children A and B. It performs the inverse of the previous operation by removing C and producing two top trees rooted at A and B.

Since we concentrate on augmentation and are going to use top trees as essentially a black box, we summarize the update algorithm of the dynamic forest in the following statement:

Lemma 8 (Alstrup et al [1])**.** *We can maintain top trees of height $O(\log n)$ and with $O(\sqrt{m})$ cluster nodes supporting Link, Cut, and Expose above with a sequence of $O(\log n)$ Merge and Split and $O(1)$ Create and Destroy operations per update. The sequence itself is computed in $O(\log n)$ time.*

5 Augmenting Top Trees

Recall that our goal is to extract the min-cut crossed exactly once from a top-tree. For any edge in the tree e we say that a non-tree edge *covers* e if the path in the tree between its endpoints contains e. We denote by $c(e)$ the number of edges covering e. Notice that the partition induced by removing e from the tree has cut value $c(e) + 1$. Therefore, if a tree crosses *some* min-cut only once, the minimum of $c(e) + 1$ over all edges in the tree is exactly the edge connectivity of the graph.

Another black box we rely on in the following description is the sparsification technique by Eppstein et al [2]. Essentially, it allows a reduction of the problem to the case with $m = O(n)$ by using an hierarchy of sparse graphs preserving the properties of the given graph. Thus, we can assume from now on that $m = \tilde{O}(n)$ for our particular type of the problem with polylogarithmic edge connectivity.

Given a top tree T we would like to support the following query:

Min-cut returns $\min_{e \in T}(\tilde{c}(e) + 1)$ where $\tilde{c}(e) = c(e)$ for $c(e) \leq \sqrt{m}$ or $c(e) \geq \tilde{c}(e) \geq \sqrt{m}$, otherwise.

This query returns the correct value of the cover if it does not exceed $\sqrt{m}$ but may return smaller value, otherwise. This is a reasonable assumption for our data structure since our minimal covering is up to polylogarithmic size. Remark that it is easy to support the minimum covering and answer the query once we know how to update every value $\tilde{c}(e)$ for each graph update operation. Also, for operations involving tree edges, we can use the above algorithm to produce a sequence of logarithmic number of primitive operations. Therefore, it suffices to show how we can maintain all values $\tilde{c}(e)$ for each of Create, Destroy, Merge, and Split operations and non-tree edge updates in $\tilde{O}(\sqrt{m})$ time.

We introduce the following augmentation to the top trees:

- For each pair of clusters A and B we maintain the number of non-tree edges between A and B. We use notation $E(A, B)$ to denote the set of these edge such that every edge $(v, w) \in E(A, B)$ has $v \in A$ and $w \in B$. For each pair of leaf clusters we maintain $E(A, B)$ directly.
- For a cluster C, we maintain $\tilde{c}(e)$ for edges $e \in C$ not on the cluster path $\pi(C)$. In addition, for edges e on the cluster path we keep the covering of e up to $\tilde{c}(e)$ by edges with both end-points in C.

Notice that for any pair of clusters, not necessarily leaf clusters, we can obtain the set of edges between them by following the counters of the clusters down the top tree to the leaves and output them in $\mathrm{O}(\log n)$ time per edge according to lemma 8. A reasoning behind distinction between leaf and internal nodes is economy of space the sets $E(A, B)$ occupy since we can afford additional $\log n$ factor for edge enumeration. For each of the primitive operation we describe the necessary update required to support the counters and sets $E(A, B)$:

Create(L), **Destroy**(L) : Since leaf clusters have at most $\sqrt{m}$ vertices and incident edges, and each edge in it belongs to at most $\mathrm{O}(\log n)$ clusters, we need to update $\mathrm{O}(\sqrt{m} \log n)$ counters and sets by adding or removing each of these edges.

Merge(A, B): For each cluster C among $\mathrm{O}(\sqrt{m})$ clusters in the top tree, we simply set

$$\#E(A \cup B, C) = \#E(A, C) + \#E(B, C)$$

Split(C): There is nothing to update in this case.

Thus, the first augmentation is supported in $\tilde{O}(\sqrt{m})$ time per update. The second augmentation allows us to retrieve the values $\tilde{c}(e)$ for each edge in the tree by looking at the root cluster since all edges must connect nodes within it. Observe that all edges in any cluster outside the cluster path can only be covered by edges incident to the cluster, but that is not true for edges in the cluster path. This is the primary reason why we differentiate between these two types of edges. We make the following changes to the primitive operations:

Create(L): This is straightforward from the above observation.

Destroy(L): No changes required.

Merge(A, B): To handle this operation, we need to consider various cases of two neighbors forming a cluster as in Figure 2. If a cluster A does not have a cluster path $\pi(A)$ then merging it with any other cluster does not affect coverings within A. Therefore, we consider only the

cases of A having a cluster path, i.e. the shaded clusters in the figure. Now the resulting cluster C might also have a cluster path, so we consider these two cases separately:

C has a cluster path.
We need to increment covering of edges in $\pi(A)$ by cross-cluster edges $E(A, B)$. Since we know the counter $\#E(A, B)$, we can compare $\#E(A, B)$ to $\sqrt{m}$.

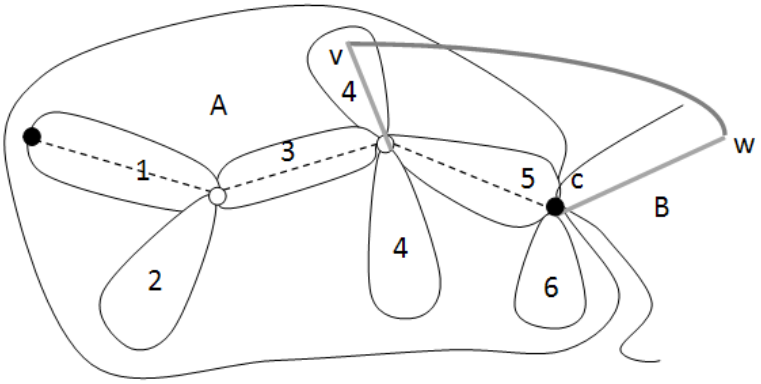

Figure 3: The ordering of the leaves in a neighbor cluster across its marked cluster path. We are indifferent between leaves with the same assigned number.

If $\#E(A, B) > \sqrt{m}$, we want to pick only a subset of edges $\tilde{E}(A, B)$ which maximizes the number of covered edges in the path $\pi(A)$. We can do that by traversing the leave clusters of the top subtree rooted at A in the order shown in Figure 3. Notice that for any leaf cluster L in A, each edge in $E(L, B)$ covers same number of path edges in $\pi(A)$. We visit the leaf clusters in the decreasing order of these numbers, where we choose leaf clusters with the same number arbitrarily. We construct the set $\tilde{E}(A, B)$ by adding all edges $E(L, B)$ every time we visit a leaf, until the size $\tilde{E}(A, B)$ exceeds $\sqrt{m}$. Therefore, since we can output any edge in $E(L, B)$ in $\mathrm{O}(\log n)$ time, the selection procedure takes $\tilde{O}(\sqrt{m})$. Since each leaf has less than $\sqrt{m}$ incident non-tree edges, we end up with a set $\tilde{E}(A, B)$ such that:

$$\sqrt{m} \leq \#\tilde{E}(A, B) < 2\sqrt{m}$$

If $\#E(A, B) \leq \sqrt{m}$ then we can simply set $\tilde{E}(A, B) = E(A, B)$. We can justify our reduction of $E(A, B)$ by the fact that each contribution of an edge in $E(A, B) \setminus \tilde{E}(A, B)$ is made to an edge of the path $\pi(A)$ which is already covered by every edge in $\tilde{E}(A, B)$. Therefore, for $\#\tilde{E}(A, B) \geq \sqrt{m}$, our value $\tilde{c}(e)$ is allowed to be less than actual covering $c(e)$ since it is at least $\#\tilde{E}(A, B)$.

According to [4], using either link-cut trees by Sleator and Tarjan or extending our top tree structure we can for any path in the tree add a value to all its edges in $\mathrm{O}(\log n)$ time. For every edge (v, w) in $\tilde{E}(A, B)$ we need to increment the values of the cluster path edges:

$$\pi(A) \cap (v \to w)$$

where $v \to w$ denotes a path from v to w within the tree. If we denote the common boundary node of A and B as c, this can be done by incrementing each edge in the path $v \to w$ by

1/2 and decrementing edges in the path $c \to w$. Repeating this procedure for every edge in $\tilde{E}(A, B)$, we complete Merge procedure in $\tilde{O}(\sqrt{m})$ time.

C does not have a cluster path.
In this case, we perform the above procedure to increment the covering of path edges in $\pi(A)$ by edges within the cluster C. After that, we repeat the same procedure but now with the neighbor cluster being the complement $V \setminus C$. There is no difficulty in doing that since the set counters are $\#E(L, V \setminus C)$ can be calculated for any cluster in C by negating the $\#E(L, C)$ from m, and all edges can be listed at the cost $\mathrm{O}(\log n)$ per edge.

Split(C): This operation essentially reverses our procedure above. To do that efficiently, we can story the set $\tilde{E}(A, B)$ at every internal node $A \cup B$ after each Merge operation, and then use it to undo all the updates to the edge values $\tilde{c}(e)$.

Returning to the question of non-tree edge updates, we observe that each such edge affects the values in $\mathrm{O}(\log n)$ clusters to which its end-points belong, and for a single cluster we can perform the update in $\mathrm{O}(\sqrt{m})$ time. Thus, we are able to maintain $\tilde{c}(e)$ for each edge under all graph updates.

6 Randomized Sparsification

A general method that allows us to extend a global edge connectivity data structure from poly-logarithmic λ to larger values is presented in paper [5]. The technique uses a sequence of random subgraphs $G(p)$ in which each edge of G appears independently with probability p. We maintain of sequence of such sparse subgraphs for $p = 1, \frac{1}{2}, \frac{1}{4}, \ldots$. For each of these subgraphs, we maintain a data structure which supports edge connectivity up to $\log^2 n$. The edge updates are performed on each of these subgraphs. A result by Karger then correlates the value of any cut in $G(p)$ to its expected value. Precisely, for $p\lambda \geq 6 \ln n/\varepsilon^2$ the values differ by multiplicative factor $(1 + \varepsilon)$ with probability $\mathrm{O}(1/n)$. Therefore, with high probability the approximate edge probability of G is the edge probability of $G(p_{max})$ divides by p_{max}, where p_{max} is the highest value for which the edge connectivity query does not announce that it is too large.

Using this technique, both results can be extended to arbitrary values of λ. However, the exact solution by Thorup can only output $(1 \pm \varepsilon)$-approximate value of the edge connectivity due to randomization we introduce. It is left as an open question in [4] whether there exists a data structure that can give an exact value of λ in general case.

References

[1] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In Proc. 24th International Colloquium on Automata, Languages, and Programming (ICALP), pages 270–280, 1997

[2] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. In CRC Handbook of Algorithms and Theory of Computation, Chapter 22. CRC Press, 1997

[3] Jacob Holm, Kristian de Lichtenberg, Mikkel Thorup: Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. J. ACM 48(4): 723-760 (2001)

[4] Mikkel Thorup: Fully-dynamic min-cut. STOC 2001: 224-230

[5] Mikkel Thorup, David Karger: Dynamic Graph Algorithm with Applications. SWAT 2000: 667-673