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

[13] D. Karger, "Using randomized sparsification to approximate minimum cuts" *Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1994, 424–432.

[14] D. Karger, "Random sampling in cut, flow, and network design problems", *Proc. 26rd Annual Symposium on Theory of Computing*, 1994, 648–657.

[15] H. La Poutré, "Maintenance of 2- and 3-connected components of graphs, Part II: 2- and 3-edge-connected components and 2-vertex-connected components", Tech.Rep. RUU-CS-90-27, Utrecht University, 1990.

[16] D. W. Matula, "A linear time $2 + \epsilon$ approximation algorithm for edge connectivity" *Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1993, 500–504.

[17] H. Nagamochi and T. Ibaraki, "Linear time algorithms for finding a sparse k-connected spanning subgraph of a k-connected graph", *Algorithmica* 7, 1992, 583–596.

[18] J. Westbrook and R. E. Tarjan, "Maintaining bridge-connected and biconnected components on-line", *Algorithmica* (7) 5/6, 1992, 433–464.

Since κ increases by a factor of $(1+\epsilon)$ per phase, there are $(\log \kappa)/\epsilon$ phases. Thus, the total time for the algorithm is $O(m_0 + \kappa n^2/\epsilon + m_1 n) = O(mn/\epsilon)$. ∎

Note: The same technique can be used to maintain the (exact) minimum edge cut $\lambda(s,t)$ or the minimum vertex cut $\kappa(s,t)$ between two given nodes s and t of G in amortized time $O(\lambda(s,t))$, resp. $O(\kappa(s,t))$ per insertion.

Acknowledgements

We want to thank David Karger for useful discussions.

References

[1] J. Cheriyan, M. Y. Kao, and R. Thurimella, "Scan-first search and sparse certificates—an improved parallel algorithm for k-vertex connectivity", *SIAM Journal on Computing*, 22, 1993, 157–174.

[2] E. A. Dinitz, A. V. Karzanov, and M.V. Lomonosov, "On the structure of the system of minimal edge cuts in a graph", *Studies in Discrete Optimization*, 1990, 290–306.

[3] Ye. Dinitz, "Maintaining the 4-edge-connected components of a graph on-line", *Proc. 2nd Israel Symposium on Theory of Computing and Systems (ISTCS'93)*, IEEE Computing Society press, 1993, 88–97.

[4] Ye. Dinitz and A. Vainshtein, "The connectivity carcass of a vertex subset in a graph and its incremental maintenance", *Proc. 26nd Annual Symposium on Theory of Computing*, 1994, 716–725.

[5] S. Even, "An algorithm for determining whether the connectivity of a graph is at least k" *SIAM Journal on Computing*, 4, 1975, 393–396.

[6] S. Even and R. E. Tarjan, "Network flow and testing graph connectivity", *SIAM Journal on Computing*, 4, 1975, 507–518.

[7] M. L. Fredman and M. E. Saks, "The cell probe complexity of dynamic data structures", *Proc 21st Annual Symposium on Theory of Computing*, 1989, 345–354.

[8] H. N. Gabow, "A matroid approach to finding edge connectivity and packing arborescences" *Proc. 23rd Annual Symposium on Theory of Computing*, 1991, 112–122.

[9] H. N. Gabow, "Applications of a poset representation to edge connectivity and graph rigidity" *Proc. 32nd Annual Symposium on Foundations of Computer Science*, 1991, 812–821.

[10] Z. Galil, "Finding the vertex connectivity of graphs", *SIAM Journal on Computing*, 1980, 197–199.

[11] Z. Galil and G. P. Italiano, "Maintaining the 3-edge-connected components of a graph on-line" *SIAM Journal on Computing*, 1993, 11–28.

5.4 An incremental $(2+\epsilon)$-approximation of the minimum vertex cut

Using the generic incremental algorithm, we construct an incremental $(2+\epsilon)$-approximation algorithm which takes time $O(m_0 + (\kappa n^2 \log \kappa)/\epsilon)$ to build a minimum vertex cut algorithm. Since $\kappa n = O(m)$, the amortized time per insertion is $O(n/\epsilon)$ if the algorithm is started on an empty graph.

Let $b_1, \ldots b_{n/d}$ be all the nodes b created by the static 2-approximation algorithm. To quickly test if the current solution is still correct, we maintain the minimum degree in G and update the $\kappa(a, b_i)$-values and their minimum after each insertion. If the minimum degree in G has been increased by a factor of $(1+\epsilon)$, we compute a new solution. The pair algorithm that computes $\kappa(a, b_i)$ computes a maximum flow in a directed graph. We store the maximum flow for each pair (a, b_i). After each insertion we find the pair (a, b_i) such that $\kappa(a, b_i)$ is minimum and try to augment the maximum flow from a to b_i. For this purpose we try to compute a blocking flow from a to b_i using dynamic trees. If we keep a path from a to every node reachable from a, then the total time spent for incrementally computing a blocking flow is $O(m+n)$. This leads to the following algorithm.

An Incremental $(2+\epsilon)$-Approximation Algorithm

1. Determine the minimum degree δ of G, compute a msfd $F_1, \ldots F_m$ of G and replace G by $\cup_{i \leq \delta} F_i$.
   Compute a 2-approximation k of G using the static algorithm and keep the maximum flow and
   For each b_i keep a path from a to all nodes reachable from a in the residual graph of the maximum flow in a dynamic tree.
   while the minimum degree in G is $< \delta(1+\epsilon)$ or $k < \delta/2$ **do**
      if the next operation is a query **then**
         return k
      else
         Try to augment the maximum flow $\kappa(a, b_i)$ by adding e to its residual graph.
         Maintain k as the minimum of all $\kappa(a, b_i)$.
   endwhile
   Goto 1.

Theorem 5.9 *Let G be a graph with n nodes and let m_0 be the number of initial edges. The total time for maintaining a $(2+\epsilon)$-approximation of the minimum vertex cut κ of G during m_1 insertions is*

$$O(m_0 + \kappa n^2/\epsilon + m_1 n).$$

Each query can be answered in constant time.

Proof: Let $k_1, \ldots, k_f$ be the values that k assumes during the algorithm. Let Phase i be all the steps executed while $k = k_i$ and let ins_i be the number of insertions during Phase i. We show first that the total time spent in Phase i for maintaining $\kappa(a, b_j)$ for a fixed j is $O(k_i(\delta n + ins_i))$. It takes time $O(\delta n + ins_i)$ to increment $\kappa(a, b_i)$ by one. Since $\kappa(a, b_j) \leq k_i$, we spend $O(k_i(\delta n + ins_i))$ time for each j. Thus, the total time spent in Phase i is $O(k_i(n^2 + ins_i n/\delta)) = O(k_i n^2 + ins_i n)$.

Create a new graph G' by connecting in G' all neighbors of b by n parallel edges to a and discard b.

endwhile

return G is k-connected.

5.2 Correctness

The next two lemmata are extensions of lemmata in [5]. Essentially they show that instead of computing the $\kappa(s,t)$ for every pair s and t of vertices of G, it suffices to compute $\kappa(a,b)$ for every node b. The minimum of these minimum vertex cuts is $\kappa(G)$.

Lemma 5.2 *Let $L = \{v_1, \ldots, v_d\}$ be a set of nodes that are pairwise d-connected in G. Let G' be the graph created from G by creating a new node a and adding n edges (a, v_i) for every i and let $k \leq d$. Then $\kappa(a,u) \geq k$ for every node $u \notin L$ in G' iff $\kappa(G) \geq k$.*

Proof: Assume there exists a partition X, C, and Y of V such that there is no edge between a node of X and a node of Y and $|C| < k$. Since $|L \cup \{a\}| > |C|$, the nodes of $L \cup \{a\}$ are either contained in $X \cup C$ or in $Y \cup C$. This implies that a is either contained in X or in Y and, thus, there exists a node $u \notin L$ such that $\kappa(a,u) = |C| < k$ in G'. The same argument can be used to show the existence of a cut of size $< k$ in G if $\kappa(a,u) < k$ in G. ∎

Lemma 5.3 *Let $L = \{v_1, \ldots, v_d\}$ be a set of nodes that are pairwise d-connected in G. Let G'' be the graph created from G' by adding a vertex b and edges (b, v_i) for every i and let $k \leq d$. Then $\kappa(a,b) \geq k$ in G'' iff $\kappa(a, v_i) \geq k$ for every i in G'.*

To complete the proof of correctness we have to show that the last $\delta/2$ nodes in mcs-order are indeed $\delta/2$-connected.

Lemma 5.4 *The last $\delta/2$ nodes in mcs-order are connected in $F_{\delta/2}$.*

Lemma 5.5 *All nodes that are connected in F_i are i-connected.*

Corollary 5.6 *The last $\delta/2$ nodes scanned in the msfd algorithm are $\delta/2$-connected.*

5.3 Running Time Analysis

The following lemma shows that each execution of the while loop, except for possibly the last one, adds $\delta/2$ nodes to the neighborhood of a. Thus, $2n/\delta$ executions of the loop suffice.

Lemma 5.7 *If there are more than $\delta/2$ non-neighbors of a in G', then the last $\delta/2$ nodes scanned by the msfd algorithm are non-neighbors of a. Otherwise, all nodes not incident to a are contained in the last $\delta/2$ nodes.*

Theorem 5.8 *A 2-approximation of the minimum vertex cut can be computed in time $O(n^2 \min(\sqrt{n}, \kappa))$, where κ is the size of the minimum vertex cut.*

Proof: The previous lemma shows that $2n/\delta$ executions of the while-loop suffice. The first execution of the msfd algorithm takes time $O(m)$, all further take time $O(\delta n)$, even with the added parallel edges. The parallel edges can be ignored when computing the $\kappa(a,b)$. Thus, one iteration of the while-loop takes time $O(\delta n \min(\sqrt{n}, \kappa))$ which gives a total time of $O(n^2 \min(\sqrt{n}, \kappa))$. ∎

5 A 2-Approximation of the Minimum Vertex Cut Algorithm

5.1 A Static Algorithm

Let δ be the minimum degree in G, let κ be the vertex connectivity of G, and let $d = \delta/2$. We give an algorithm that given G tests if $\kappa \leq \delta/2$ in time $O(n^2 \min(\sqrt{n}, \kappa))$ and if $\kappa \leq \delta/2$, it outputs the minimum vertex cut in G. Thus, the algorithm computes a 2-approximation of the minimum vertex cut. To our knowledge, there was no approximation algorithm for this problem known, the best algorithm to compute the exact solution takes time $O(\kappa^3 n^{3/2} + \kappa^2 n^2)$ [10, 1, 17]. This is an improvement of κ if $\kappa \leq \sqrt{n}$ and of $\kappa^3/n \geq \kappa$, otherwise.

Even and Tarjan [6] give an algorithm to compute the minimum vertex cut $\kappa(a, b)$ between two given nodes a and b in time $O(m\sqrt{n})$, which we refer to as the *pair algorithm*. Galil's minimum vertex cut algorithm [10] makes $O(k^2 + n)$ calls to the pair algorithm. Our algorithm reduces the number of calls to n/d using the following two observations:

1. The last d nodes in the mcs-order are pairwise d-connected
2. Given d pairwise d-connected nodes $L = \{v_1, \ldots, v_d\}$ and a node $a \notin L$, if a node b is added to G and connected to every v_i, then a is d-connected to each v_i iff a is d-connected to b.

Thus, we use the decomposition algorithm DA to find d pairwise d-connected nodes and test if all of them are d-connected to a with one call to the pair algorithm. To guarantee that each call to DA produces different d-connected nodes, we start DA at a and connect each set of nodes with n parallel edges to a after its minimum vertex cut with a has been computed. This guarantees that the msc-order determined by DA starts with a followed by all nodes that are connected to a by n parallel edges. These ideas are similar to Matula's $(2 + \epsilon)$-approximation algorithm for the minimum edge cut [16].

To improve the performance of the pair algorithm we first "sparsify" G using the following lemma.

Lemma 5.1 [17] *Let $F_1, \ldots, F_m$ be a maximal spanning forest decomposition of a graph G and let δ be the minimum degree in G. The graph G is k-vertex connected iff $F_1 \cup \ldots \cup F_\delta$ is k-vertex connected.*

This leads to the following algorithm.

A 2-Approximation Algorithm for the Minimum Vertex Cot
Compute a msfd $F_1, \ldots F_m$ of G and replace G by $\cup_{i \leq \delta} F_i$.
Create a graph G' by adding a node a to G and connecting a by n parallel edges to each of the last $\delta/2$ nodes in mcs-order.
$k = \delta/2$;
while there exists a node in G' not incident to a **do**
    Compute a msfd of G' starting at a.
    Create a new node b and connect b to each of the last $\delta/2$ nodes scanned by the msfd algorithm by an edge. Call the new graph G''.
    Compute the minimum vertex cut $\kappa(a, b)$ between a and b in G''.
    if $\kappa(a, b) < k$ **then**
        $k = \kappa(a, b)$;

Compute for each minimum cut in $G(p)$ the size of the corresponding cut in G and store all the sizes of these cuts in a heap h.

2. $N = \emptyset$;
   while there is > 1 node in the cactus-tree **do**
   if the next operation is a query **then**
   return the minimum value in h
   else
   Sample the new edge with probability p and if sampling is successful, insert the new edge into the cactus tree and into N and remove the values corresponding to discarded minimum cuts in $G(p)$ from the heap h.
   endwhile
3. Add all edges of N to the incremental $(2+\epsilon)$-approximation algorithm and determine a new 3-approximation k'.
   if $k' \geq 3k$ **then**
   $k = k'$, $p = 8(\ln n)/(k\epsilon^2)$.
   Construct $G(p)$ by sampling every edge with probability p.
   Compute the size λ of the minimum cut of $G(p)$, augment the complete intersection to become a λ intersection, and construct a new cactus-tree.
   Compute for each minimum cut in $G(p)$ the size of the corresponding cut in G and store all the sizes of these cuts in a heap h.
   Goto 2.

Lemma 4.7 *The value returned by the algorithm is a $(1+\epsilon)$-approximation of λ with high probability for $\epsilon \leq 1$.*

Proof: Since $k \leq \lambda$ it follows that $p = 8 \ln n/k \geq 8 \ln n/\lambda$. Since increasing p increases the probability that the previous lemma holds, it follows that with high probability the incremental algorithm returns a $(1+\epsilon)$-approximation of the minimum cut in G. ∎

Theorem 4.8 *Let G be a graph with n nodes and let m_0 be the number of initial edges. The given algorithm maintains a $(1+\epsilon)$-approximation of the minimum edge cut of G with probability $1 - O(1/n)$. The total expected time for inserting m_1 edges is*

$$O((m_0 + m_1)(\log n)^2(\log \lambda)/\epsilon^2)$$

where λ is the size of the minimum cut in the final graph. Each query can be answered in constant time.

Proof: The expected size of λ is $O(\log n/\epsilon^2)$. Thus, the expected time for Step 1 is $O(m + n((\log n)/\epsilon)^2)$.

Let $k_1, \ldots k_f$ be the values that k assumes during the execution of the algorithm. *Phase i* of the algorithm consists of all steps executed while $k = k_i$. The same argument as in the previous section implies the time in each phase is dominated by the time for computing a complete intersection and a cactus tree in this phase, which is $O(\lambda m \log n) = O(m(\log n)^2/\epsilon^2)$.

Let λ be the size of the minimum cut in the final graph. Note that there are $O(\log \lambda)$ phases, since k increases by a factor of 3 in every phase and $k_f = O(\lambda)$. Thus, the total expected time for all phases is $O((m_0 + m_1)(\log n)^2(\log \lambda)/\epsilon^2)$. ∎

Let $\lambda_0, \ldots, \lambda_f$ be the values that λ assumes in Step 2 during the execution of the algorithm in increasing order. We define *Phase* i to be Step 2 executed while $\lambda = \lambda_i$ and the following Step 3. Let ins_i be the number of insertions in Phase i.

In Phase i we augment the complete intersection and maintain the cactus tree. The total time for the later part is $O(n \log n + (n + ins_i)\alpha(ins_i, n)) = O(n \log n + ins_i \alpha(n, n))$.

To augment the complete intersection we repeatedly double λ, compute a msfd, and augment the complete intersection by at most $\lambda/2$. Computing the msfd takes time $O(m_0 + m_1)$. If $\lambda_{i+1} \leq 2\lambda_i$, augmenting the intersection takes time $O((\lambda_{i+1} - \lambda_i)\lambda_i n \log n)$.

If $2^{p-1}\lambda_i < \lambda_{i+1} \leq 2^p \lambda_i$ for $p \geq 2$, the while-loop in Step 3 is executed p times. Since the i'th augmentation for $i' \leq p$ is executed on a graph with $O(\lambda_i 2^{i'} n)$ edges, it takes time $O((\lambda_i 2^{i'+1} - \lambda_i 2^{i'})\lambda_i 2^{i'} n \log n)$. Thus, all augmentations in Phase i sum up to $O((\lambda_{i+1} - \lambda_i)\lambda_{i+1} n \log n)$.

Thus, the total work in all phases gives a telescoping sum which adds up to $O(\lambda(m_0 + m_1) + \lambda^2 n \log n)$. ∎

Note: Since each complete λ intersection consists of λ arborescences and a deletion of an edge can only disconnect one arborescence, this algorithm can also be extended to an fully dynamic algorithm, with $O(m + n \log n)$ deletion and $O(\lambda \log n)$ insertion time.

4.3 A randomized $(1+\epsilon)$-approximation for the minimum cut

In this section we present an incremental randomized Monte Carlo algorithm that maintains a $(1+\epsilon)$-approximation of the minimum cut in expected time $O((\log \lambda)((\log n)/\epsilon)^2)$ per insertion.

Karger pointed out in [13] that dynamically approximating connectivity can be reduced to dynamically maintaining exact connectivity in $O(\log n)$-connected graph using randomized sparsification. We use this idea to maintain a $(1+\epsilon)$-approximation of the minimum cut as follows: We put each inserted edge with probability p into a subgraph $G(p)$ of G and maintain $G(p)$ incrementally. As stated in the following lemma, with high probability the minimum cut of $G(p)$ (1) has size $O((\log n)/\epsilon^2)$ and (2) is a $(1+\epsilon)$-approximation of the minimum cut. Thus, the resulting incremental algorithm maintains a $(1+\epsilon)$-approximation of the minimum cut with high probability.

Lemma 4.6 [13] *Let G be any graph with minimum cut λ and let $p = 8(\ln n)/(\epsilon^2 \lambda)$, where $\epsilon \leq 1$. (1) With probability at least $1 - O(1/n^2)$ the minimum cut in $G(p)$ has value between $(1-\epsilon)p\lambda$ and $(1-\epsilon)p\lambda$. (2) With high probability the minimum cut in $G(p)$ corresponds to a $(1+\epsilon)$-minimal cut of G.*

This leads to the following algorithm:

An Incremental $(1+\epsilon)$-Approximation Algorithm

1. Compute a 3-approximation k of the size of the minimum cut in G using the incremental $(2+\epsilon)$-approximation algorithm.
   $p = 8(\ln n)/(k\epsilon^2)$.
   Construct $G(p)$ by sampling every edge with probability p. Compute the size λ of the minimum cut in $G(p)$, a complete λ intersection, and build the cactus tree of $G(p)$.

cut size λ contains more than one node. This gives an efficient test if the current solution is still correct and also if two nodes are separated by a minimum cut.

To quickly compute the cactus tree representation we use an algorithm by Gabow [9]. This algorithm converts G into a directed graph G' by making each edge bidirectional and then computes a subgraph of G, called *complete $\lambda(G)$ intersection*. Given a complete $\lambda(G)$ intersection the cactus tree can be computed in time $O(m)$. This algorithm has the property that given a complete k intersection for some integer k a complete $\lambda(G)$ intersection can be computed in time $O((\lambda(G) - k)m)$. This is called *augmenting the complete intersection*. Thus, it allows to quickly compute a new complete intersection given a previous solution. We compute msfd as before to guarantee that Gabow's algorithm is executed on a graph of size $O(\lambda(G)n)$. This leads to the following algorithm.

An Incremental Minimum Edge Cut Algorithm

1. Compute the size λ of the minimum cut, a msfd $F_1^{(1)}, \ldots, F_m^{(1)}$ of order m, a complete λ intersection, and build a cactus-tree.
   $j = 1$;
2. **while** there is > 1 node in the cactus-tree **do**
   if the next operation is a query **then**
   return λ
   else
   insert the new edge into the cactus tree.
   endwhile
3. Compute a msfd $F_1, \ldots, F_m$ of order m of G;
   $\lambda' = \lambda$;
   while $\lambda' = \lambda$ **do**
   $\lambda = 2\lambda$, $j = j + 1$;
   Compute the size λ' of the minimum cut of $\cup_{i \leq \lambda} F_i$ and augment the complete intersection to become a λ' intersection.
   endwhile
   Construct a new cactus-tree.
   Goto 2.

Theorem 4.5 *Given a graph G with n nodes and m_0 the total time for inserting m_1 edges and maintaining a minimum edge cut of G is*

$$O(\lambda(m_0 + m_1) + \lambda^2 n \log n),$$

where λ is the size of the minimum cut in the final graph. The size of the minum cut can be output in constant time, a query if two given nodes are separated by a minimum cut can be answered in amortized time $O(\alpha(m, n))$.

Started on an empty graph the time for m_1 insertions is $O(\lambda m_1 \log n)$.

Proof: Computing the complete λ_0 intersection and the cactus tree in Step 1 takes $O(m_0 + \lambda_0^2 n \log n)$.

For $j \geq 0$ let x be the number of insertions and n' be the number of nodes in G' after the jth and before the $j+1$st execution of Step 3. Execution $j+1$ of Step 2 takes time $O(x + kn'/\epsilon)$ since each execution of the repeat-loop reduces the number of edges in G' by a factor of $(1-\epsilon)$. Note that $x = 0$ for $j = 1$, since k is a $(2+\epsilon)$-approximation implies that $m \geq k/2n$. For $j > 1$, $x \geq (k_i/2)n' - p(n'-1) = \Omega(\epsilon k_i n')$. Thus, the time spent during execution $j+1$ is $O(x/\epsilon^2)$. Thus, the total time for Phase i is $O(k_i n/\epsilon + ins_i/\epsilon^2)$. ∎

Now we can bound the time for all insertions.

Theorem 4.4 *Given a graph with n nodes and m_0 the total time for inserting m_1 edges and maintaining a $(2+\epsilon)$-approximation of the size of the minimum cut is*

$$O(m_0/\epsilon + (m_1 + \lambda n)/\epsilon^2),$$

where λ is the size of the minimum cut in the final graph and $0 < \epsilon \leq 4$.

Proof: The running time for all operations is the time for Step 1 plus the time spent in all phases. Step 1 takes time $O(m_0/\epsilon)$. The time spent in all phases is $O(\sum_i k_i n/\epsilon + ins_i/\epsilon)$. Since $\sum_{i \leq q} k_i = O(\lambda/\epsilon)$ and since all ins_i sum to m_1, the total time is $O(m_0/\epsilon + \sum_i (k_i n/\epsilon + ins_i/\epsilon^2) = O(m_0/\epsilon + (m_1 + \lambda n)/\epsilon^2)$. ∎

4.2 An incremental exact algorithm

In this section we present a deterministic incremental algorithm that maintains the (exact) size λ of the minimum edge cut in amortized time $O(\lambda \log n)$ per insertion.

The basic idea for testing efficiently if the current solution is still correct is to compute and store *all* minimum edge cuts when λ is increased by 1. If an insertion increments the size of one of these cuts, it is no longer minimum. Thus, the current solution is correct as long as there exists still a minimum cut whose size has not been increased.

To store all minimum edge cuts we use the *cactus tree* representation [2]. A cactus tree of a graph $G = (V, E)$ is a graph $G_c = (V_c, E_c)$ with a weight function w defined as follows:

There is a mapping $\phi : V \to V_c$ such that

(1) every node in V maps to exactly one node in V_c and V_c corresponds to a possibly empty subset of V,

(2) $\phi(u) = \phi(v)$ iff u and v are at least $\lambda(G) + 1$- edge connected,

(3) each minimum cut in G_c corresponds to a minimum cut in G, each minimum cut in G corresponds to at least one minimum cut in G_c.

(4) If λ is odd, every edge of E_c has weight λ and G_c is a tree. If λ is even, two simple cycles of G_c have at most one common node, every edge that does not belong to a cycle has weight λ, and every edge that belongs to a cycle has weight $\lambda/2$.

Galil and Italiano [11] gave a data structure to maintain the 3-edge connected components of a connected graph under edge insertions La Poutré [15] extended the result to undirected graphs. (Our algorithm can use either.) As observed by Dinitz [3], these data structures maintain a decomposition of a graph into simple cycles that share at most one vertex under edge insertions. Thus, they can be used to maintain the cactus tree under edge insertions. The definition of a cactus tree implies that there exists a cut of size λ in the current graph iff the cactus-tree built for minimum

while $m' < (k/2)n'$ **do**

if the new operation is a query **then return** k

else add the inserted edge to N and to E'.

endwhile

3. **repeat**

$k = k(1+\epsilon')$, $p = \lceil k/(2+\epsilon') \rceil + 1$, $V' = V$, $E' = N \cup \cup_{l \le j} \cup_{q \le p} F_q^{(l)}$.

Contract all edges in all forests $F_p^{(l)}$ with $l \le j$.

repeat

$j = j + 1$.

Compute a msfd $F_1^{(j)}, \ldots, F_{m'}^{(j)}$ of $G' = (V', E')$ of order m' using DA.

Contract all edges in the forest $F_p^{(j)}$ with $p = \lceil k/(2+\epsilon) \rceil + 1$.

until $m' \le \lceil k/(2+\epsilon') \rceil (n'-1)$

until $n' > 1$

Goto 2.

Since the running time of the algorithm is dominated by the time for computing all msfd's and we have to store all of them, the space is proportional to the running time. We first show the correctness of the algorithm and then we analyze its running time.

Lemma 4.1 *The value k returned by the algorithm is a $(2+\epsilon)$-approximation of λ for $O < \epsilon \le 4$.*

Proof: We show by induction that $k/(2+\epsilon) \le \lambda \le k$. Initially k is a $(2+\epsilon)$-approximation and, thus, $k/(2+\epsilon) \le \lambda \le k$ before the first insertion. Assume that the claim holds and consider the next insertion. Note that insertions only increase λ. We distinguish two cases: (1) If $m' < (k/2)n'$ after the insertion, then Lemma 3.2 shows that $\lambda < k$ also after the insertion. (2) If $m' = (k/2)n'$ after the insertion, then the algorithm repeatedly multiplies in Step 3 k by $(1+\epsilon)$ until it finds the smallest k such that the contractions stop with $m' \le \lceil k/(2+\epsilon) \rceil (n'-1)$ and $n' > 1$. This implies that there exists a node in G' with degree less than $2\lceil k/(2+\epsilon) \rceil \le k+1$. Thus, $\lambda \le k$ and $\lambda \ge \lceil k/(2+\epsilon') \rceil / (1+\epsilon') \ge k/(2+4\epsilon') = k/(2+\epsilon)$ for $\epsilon' \le 1$. ∎

Lemma 4.2 *Let $k(j)$ be the value of k after the jth and before the $j+1$st msfd is computed. For $p \ge \lceil k(j)/(2+\epsilon') \rceil + 1$ the graph $\cup_{l \le j} F_p^{(l)}$ is a forest.*

We show next that the total running time of the algorithm is $O(m_0/\epsilon + (m_1 + cn)/\epsilon^2)$, where m_0 is the number of edges in the initial graph and m_1 it the total number of insertions. Let k_0 be the initial value of k returned by the static algorithm and let $k_i = k_0(1+\epsilon')^i$. We denote by *Phase i* all steps that are executed while $k = k_i$. Let ins_i be the number of insertions during phase i.

Lemma 4.3 *Phase i takes time $O(k_i n/\epsilon + ins_i/\epsilon^2)$.*

Proof: Let j_i be the value of j at the beginning of Phase i and let $p = \lceil k_i/(2+\epsilon) \rceil + 1$. By the previous lemma the edges in $\cup_{l \le j_i} F_p^{(l)}$ forms a forest of G. Thus, there are at most $n-1$ edges in $\cup_{l \le j} F_p^{(l)}$, contracting all of them takes time linear in the number of edges in $N \cup \cup_{l \le i} \cup_{j \le p} F_j^{(l)}$, which is $O(k_i n + ins_i)$. The resulting graph has $O(k_i n)$ edges.

The difficulty lies in deciding how to quickly test if the current solution is still correct and how to efficiently compute a new solution using previous solutions.

3 Basic definitions

A *maximal spanning forest decomposition (msfd) of order k* is a decomposition of a graph G into k edge-disjoint spanning forests F_i, $1 \leq i \leq k$, such that F_i is a maximal spanning forest of $G \backslash (F_1 \cup F_2 \cup \ldots \cup F_{i-1})$. Let G_i be the multigraph $(V, F_1 \cup F_2 \cup, \ldots, F_i)$. Nagamochi and Ibaraki [17] give a linear time algorithm that given a graph G with n nodes and m edges computes a msfd of order m in time $O(m+n)$. We will refer to this algorithm as the *decomposition algorithm (DA)*. This algorithm also determines an linear order on the nodes, called the *maximum cardinality search order (mcs-order)*.

An edge (x, y) is *contracted* if x is identified with y and all self-loops are discarded. A contraction reduces the number of nodes in G, but does not reduce the size of the minimum edge cut.

In the following we make repeatedly use of the following facts.

Lemma 3.1 [17] *If x and y are connected F_k, then x and y are k-edge connected in G.*

Lemma 3.2 [12] *If a n-node graph is k-edge connected, then it contains at least $(k/2)n$ edges.*

4 Incremental algorithms for the minimum edge cut

4.1 An incremental $(2+\epsilon)$-approximation

In this section we present an incremental algorithm that maintains a $(2+\epsilon)$-approximation of the minimum cut in amortized time $O(1/\epsilon^2)$ per insertion. We denote by G_0 the initial graph and by m_0 the number of edges in the initial graph.

The basic idea for testing fast if the current solution is still correct is as follows: If k is the current solution, i.e. $k/(2+\epsilon) \leq \lambda \leq k$, we repeatedly compute a msfd and contract all edges in F_i for $p = \lceil k/(2+\epsilon) \rceil$ until F_p (and all F_j for $j > p$) does not contain any edges. Let n' be the number of nodes and m' be the number of edges in the resulting graph. Since F_i is empty, $m' < (i-1)n'$, which implies that $\lambda \leq k$ (Lemma 3.2). This shows that k is a $(2+\epsilon)$-approximation of λ until $m' = (k/2)n'$. Thus, after computing n', we know that the current solution will be correct for the next $(k/2)n' - (p-1)n' = \Omega(\epsilon k n')$ insertions.

To compute a new solution k efficiently using previous solutions we contract all edges that belonged to F_i in a previous msfd.

The details are given below. We denote by $G' = (V', E')$ with $n' = |V'|$ and $m' = |E'|$ the graph resulting from the contractions.

An Incremental $(2+\epsilon)$-Approximation Algorithm

1. $\epsilon' = \epsilon/4$, $j = 0$.
   Compute a $(2+\epsilon')$-approximation k of λ using the static algorithm and a msfd $F_1^{(0)}, \ldots, F_m^{(0)}$ of $G' = (V', E')$ of order m using DA.
2. $N = \emptyset$;

Finally we combine the previous two (deterministic) algorithms with random sampling to achieve an incremental Monte Carlo algorithm that maintains a $(1+\epsilon)$-approximation of the minimum edge cut with high probability. The total expected time for m_1 insertions is $O((m_0+m_1)(\log\lambda)((\log n)/\epsilon)^2)$. Thus, the amortized expected time per insertion is $O((\log\lambda)(\log n/\epsilon)^2)$. This technique is the same as used by Karger [14] in his static algorithm which takes time $O(m+n((\log n)/\epsilon)^3)$ for computing a $(1+\epsilon)$-approximation.

There are two previous results on maintaining the size of the minimum edge cut under edge insertions: (1) Karger [13] gives an algorithm, which maintains a $\sqrt{1+2/\epsilon}$-approximation in expected time $O(n^\epsilon)$ per insertion. Thus, to achieve a $2+\epsilon$ approximation this algorithm takes time $O(n^{2/(3+4\epsilon+\epsilon^2)})$ per insertion and to achieve a $(1+\epsilon)$-approximation it takes time $O(n^{1/\epsilon})$. Note that our algorithm for a $(1+\epsilon)$-approximation gives an exponential improvement in the running time. Our algorithm for the $(2+\epsilon)$-approximation takes constant time and is deterministic. (2) Dinitz and Vainshtein [4] give an $O(\min(mn,kn^2)+u\alpha(u,n))$-time algorithm for maintaining all minimum cuts between any two nodes of a subset S of V under a sequence of u insertions that do not change the size k of the minimum cut between two nodes in S. Note that k can be larger than λ since only minimum cuts between nodes in S are considered. Their data structure answers queries that ask if two given nodes of S are separated by a cut of size k in amortized time $O(\alpha(u,n))$. Our data structure requires $S=V$, i.e. it answers queries if two nodes are separated by a cut of size λ in amortized time $O(\alpha(m,n))$, but it allows the size of the minimum cut to change.

We also give new results for the minimum vertex cut in a graph. We present a static algorithm that computes a 2-approximation of the size κ of the minimum vertex cut in time $O(n^2\min(\sqrt{n},\kappa))$. To be precise, let δ be the minimum degree in G. Note that $\kappa\leq\delta$. Our algorithm computes the exact size of κ if $\kappa\leq\delta/2$ and it returns $\delta/2$ if $\kappa>\delta/2$. This is a speed-up of a factor at least κ over the fastest exact algorithm for computing κ, which takes time $O(\kappa^2n^2+\kappa^3n^{1.5})$ [10, 1, 17]. Using the new 2-approximation algorithm as subroutine gives an incremental algorithm with total time $O(m_0+\kappa n^2/\epsilon+m_1n)$. Since $\kappa n=O(m_0+m_1)$, the amortized time per insertion is $O(n/\epsilon)$ if the initial graph is empty.

Section 2 presents the basic structure that is common to all incremental algorithms in this paper. In Section 3 we give some basic definitions. Section 4 presents the incremental algorithms for the minimum edge cut, Section 5 gives the results for the minimum vertex cut.

2 A generic incremental algorithm

To maintain the exact or approximate minimum edge or vertex cut in a graph the following generic algorithm is used.

1. Compute the solution in the initial graph using the static algorithm.
2. **while** the current solution is correct **do**
   if the new operation is a query **then**
   output the current solution
   else /*the new operation is an insertion */
   check if the current solution is still correct.
   endwhile
3. Compute a new solution using previous solutions.
   Goto 2.

1 Introduction

Computing the connectivity of a graph is a fundamental problem with applications to chip design, system reliability, and communications networks. Since in many of these applications the underlying graph can change incrementally it is important to efficiently maintain the connectivity of the graph during these changes. Two vertices of a graph G are *k-edge connected* (*k-vertex connected*) if there are k edge-disjoint (vertex-disjoint) paths between them. The graph G is *k-edge-connected* (*k-vertex connected*) if all its vertices are k-edge-connected (*k-vertex connected*). A *minimum edge cut* (*minimum vertex cut*) of a graph G is a minimum cardinality set of edge (vertices) of G whose removal disconnects G. If the minimum edge cut has size λ, then there exists a pair of vertices that are not $(\lambda+1)$-edge-connected, and, thus, G is not $(\lambda+1)$-edge-connected. Hence, determining the size of the minimum cut of a graph is equivalent to computing the edge connectivity of G. Computing the cardinality of the minimum vertex cut κ is equivalent to computing the vertex connectivity of G. An integer k is a c-approximation of λ (κ) if $\lambda \leq k \leq c\lambda$ ($\kappa \leq k \leq c\kappa$).

This paper presents simple algorithms for maintaining the exact and approximate size of the minimum edge cut and the approximate size of the minimum vertex cut of a graph. The basic idea is to use the static algorithm for computing the solution and to build a data structure that quickly tests if the solution computed last by the static algorithm is still the correct solution for the current graph. If yes, the data structure is updated, if no, the static algorithm is called. The difficulty lies in finding the appropriate data structure and to amortize the cost for the static algorithm over previous insertions. The algorithms can output the exact or approximate size k of the minimum cut in constant time and a cut of size k in time proportional of its size.

Given a static exact, $(1+\epsilon)$-approximate, or $(2+\epsilon)$-approximate minimum edge cut algorithm with running time $T(n,m)$ we convert it into a semi-dynamic algorithm such that the total time for m insertions is $O(T(n,m))$, where n is the number of nodes in the graph. Thus, the algorithms are optimal in the sense that they match the performance of the best static algorithm for the problem (up to a factor of $O(1/\epsilon)$). The overhead of making the algorithm incremental contributes only a constant factor to the running time.

In the following we denote by m the total number of edges in the graph, consisting of m_0 initial edges and m_1 inserted edges, by n the number of nodes of G, by λ the size of the minimum edge cut in the final graph, and by κ the size of the minimum vertex cut. Note that $\kappa \leq \lambda$ and $\lambda n = O(m_0 + m_1)$.

We give first an algorithm that maintains a $(2+\epsilon)$-approximation of λ in total time $O(m_0/\epsilon + (m_1+\lambda n)/\epsilon^2)$. This algorithm is an incremental version of Matula's static algorithm that computes a $(2+\epsilon)$-approximation of λ in time $O(m/\epsilon)$ [16]. Thus, after a preprocessing step with running time $O(m_0/\epsilon)$ the amortized time per operation is $O(1/\epsilon^2)$.

Next we use Gabow's (exact) minimum edge cut algorithm [8] as a subroutine. This algorithm computes the size of the minimum edge cut, all minimum edge cuts in G, and a cactus-tree representation in time $O(m + \lambda^2 n \log(n^2/m))$. Our incremental algorithm takes time $O(\lambda(m_0+m_1) + \lambda^2 n \log n)$. Thus, if the algorithm starts with an empty graph, the amortized time per insertion is $O(\lambda \log n)$. Our algorithm can answer queries that ask if two given nodes are separated by a cut of size λ in amortized time $O(\alpha(m,n))$.

Even if λ is a constant, the running time of our incremental algorithm is close to the running time of the best "special purpose" algorithms: Determining if G is 1-edge-connected, 2-edge-connected, or 3-edge-connected takes amortized time $O(\alpha(m,n))$ per insertion (see [18, 11, 15]).

INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Approximating Minimum Cuts under Insertions

Monika Rauch Henzinger *

TR-94-063

(Extended Abstract)

Abstract

This paper presents insertions-only algorithms for maintaining the exact and approximate size of the minimum edge and vertex cut of a graph. The algorithms are optimal in the sense that they match the performance of the best static algorithm for the problem. We first give an incremental algorithm that maintains a $(2+\epsilon)$-approximation of the size minimum edge cut in amortized time $O(1/\epsilon^2)$ per insertion and $O(1)$ per query. Next we show how to maintain the exact size λ of the minimum edge cut in amortized time $O(\lambda \log n)$ per operation. Combining these algorithms with random sampling finally gives a randomized Monte-Carlo algorithm that maintains a $(1+\epsilon)$-approximation of the minimum edge cut in amortized time $O((\log \lambda)((\log n)/\epsilon)^2)$ per insertion.

Finally we present the first 2-approximation algorithm for the size κ of the minimum vertex cut in a graph. It takes time $O(n^2 \min(\sqrt{n}, \kappa))$. This is an improvement of a factor of κ over the time for the best algorithm for computing the exact size of the minimum vertex cut, which takes time $O(\kappa^2 n^2 + k^3 n^{1.5})$. We also give the first algorithm for maintaining a $(2+\epsilon)$-approximation of the minimum vertex cut under insertions. Its amortized insertion time is $O(n/\epsilon)$. The algorithms output the approximate or exact size k in constant time and a cut of size k in time linear in its size.

*Department of Computer Science, Cornell University, Ithaca, NY 14853. Email: mhr@cs.cornell.edu. This research was done while visiting at the International Computer Science Institute, Berkeley, CA.